Day 11 - 21st June 2025

File Handling

Streams

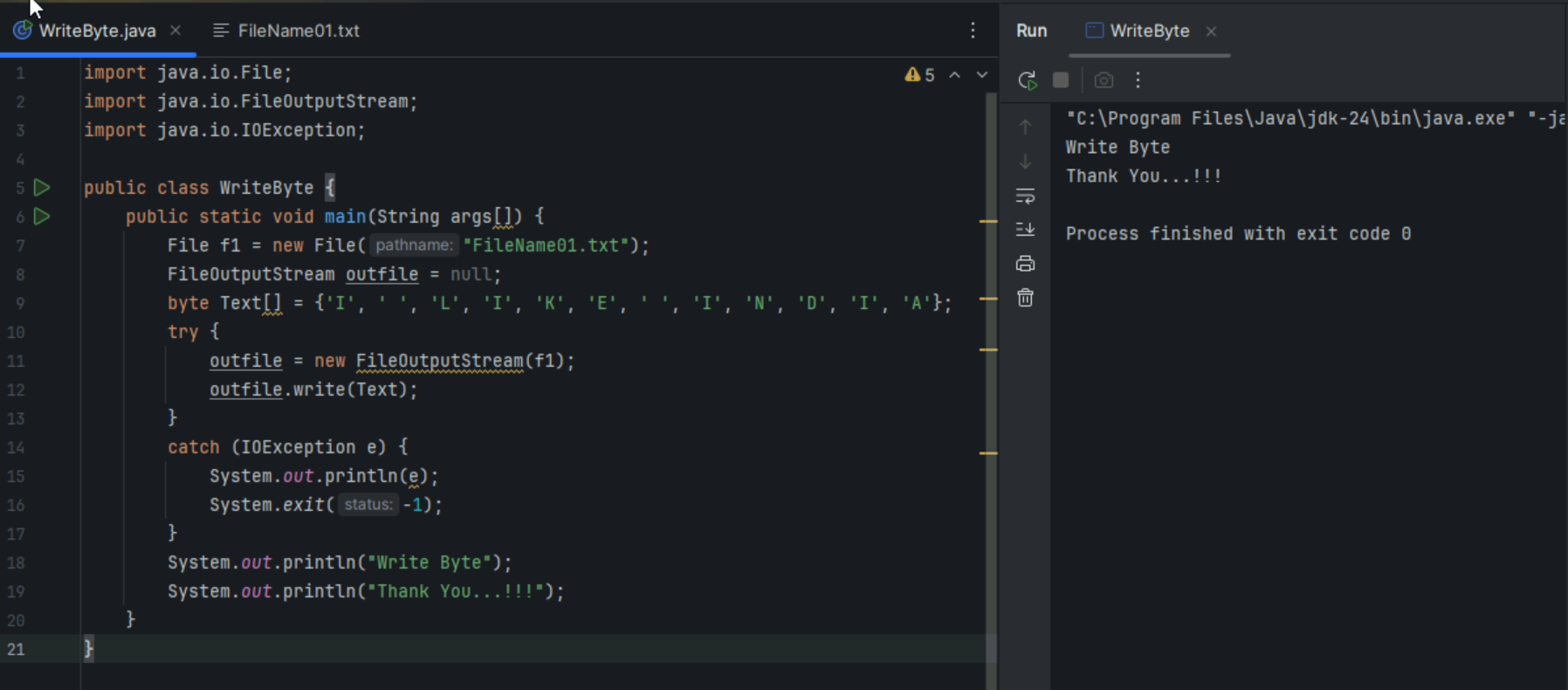
========================================================================

File handling:

Task 1:

Run the below code and see the file with the given name created or not..

Run it again with “I like India” instead of “I love India”.. And see the file …



import java.io.File;

import java.io.FileOutputStream;

import java.io.IOException;

public class WriteByte {

public static void main(String args[]) {

// Line 1

File f1 = new File("FileName01.txt");

// Line 2

FileOutputStream outfile = null;

// Line 3

byte Text[] = {'I', ' ', 'L', 'O', 'V', 'E', ' ', 'I', 'N', 'D', 'I', 'A'};

// Line 4

try {

// Line 5

outfile = new FileOutputStream(f1);

// Line 6

outfile.write(Text);

}

// Line 7

catch (IOException e) {

System.out.println(e);

System.exit(-1);

}

// Line 8

System.out.println("Write Byte");

System.out.println("Thank You...!!!");

}

}

Line 1: File f1 = new File("FileName01.txt");

It creates a File object in your program's memory. By using the new keyword to call the File class constructor with the desired filename. To have a reference or a "pointer" to the file you intend to work with on your computer's disk. Important: This line does not create the actual physical file on the disk. It just creates the idea of the file in Java.

we use this at the beginning of any file operation to specify which file you are targeting.

Line 2: FileOutputStream outfile = null;

It declares a variable named outfile that can hold a FileOutputStream object. It's initialized to null (nothing). By stating the type (FileOutputStream) and name (outfile).

It's declared here, outside the try block, so that it could theoretically be accessed in other parts of the method, like a finally block (though one isn't used here).

This is a common pattern in older Java code. A more modern approach is to declare it inside a try-with-resources statement (more on this later).

Line 3: byte Text[] = {'I', ' ', ... };

It creates an array of bytes. A byte is the most fundamental unit of data that computers work with.By manually listing each character. The computer stores each character as a number (a byte).

FileOutputStream is a low-level stream that works directly with bytes, not text. So, you must convert your text into its raw byte representation before you can write it. (A more common way to do this is String myString = "I LOVE INDIA"; byte[] Text = myString.getBytes();)

We create your data before you attempt to write it to the file.

Line 4: try { ... }

This starts a try block. It tells Java: "Try to execute the following code, but be prepared for things to go wrong."Using the try keyword. File operations are risky! The disk could be full, you might not have permission to write the file, or the drive could be disconnected. A try block lets you handle these potential errors gracefully instead of crashing.

We should always wrap risky operations like file I/O (Input/Output) in a try...catch block.

Line 5: outfile = new FileOutputStream(f1);

This is the most important line. It creates the FileOutputStream object, which is the "pipe" that connects your program to the physical file on the disk. It calls the FileOutputStream constructor, passing in the File object we created earlier. To establish the connection for writing. Crucially, this action opens the file.

If FileName01.txt does not exist, it will be created.

If FileName01.txt already exists, its contents will be erased (this is why your text was overwritten!).

This is the first step inside your try block, as you need the pipe before you can send data through it.

Line 6: outfile.write(Text);

It takes the byte array (Text) and sends it through the stream (outfile) into the file.By calling the write() method on the FileOutputStream object.This is the actual action of writing your data.

After the stream has been successfully opened and you have the data ready.

Line 7: catch (IOException e) { ... }

This block "catches" any IOException that might have been thrown from inside the try block.Using the catch keyword.To handle errors. If writing failed, the code inside catch will run. Here, it prints the error message and exits the program.

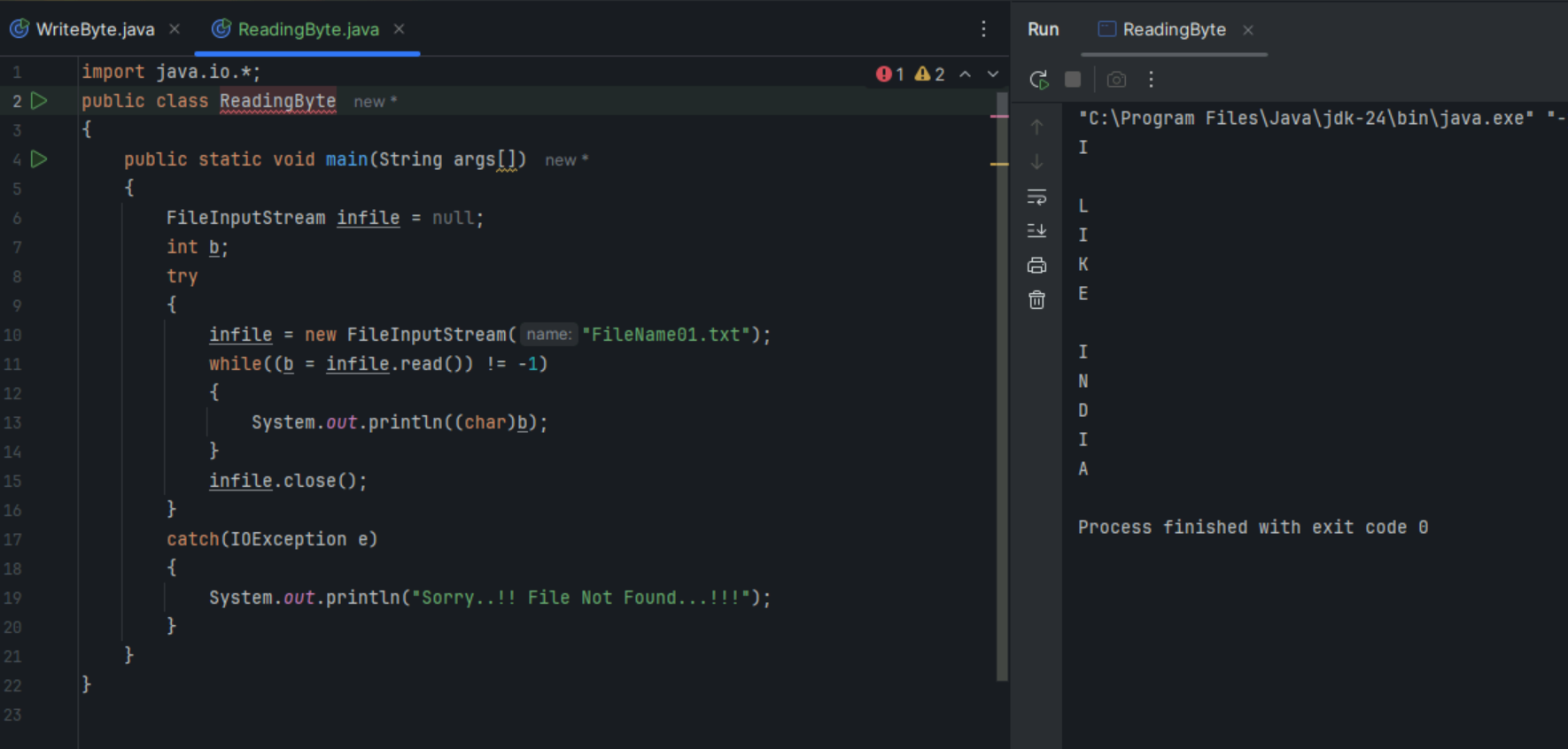
It immediately follows a try block to define the error-handling plan.

Line 8: System.out.println(...)

Prints messages to the console.Standard Java print statements.To give the user feedback that the program seems to have finished its task successfully.

We see these messages only if the try block completes without any errors.

Task 2:



import java.io.\*;

public class ReadingByte {

public static void main(String args[]) {

// Line 1

FileInputStream infile = null;

// Line 2

int b;

// Line 3

try {

// Line 4

infile = new FileInputStream("FileName01.txt");

// Line 5

while ((b = infile.read()) != -1) {

// Line 6

System.out.println((char) b);

}

// Line 7

infile.close();

}

// Line 8

catch (IOException e) {

// Line 9

System.out.println("Sorry..!! File Not Found...!!!");

}

}

}

Line 1: FileInputStream infile = null;

It declares a variable named infile that is capable of holding a FileInputStream object (our "input pipe"). It's initialized to null to begin with.By stating the object type and a variable name.

To have a variable ready to hold the input stream once we create it inside the try block. This is an older Java pattern. We declare the stream variable before you attempt to use it, typically right at the start of the method.

Line 2: int b;

It declares an integer variable named b.Using the int keyword. This variable will be used to temporarily hold each byte of data as it is read from the file. Crucially, it's an int and not a byte because the read() method needs a way to signal the end of the file. It does this by returning the special value -1. A byte variable cannot hold the value -1 in the same way, so an int is used to accommodate both the data (0-255) and the end-of-file signal (-1).Before the loop where you start reading data.

Line 3: try { ... }

Starts a try block, which is a safety net for code that might fail.Using the try keyword. Reading a file is risky. The file might not exist (FileNotFoundException), or the disk could have an error. This block allows you to handle these IOExceptions without crashing.Always wrap your file reading/writing logic in a try...catch block.

Line 4: infile = new FileInputStream("FileName01.txt");

This creates the actual FileInputStream object and connects it to the file on the disk. It "opens the input pipe."By calling the FileInputStream constructor with the name of the file you want to read.To establish the stream of data flowing from the file to your program.

This is the first step inside the try block. If this line fails (e.g., the file isn't found), the code will immediately jump to the catch block.

Line 5: while ((b = infile.read()) != -1)

This is the heart of the program—a loop that continues as long as there is data to be read. Let's break it down from the inside out:infile.read(): Reads the next single byte of data from the file. It returns this byte as an integer.

b = ...: The integer value of the byte is assigned to our variable b.

(...) != -1: The entire expression is then compared to -1. If the read() method returns -1, it means we have reached the end of the file, the condition becomes false, and the loop stops.It cleverly combines reading, assigning, and checking in a single, classic Java I/O line.

To process the entire file one piece at a time until it's completely finished. This is very memory-efficient, as you never have to load the whole file into memory at once.We use this loop structure to read from any input stream until it is exhausted.

Line 6: System.out.println((char) b);

It takes the integer b (which holds a byte value) and prints it to the console as a character.Using a type cast. (char) tells Java, "Treat this number not as a number, but as the character it represents." For example, if b is 73, (char)b is I.Because infile.read() gives you the raw numeric value of the character. To see it as a human-readable letter, you must convert it.Inside the while loop, to process each byte as it's read.

Line 7: infile.close();

It closes the input stream. By calling the close() method.This is critical! It releases the file from your program's control, freeing up system resources. If you don't close streams, your application can suffer from "resource leaks."After you are completely finished reading from the file. (Note: In a better design, this would be in a finally block or handled automatically by try-with-resources).

Line 8 & 9: catch (IOException e) { ... }

This block runs only if an error occurred inside the try block.It "catches" the IOException object that describes the error.To inform the user that something went wrong (in this case, by printing a "File Not Found" message) instead of letting the program crash.Immediately after the try block.

Task 3:

import java.io.\*;

import java.util.\*;

public class WriteByte\_1 {

public static void main(String args[]) {

// Line 1

FileOutputStream outfile = null;

// Line 2 (commented out, but important to understand)

// String s=args[0];

// Line 3

Scanner sc = new Scanner(System.in);

// Line 4

String s = sc.nextLine();

// Line 5

byte b1[] = s.getBytes();

// Line 6

try {

// Line 7

outfile = new FileOutputStream("FileName02.txt");

// Line 8

outfile.write(b1);

}

// Line 9

catch (IOException e) {

System.out.println(e);

System.exit(-1);

}

// Line 10

System.out.println("Write Byte");

System.out.println("Thank You...!!!");

}

}

Line 1: FileOutputStream outfile = null;

Declares a variable outfile to hold our file output "pipe," initializing it to nothing (null). By stating the type (FileOutputStream) and the variable name.

Why: To prepare a variable that can be used to connect our program to a file for writing. This is an older style; modern Java would often declare this inside a try-with-resources statement.

When/Where: At the beginning of the method to make the variable available throughout its scope.

Line 2: //String s=args[0];

What: This is a commented-out line. If it were active, it would take input from the command line when you first run the program (e.g., java WriteByte\_1 "some text here"). args is an array of strings that holds command-line arguments. args[0] would get the first argument.It shows an alternative, non-interactive way to get input. The programmer chose the Scanner method instead for a more user-friendly, interactive experience.This is one way to provide input to a Java program, but it's less common for programs that need to "chat" with the user.

Line 3: Scanner sc = new Scanner(System.in);

This creates a Scanner object, a powerful and easy-to-use tool for reading input.It creates a new Scanner and tells it to listen to System.in. System.in is the standard input stream, which is almost always the keyboard/console.To build a bridge between the user's keyboard and our program. Think of the Scanner as the program's "ear."You create a Scanner whenever you want your program to pause and wait for the user to type something.

Line 4: String s = sc.nextLine();

This is the line that actually reads the user's input.It calls the nextLine() method on the Scanner object. The program will stop and wait at this line until the user types something and presses the Enter key. Whatever the user typed is then stored as a String in the variable s.To capture a full line of text from the user to be processed or stored.Directly after creating the Scanner, when you are ready to receive user input.

Line 5: byte b1[] = s.getBytes();

It converts the user's text (a String) into an array of raw bytes.By calling the .getBytes() method on the String object.FileOutputStream is a low-level stream that only understands how to write bytes, not complex objects like Strings. We must convert our text into this fundamental format first.After we have the final string you want to save and just before you intend to write it to a file.

Line 6: try { ... }

Starts a try block, a "safety zone" for code that might fail.Using the try keyword.File operations can fail for many reasons (e.g., no disk space, no permissions). This lets us handle those failures without crashing the program.Always wrap file I/O operations inside a try...catch block.

Line 7: outfile = new FileOutputStream("FileName02.txt");

Opens the connection to the file on the disk.By creating a new FileOutputStream object pointed at the file FileName02.txt.This creates the file if it doesn't exist. Crucially, if the file does exist, it wipes it clean (overwrites it), ready for the new content.Inside the try block, before you start writing data.

Line 8: outfile.write(b1);

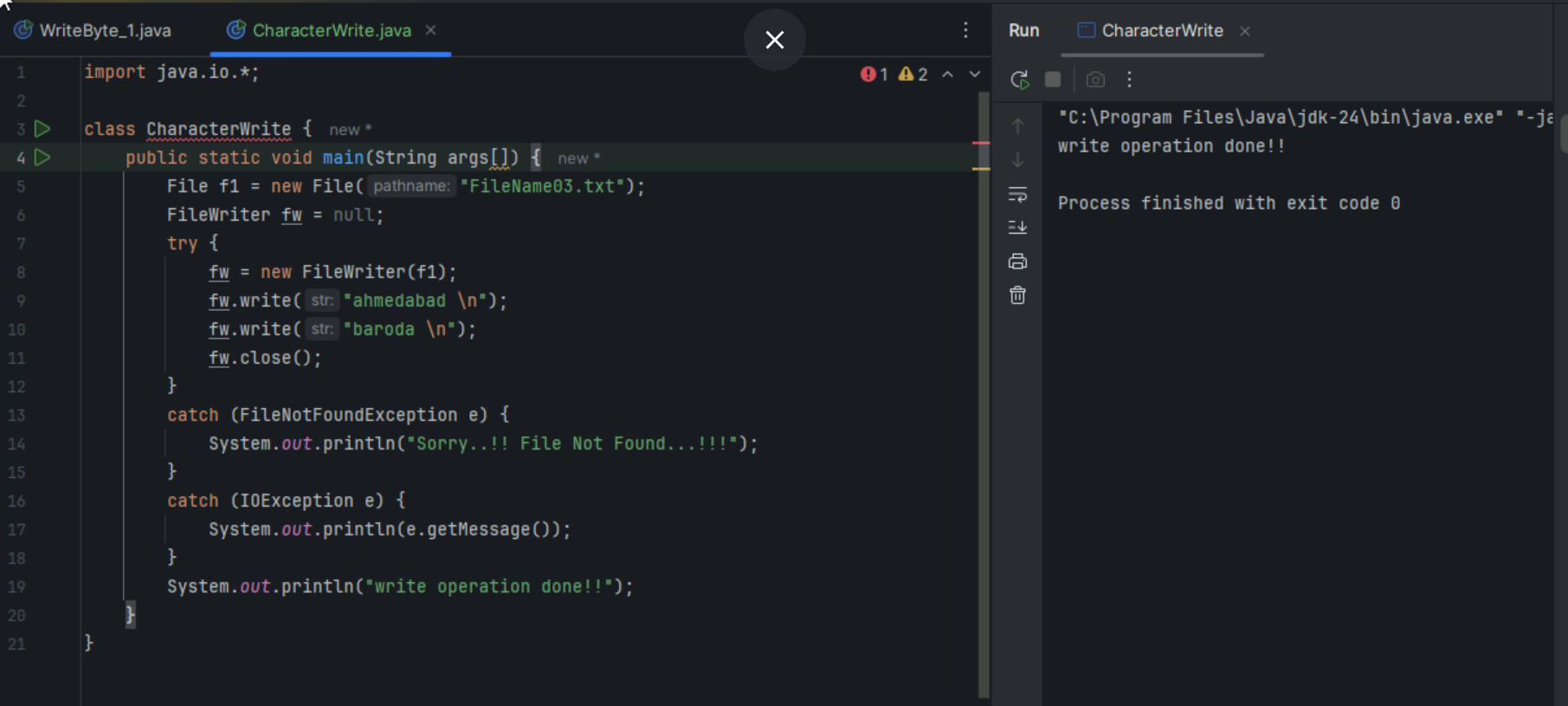
Writes the byte array (containing the user's input) into the file.By calling the write() method on the FileOutputStream object.This is the action that transfers the data from your program's memory to long-term storage on the disk.After the stream has been successfully opened.

Line 9: catch (IOException e) { ... }This is the error-handling block. It runs only if something went wrong inside the try block. It "catches" the IOException that was thrown.To manage failures gracefully. In this case, it prints the error and then calls System.exit(-1), which is a "hard stop" that immediately terminates the entire program. Immediately following the try block.

Line 10: System.out.println(...)

Prints feedback messages to the console.Using standard Java print statements.To let the user know that the program has finished its writing task successfully.This code is only reached if the try block completes without any errors.

Task 4:



import java.io.\*;

class CharacterWrite {

public static void main(String args[]) {

// Line 1

File f1 = new File("FileName03.txt");

// Line 2

FileWriter fw = null;

// Line 3

try {

// Line 4

fw = new FileWriter(f1);

// Line 5

fw.write("ahmedabad \n");

// Line 6

fw.write("baroda \n");

// Line 7

fw.close();

}

// Line 8

catch (FileNotFoundException e) {

System.out.println("Sorry..!! File Not Found...!!!");

}

// Line 9

catch (IOException e) {

System.out.println(e.getMessage());

}

// Line 10

System.out.println("write operation done!!");

}

}

Line 1: File f1 = new File("FileName03.txt");

Creates a File object in memory, representing the file FileName03.txt on your disk.By using the new keyword to instantiate the File class.To give your program a target to write to. This object acts as a path or a pointer to the actual file.This is the first step before you can perform any operations (like writing) on a specific file.

Line 2: FileWriter fw = null;

Declares a variable named fw that can hold a FileWriter object.By stating the type (FileWriter) and name (fw) and initializing it to null.To prepare a variable that will later hold the "smart secretary" or character-writing stream. This older style allows the variable to be visible outside the try block.At the start of the method, before you intend to create and use the FileWriter.

Line 3: try { ... }Starts a "safety zone" for code that might fail.With the try keyword.

File operations are inherently risky. The try...catch structure ensures that if something goes wrong (like a disk error), the program handles it gracefully instead of crashing.Always wrap your file I/O code within a try...catch block.

Line 4: fw = new FileWriter(f1);

This creates the FileWriter object and links it to the file on disk.By calling the FileWriter constructor with the File object as its target.This opens the connection for writing. Like FileOutputStream, it will create the file if it doesn't exist, and it will erase and overwrite the file if it does exist. This is the "smart secretary" getting a blank piece of paper ready.Inside the try block, before you start writing.

Lines 5 & 6: fw.write("ahmedabad \n"); and fw.write("baroda \n");

These lines write the actual text strings into the file.By calling the write() method. Notice you can pass a String directly! You do not need to call .getBytes(). The \n is the special "newline" character that tells the file to move to the next line.This is the primary advantage of FileWriter. It simplifies writing text. The writer handles the character-to-byte conversion automatically behind the scenes.After the FileWriter is open and you have text to save. You can call write() multiple times to add more content.

Line 7: fw.close();

Closes the FileWriter.By calling the close() method on the writer object.This is extremely important. It does two things:

Flushes the buffer: The writer might keep some text in a temporary memory buffer for efficiency. close() ensures everything in the buffer is written to the disk.Releases the file: It tells the operating system that your program is done with the file, freeing up resources.After you have finished all your writing operations.

Line 8: catch (FileNotFoundException e) { ... }

A specific error handler for when a file cannot be found.By catching the FileNotFoundException.To handle the specific case of a missing file. However, for FileWriter, this is less likely to happen on creation because it creates the file if it's missing. This would be more relevant if you were trying to write to a file in a directory that doesn't exist.After the try block, to handle a specific type of I/O error.

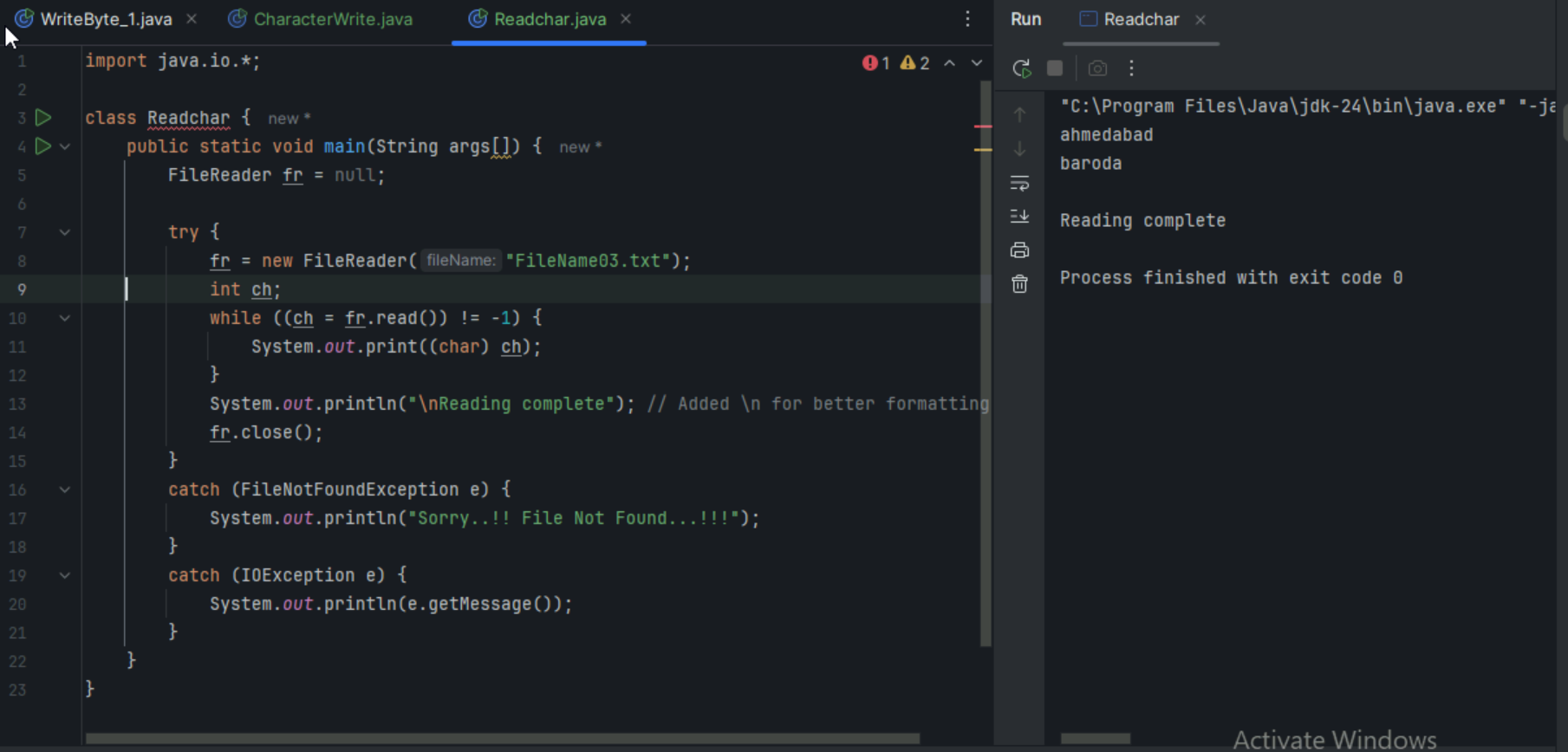
Line 9: catch (IOException e) { ... }

A more general error handler for any other Input/Output problems.By catching the broader IOException.To catch other potential issues like "disk is full" or "access denied." Since FileNotFoundException is a type of IOException, this block would catch it too if the first catch block wasn't there.After more specific catch blocks to act as a catch-all for other related errors.

Line 10: System.out.println("write operation done!!");

A confirmation message printed to the console.A standard print statement.To provide feedback to the user that the program completed its task without errors.This line is only reached if the try block executes successfully.

Task 5:



import java.io.\*;

class Readchar {

public static void main(String args[]) {

// Line 1

FileReader fr = null;

// Line 2

try {

// Line 3

fr = new FileReader("FileName03.txt");

// Line 4

int ch;

// Line 5

while ((ch = fr.read()) != -1) {

// Line 6

System.out.print((char) ch);

}

// Line 7

System.out.println("\nReading complete"); // Added \n for better formatting

// Line 8

fr.close();

}

// Line 9

catch (FileNotFoundException e) {

System.out.println("Sorry..!! File Not Found...!!!");

}

// Line 10

catch (IOException e) {

System.out.println(e.getMessage());

}

}

}

Line 1: FileReader fr = null;

Declares a variable fr that can hold a FileReader object, which is our "character-reading pipe."By stating the type (FileReader) and initializing the variable to null.To prepare a variable that will hold our character stream. This is an older Java style used to make the variable accessible outside the try block (e.g., for closing).At the beginning of a method where you intend to read characters from a file.

Line 2: try { ... }Starts the "safety zone" for our file operation.Using the try keyword.Reading a file is an operation that can easily fail (file not found, disk error, no permissions). The try...catch structure allows us to handle these failures gracefully.Always wrap file I/O operations in a try...catch block.

Line 3: fr = new FileReader("FileName03.txt");This creates the FileReader object and connects it to the physical file on the disk. It "opens the character-reading pipe."By calling the FileReader constructor with the filename.To establish the stream of characters flowing from the file to your program. Crucially, if FileName03.txt does not exist, this line will fail and throw a FileNotFoundException. This is the first step inside the try block, as you can't read from a stream that isn't open.

Line 4: int ch;Declares an integer variable ch.Using the int keyword.This variable will hold each character as it's read from the file. It is an int (not a char) because the read() method uses the special value -1 to signal the end of the file, and an int is needed to hold this value.Inside the try block, just before you start the reading loop.

Line 5: while ((ch = fr.read()) != -1)

This is the core reading loop that processes the file character by character until it's empty.fr.read(): Reads the next single character from the file, returning its numeric value.

ch = ...: Assigns that numeric value to the ch variable.

(...) != -1: Compares the result to -1. When read() returns -1, it signifies the end of the stream, making the condition false and stopping the loop.To read the entire file efficiently without loading it all into memory at once. It's the standard, classic way to process a file character by character in Java.

This loop structure is the primary mechanism for reading from any Reader in Java.

Line 6: System.out.print((char) ch);

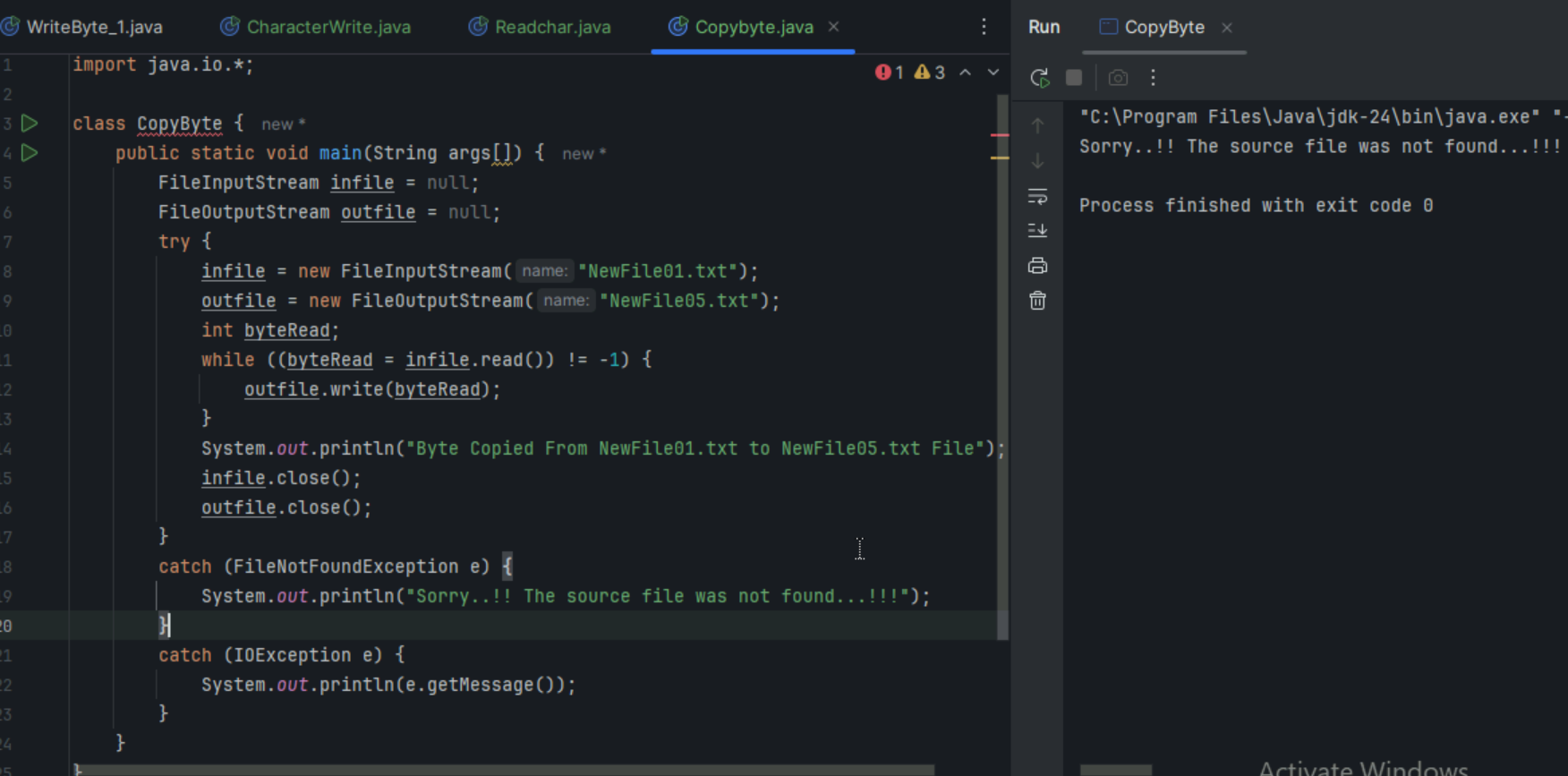
Takes the numeric value in ch and prints its character representation to the console.By using a type cast. (char) tells Java to interpret the number as a character (e.g., interpret 97 as a). The use of print instead of println keeps all the characters on the same line.To display the data in a human-readable format. fr.read() gives you a number; (char) turns it back into a letter for you to see.Inside the while loop to process each character as it arrives from the stream.

Line 7: System.out.println("\nReading complete"); A confirmation message printed to the console. A standard print statement.To provide feedback to the user that the program has finished reading the file.After the reading loop is complete.

Line 8: fr.close(); Closes the FileReader stream.By calling the close() method.This is essential for releasing the file from your program's control and freeing up operating system resources. Failing to close streams can lead to resource leaks.After you are completely finished with the file.

Lines 9 & 10: catch BlocksThese are the error handlers. The first catches the specific FileNotFoundException, and the second catches any other general IOException.Using the catch keyword to define what to do if an error is thrown in the try block.To provide a user-friendly error message instead of letting the program crash with a long, confusing stack trace.Immediately following the try block.

Task 6:



import java.io.\*;

class CopyByte {

public static void main(String args[]) {

FileInputStream infile = null;

FileOutputStream outfile = null;

try {

infile = new FileInputStream("NewFile01.txt");

outfile = new FileOutputStream("NewFile05.txt");

int byteRead; while ((byteRead = infile.read()) != -1) { outfile.write(byteRead);

}

System.out.println("Byte Copied From NewFile01.txt to NewFile05.txt File");

infile.close();

outfile.close();

}

catch (FileNotFoundException e) {

System.out.println("Sorry..!! The source file was not found...!!!");

} catch (IOException e) {

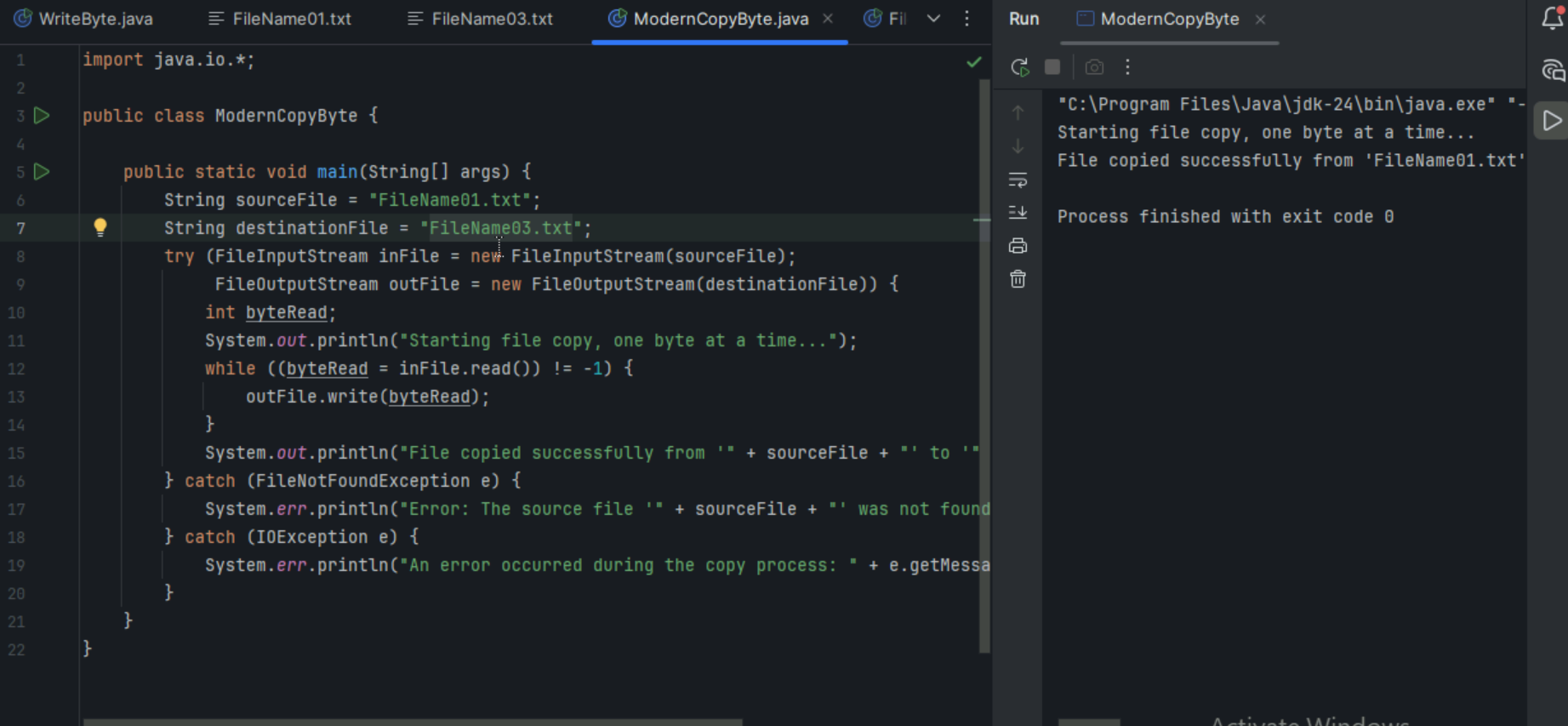
System.out.println(e.getMessage());

}

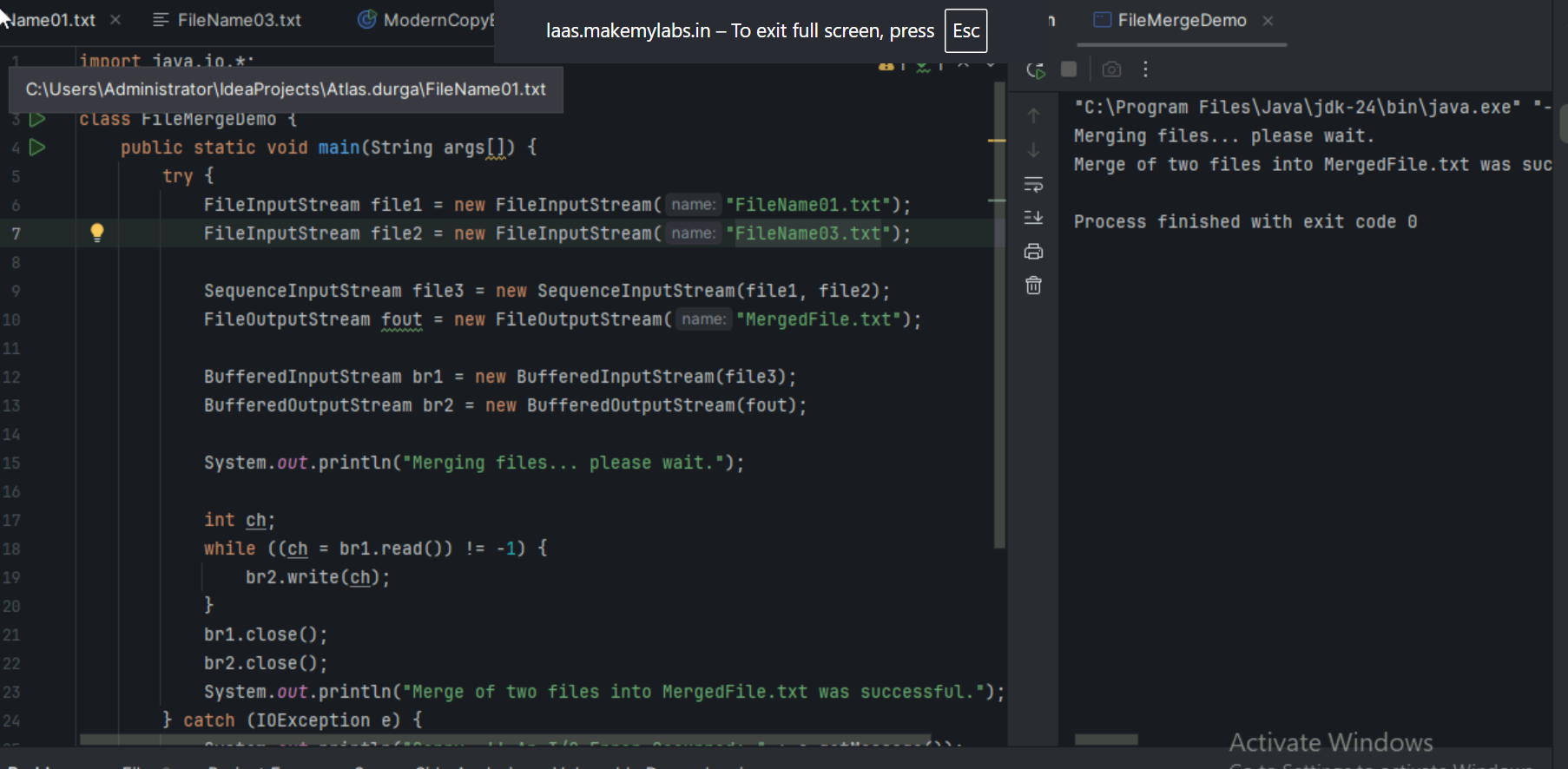
}

}

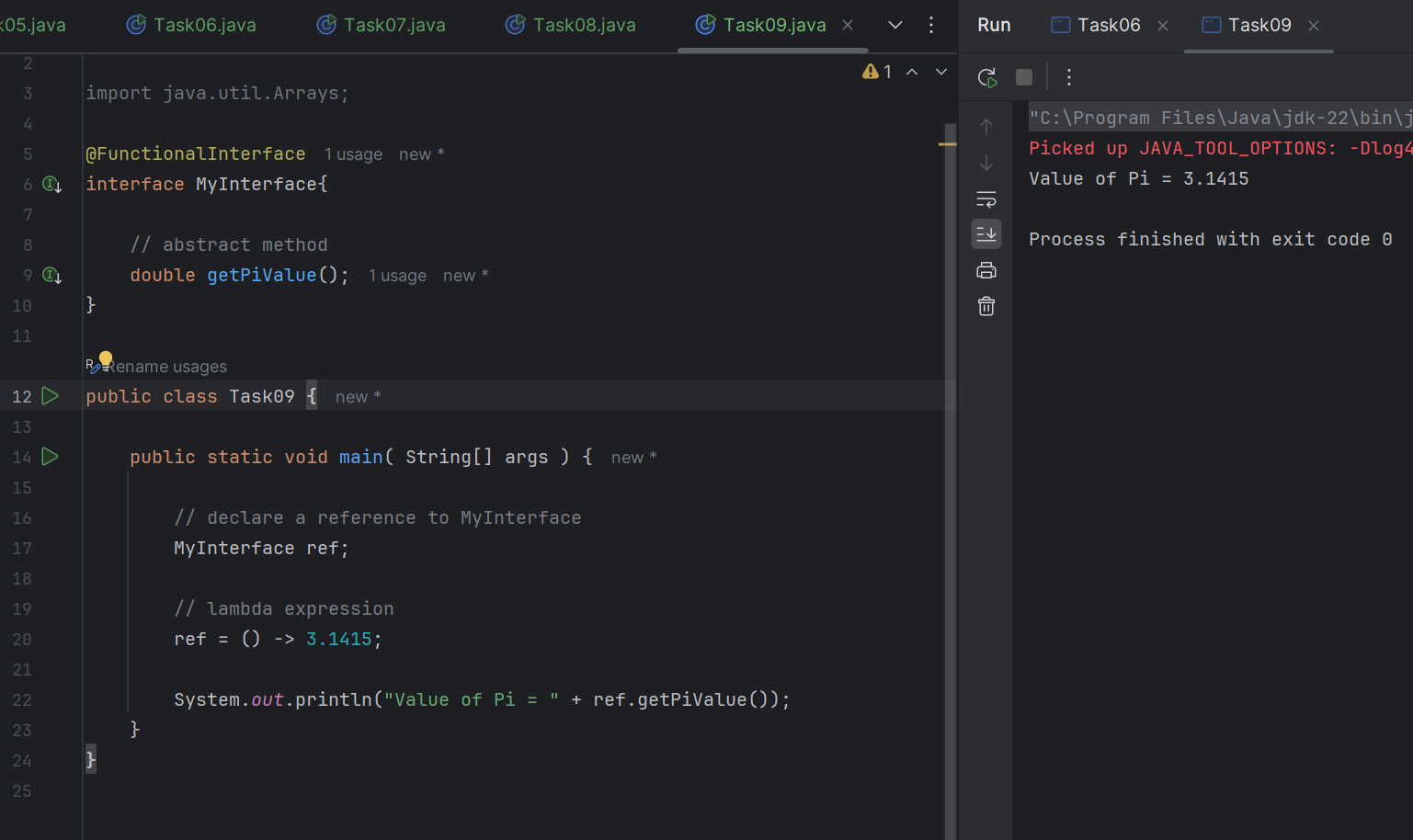
Task 7:



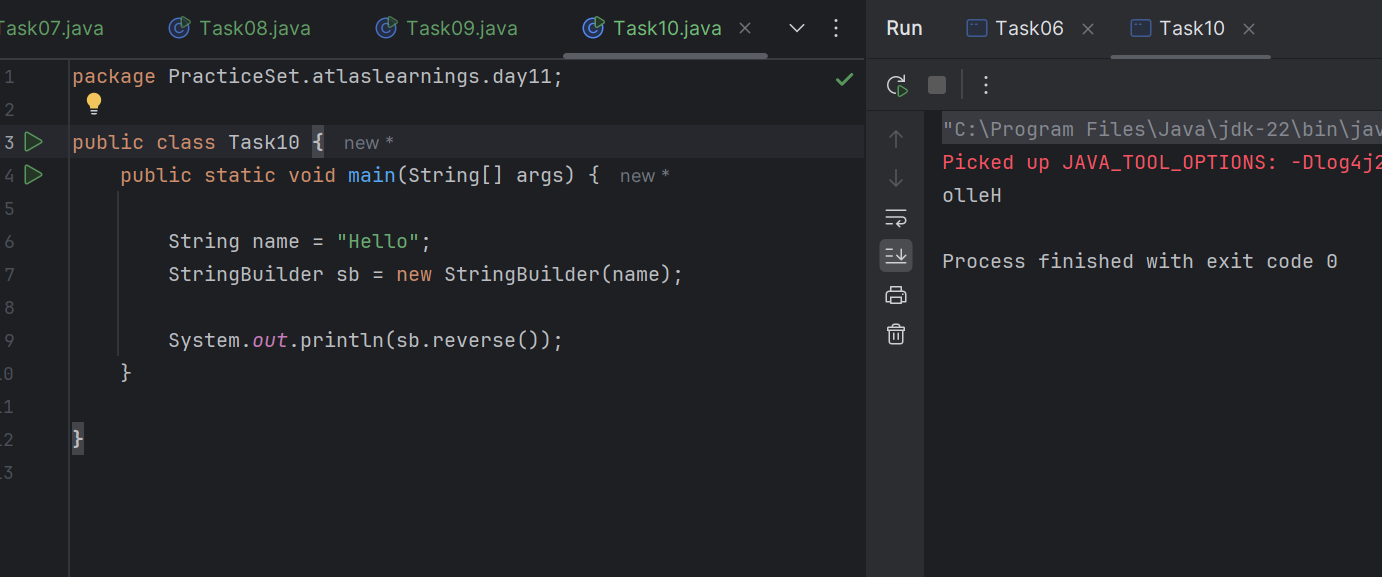
Task 8:



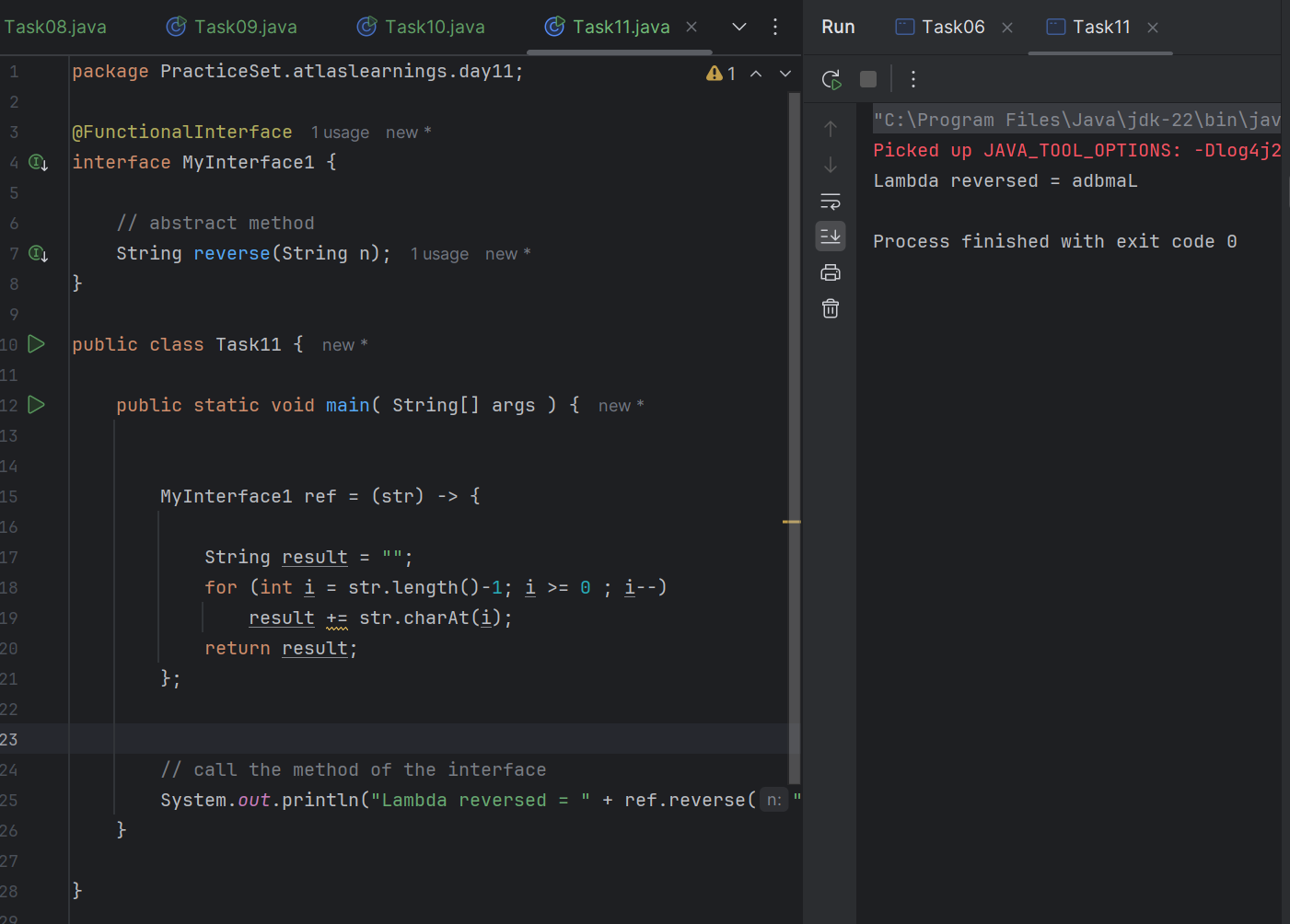
Task 09



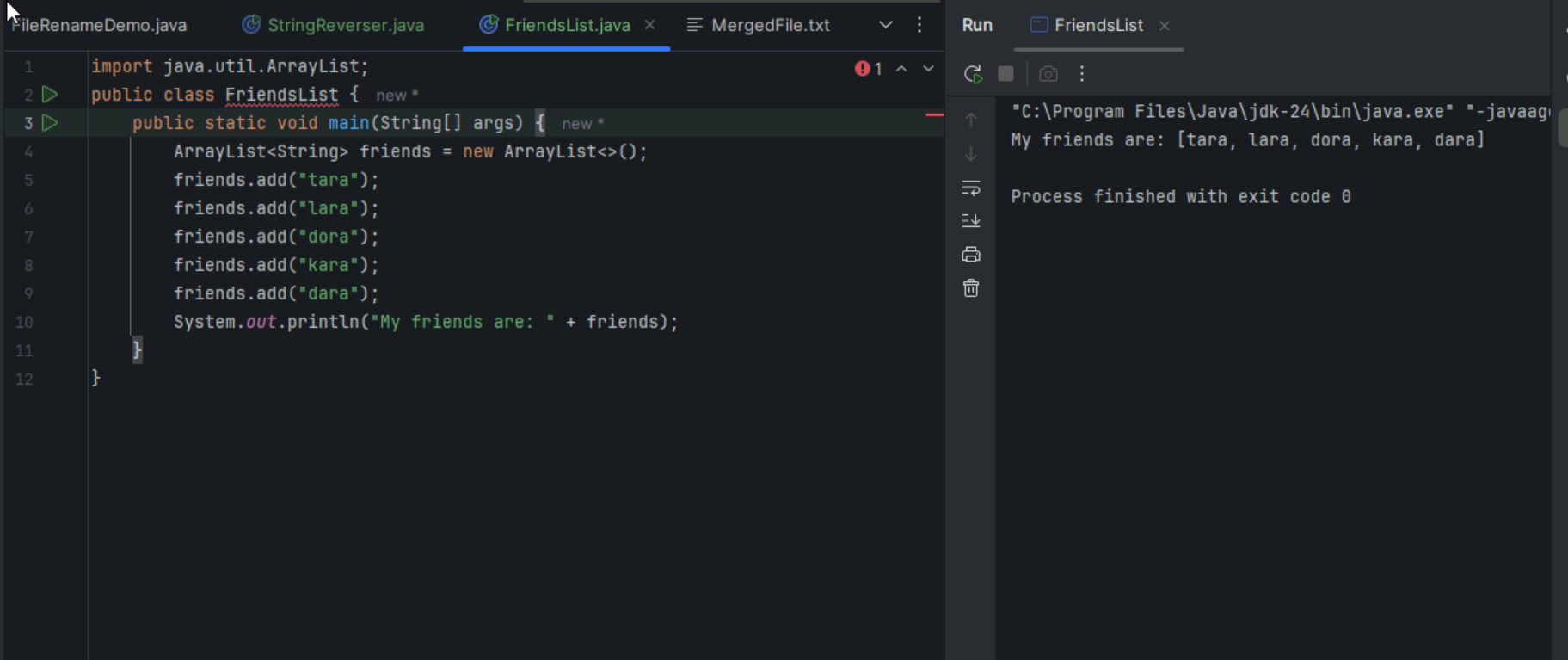
Task 10



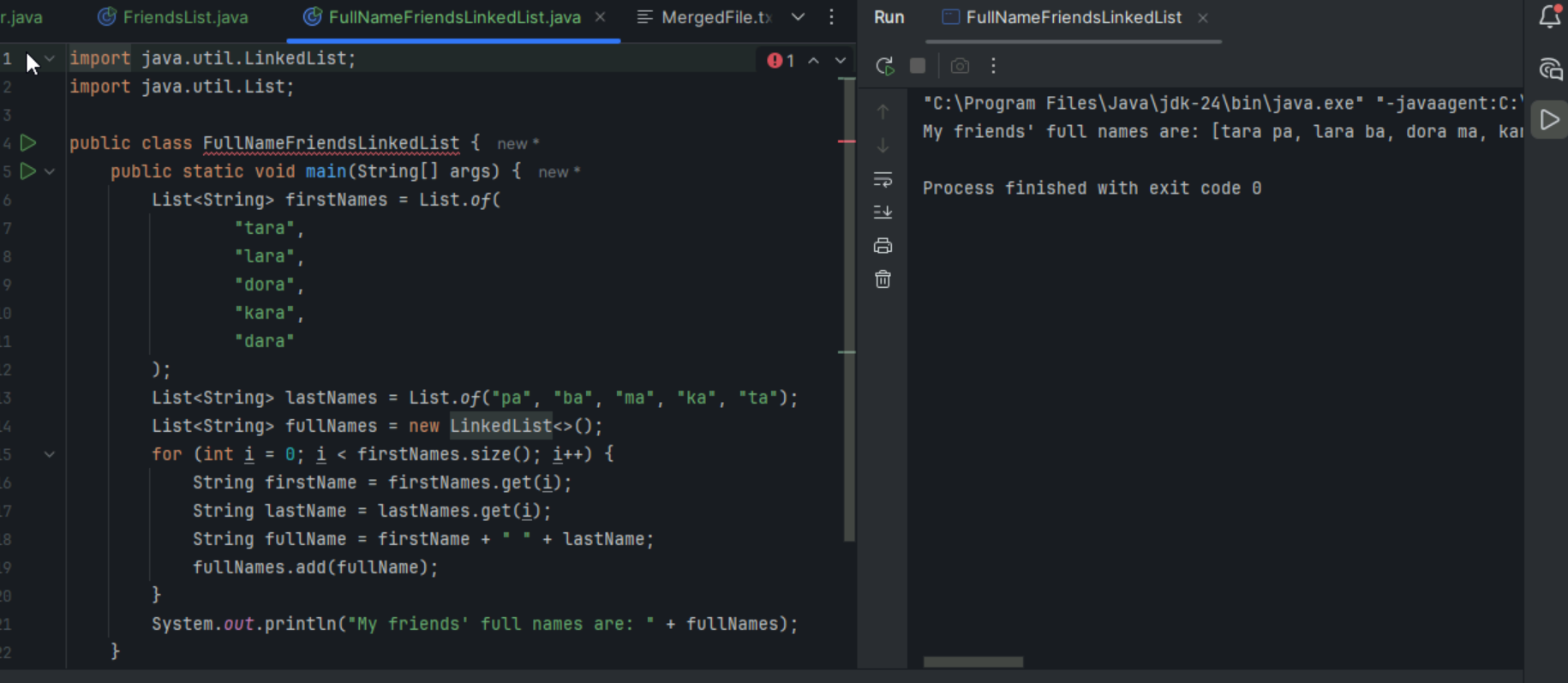
Task 11



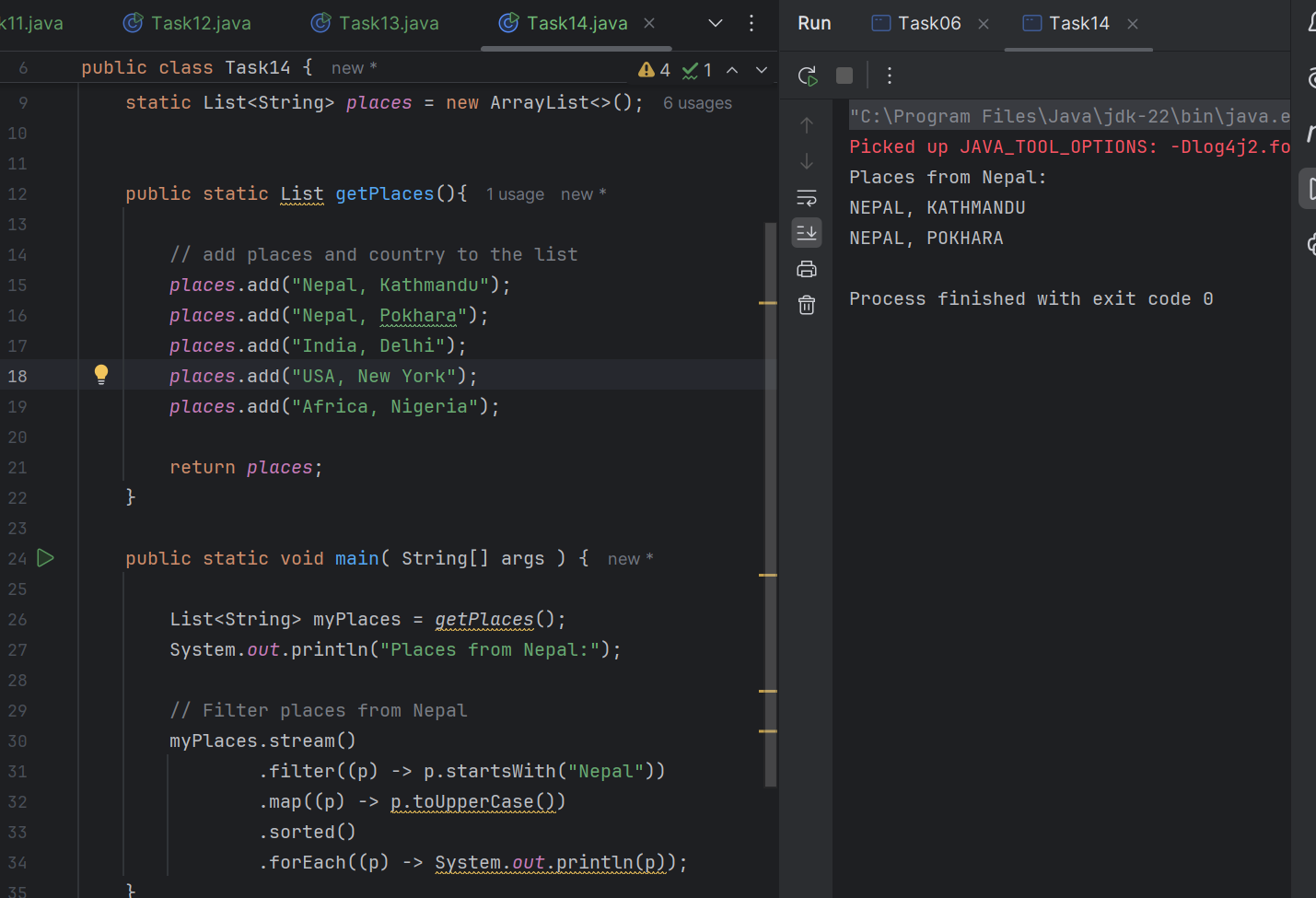
Task 12:



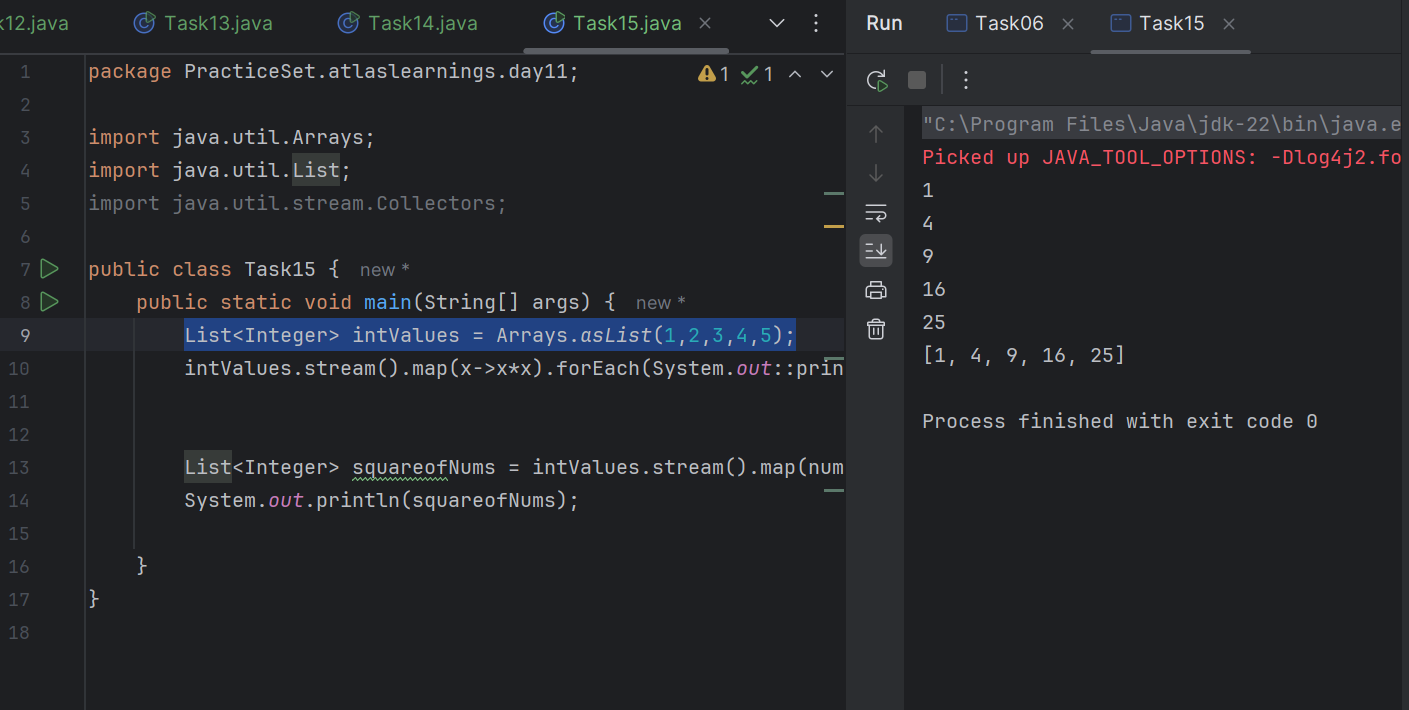
Task 13:



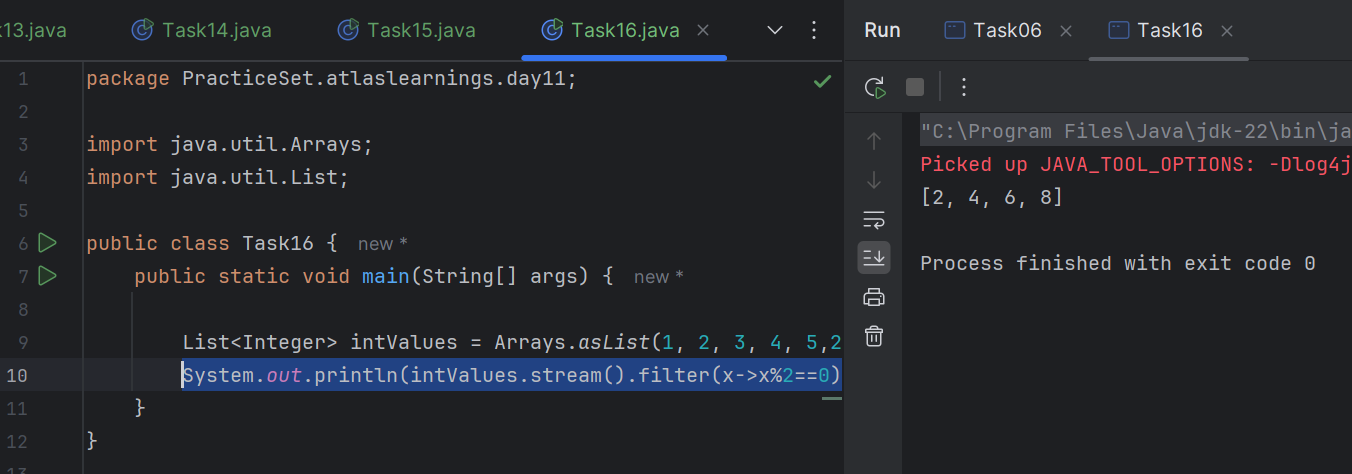
Task 14



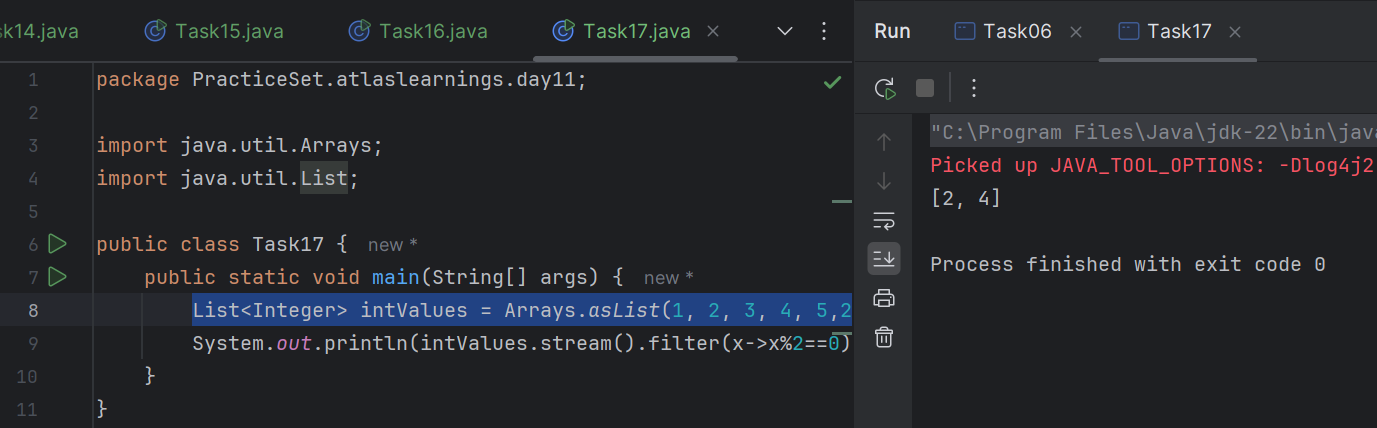
Task 15



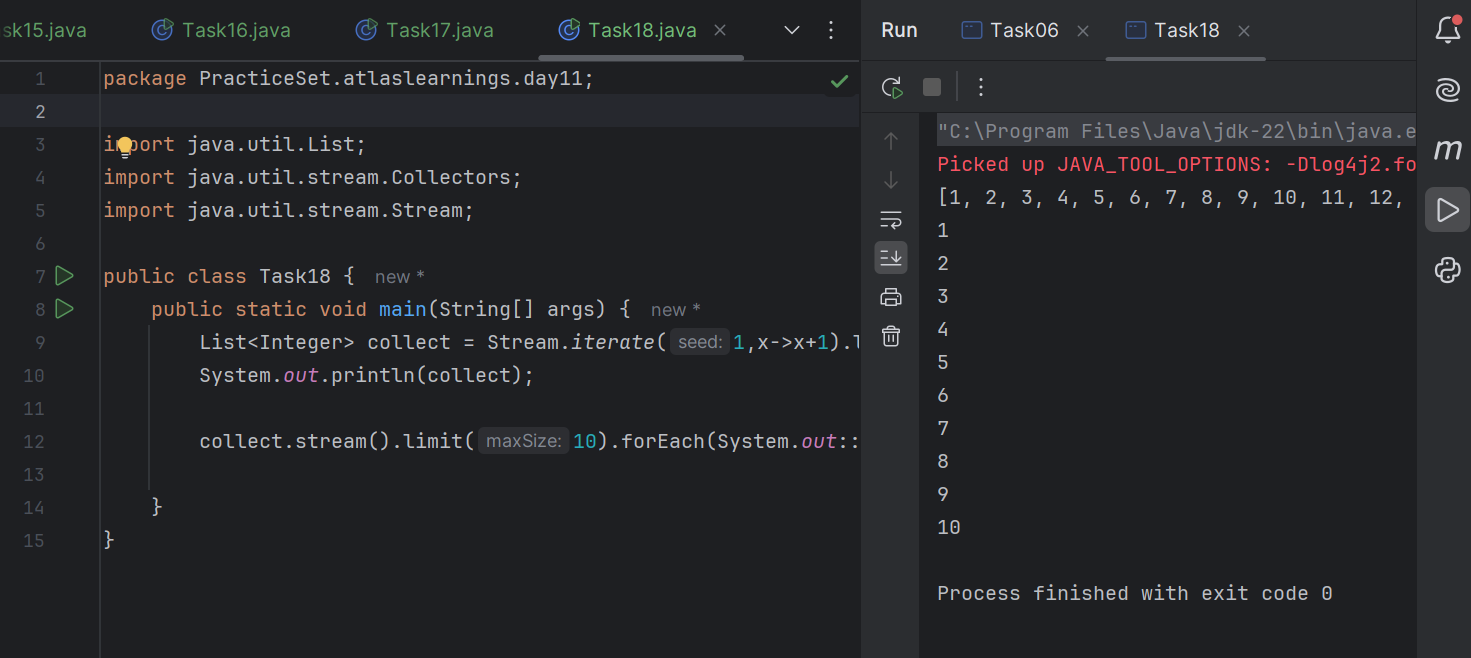
Task 16



Task 17



Task 18



Task 19

