Chapter 11

R 11.1.

When you supply the same name for the input and output files to a program in Java, the behavior depends on how you handle file input and output within your program. The consequences may include overwriting the existing file, loss of data, or potential errors.

Overwriting Data: If you open a file for writing (output) using the same file name as the input file, the contents of the existing file will be overwritten. This is because opening a file for writing typically truncates it to zero length if it already exists.

Potential Data Loss: If your program reads data from the file and then writes to the same file without preserving the original content, you might lose the data present in the file.

R 11.2

To add the total to an existing file without overwriting its content, you can follow these steps. The key is to read the existing content, calculate the total, and then append the total to the end of the file. Here's a pseudocode solution:

1. Open the existing file for reading.

2. Read the existing content from the file.

3. Close the file.

4. Perform the necessary calculations to get the total.

5. Open the same file for writing in append mode.

6. Write the total to the end of the file.

7. Close the file.

Pseudocode:

```
// Step 1: Open the existing file for reading
file = open("your_file.txt", "r")
```

// Step 2: Read the existing content from the file

existing_content = read_all_content(file)

// Step 3: Close the file

close(file)

// Step 4: Perform necessary calculations to get the total

total = calculate_total(existing_content)

// Step 5: Open the same file for writing in append mode

file = open("your_file.txt", "a")

// Step 6: Write the total to the end of the file

write_to_file(file, "Total: " + total)

// Step 7: Close the file

close(file)

R 11.3

Whenever the user tries to access a file for reading and if the the file doesn't exist then an exception is thrown.And, whenever a user tries to access a file, which doesn't exist, to write on it, then a new file is created with length 0

If you try to open a file for reading that doesn't exist:

- **Using `FileInputStream` or `BufferedReader`:** You will get a `FileNotFoundException`. This exception indicates that the file you are trying to open is not found.

try {

```
    FileInputStream fileInputStream = new FileInputStream("nonexistentfile.txt");
    // ...
} catch (FileNotFoundException e) {
    e.printStackTrace();
}
```

If you try to open a file for writing that doesn't exist:

- **Using `FileOutputStream` or `BufferedWriter`:** The file will be created, and you can start writing to it. If the file already exists, its contents will be truncated (cleared), and the new data will be written.

```
try {
    FileOutputStream fileOutputStream = new FileOutputStream("newfile.txt");
    // ...
} catch (IOException e) {
    e.printStackTrace();
}
```

R 11.4

If you try to open a file for writing, but the file or device is write-protected (read-only), you will encounter an **`IOException`** when attempting to write to the file. The exact exception message may vary, but it generally indicates that the operation is not permitted due to the file being read-only.

```
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;

public class WriteToFile {
    public static void main(String[] args) {
        try {
            // Attempting to open a write-protected file for writing
```

```
            BufferedWriter writer = new BufferedWriter(new FileWriter("readonlyfile.txt"));

            // Writing some data to the file
            writer.write("Hello, World!");

            // Closing the writer
            writer.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

R 11.5

When you write to a `PrintWriter` without closing it, the data you've written may not be flushed to the underlying stream or file. Flushing is the process of writing any buffered data to the destination. If the `PrintWriter` is not closed or flushed explicitly, there's a risk of losing data because the data may be kept in the buffer without being written to the file or stream.

Here's an example test program in Java that demonstrates the potential data loss:

```
import java.io.FileWriter;
import java.io.PrintWriter;
import java.io.IOException;

public class PrintWriterTest {
    public static void main(String[] args) {
```

```
        try {
            // Create a PrintWriter without auto-flush
            PrintWriter printWriter = new PrintWriter(new FileWriter("output.txt"));

            // Write some data to the file
            printWriter.println("This data may be lost!");

            // The PrintWriter is not closed or flushed

            // Program exits without ensuring the data is written

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

R 11.6

In Java, when specifying file paths, you need to be aware that backslashes (`\`) are escape characters in string literals. To represent a backslash in a file path, you should use a double backslash (`\\`) or use a forward slash (`/`).

For example, to open a file with the path `c:\temp\output.dat`, you can do the following:

```
String filePath = "c:\\temp\\output.dat"; // Using double backslashes
```

R 11.7

To read numbers from a file while skipping over markers such as "N/A," you can use a `Scanner` or `BufferedReader` in Java and implement logic to handle the special case when the marker is encountered.

R 11.8

In the command `java Woozle -Dname=piglet -I\eeyore -v heff.txt a.txt lump.txt`, the arguments passed to the `main` method of the `Woozle` class will be as follows:

- `args[0]`: "-Dname=piglet"
- `args[1]`: "-Ieeyore" (Note: The backslash `\` before `eeyore` is not necessary)
- `args[2]`: "-v"
- `args[3]`: "heff.txt"
- `args[4]`: "a.txt"
- `args[5]`: "lump.txt"

If you print these values in the `main` method, you would see something like:

```
public class Woozle {
    public static void main(String[] args) {
        for (int i = 0; i < args.length; i++) {
            System.out.println("args[" + i + "]: " + args[i]);
        }
    }
}
```

Output:

```
args[0]: -Dname=piglet
args[1]: -Ieeyore
args[2]: -v
args[3]: heff.txt
```

args[4]: a.txt

args[5]: lump.txt

R 11.9

Throwing an Exception:

- **Definition:** Throwing an exception refers to the act of signaling that an exceptional condition or error has occurred during the execution of a program.
- **Syntax:** In Java, exceptions are thrown using the `throw` keyword followed by an instance of a class that extends `Throwable` (usually a subclass of `Exception` or `Error`).
- **Purpose:** Throwing an exception is used to indicate that something unexpected or erroneous has happened in the code, and the normal flow of the program cannot continue.

Catching an Exception:

- **Definition:** Catching an exception involves handling or dealing with an exception that has been thrown by surrounding the code that might throw an exception with a try-catch block.
- **Syntax:** In Java, catching an exception is done using the `try` and `catch` blocks. The `try` block contains the code that might throw an exception, and the `catch` block specifies how to handle the exception.
- **Purpose:** Catching an exception allows the program to gracefully respond to exceptional situations, preventing the program from terminating abruptly. It enables developers to handle errors in a controlled manner.

R 11.10

Checked exceptions are exceptions that are checked at compile time. The compiler ensures that the code handles or declares the exception, either by using a try-catch block to handle it or by declaring the exception using the `throws` keyword in the method signature.

`IOException` is a common checked exception. It is thrown when there is an error during input/output operations, such as reading from or writing to a file.

Unchecked exceptions, also known as runtime exceptions, are exceptions that are not checked at compile time. They typically represent programming errors or exceptional conditions that might occur at runtime.

`ArithmeticException` is a common unchecked exception. It is thrown when an arithmetic operation, such as division by zero, is performed.

In summary:

- **Checked Exception:** Checked at compile time, and you are required to handle or declare them using the `throws` keyword. Example: `IOException`.
- **Unchecked Exception (Runtime Exception):** Not checked at compile time. Example: `ArithmeticException`.
- **Using `throws`:** Required for checked exceptions in the method signature.

R 11.11

The `IndexOutOfBoundsException` is an unchecked exception, specifically a subclass of `RuntimeException`. Unchecked exceptions, including `IndexOutOfBoundsException`, are not required to be declared in the method signature using the `throws` keyword. This is because unchecked exceptions represent errors that typically result from programming mistakes or unforeseen runtime conditions.

R 11.12

When a `throw` statement is encountered, the specified exception is created, and control is immediately transferred to the nearest applicable `catch` block.

The runtime searches for a `catch` block that can handle the thrown exception. It looks for the nearest applicable `catch` block in the current method or in any calling method up the call stack.

R 11.13

If an exception is thrown, but there is no matching `catch` clause to handle that specific type of exception, the program's normal flow is disrupted, and the runtime searches for an appropriate exception handler. If the exception is not caught in the current method, it propagates up the call stack to the calling method and continues this process until it finds a suitable `catch` block or until it reaches the top-level of the program.

R 11.14

When a `catch` clause is executed in response to an exception, it receives an exception object that contains information about the exception. This object, typically of type `Exception` or a subtype, can be used by your program to gather information about the exception and potentially take appropriate actions.

R 11.15

No, the type of the exception object is not always the same as the type declared in the catch clause that catches it. In Java, polymorphism allows you to catch exceptions using a broader type (usually a superclass) than the actual type of the thrown exception. This provides flexibility in handling different types of exceptions in a more generalized way.

R 11.16

The `try-with-resources` statement in Java is used to automatically close resources, such as files, sockets, or database connections, when they are no longer needed. It was introduced in Java 7 to simplify the management of resources that require explicit closing.

The syntax for `try-with-resources` is as follows

```
try (resourceType resource1 = initialization; resourceType resource2 = initialization) {
    // Code that may throw exceptions
```

```
} catch (ExceptionType e) {
    // Exception handling
}
```

R 11.17

When an exception is thrown in the `try` block of a `try-with-resources` statement, and the `close` method of a resource also throws an exception, the exception thrown during the resource closing process takes precedence. The exception thrown by the `close` method is the one that gets propagated, and the original exception from the `try` block is suppressed.

Here's an example to demonstrate this behavior

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class TryWithResourcesExceptionExample {

    public static void main(String[] args) {
        // Specify a non-existent file path to intentionally cause an IOException
        String filePath = "nonexistent.txt";

        try {
            // Using try-with-resources with a BufferedReader
            try (BufferedReader br = new BufferedReader(new FileReader(filePath))) {
                // Attempt to read from a non-existent file
                String line = br.readLine();
```

```
            System.out.println(line); // This line won't be reached due to the IOException
        } catch (IOException e) {
            // This catch block handles the IOException thrown in the try block
            System.err.println("IOException caught: " + e.getMessage());
            // Rethrow the caught exception
            throw e;
        } finally {
            // This finally block will execute even if an exception is thrown in the try block
            System.out.println("Finally block executed");
        }
    } catch (Exception e) {
        // This catch block handles the exception thrown during resource closing
        System.err.println("Exception caught during resource closing: " + e.getMessage());
        // Access suppressed exceptions (the original IOException)
        for (Throwable suppressed : e.getSuppressed()) {
            System.err.println("Suppressed exception: " + suppressed.getMessage());
        }
    }
  }
}
```

R 11.18

The `next()` and `nextInt()` methods of the `Scanner` class in Java can throw different exceptions, and these exceptions are checked exceptions. Here are the details:

1. `next()` **Method:**
   - **Exceptions:**
     - `NoSuchElementException`: If no more tokens are available.

    - `IllegalStateException`: If the scanner is closed.
- **Checked Exception:**
    - Both `NoSuchElementException` and `IllegalStateException` are checked exceptions.

2 . `nextInt()` Method:

- **Exceptions:**
    - `InputMismatchException`: If the next token does not match the Integer regular expression, or is out of range for the data type.
    - `NoSuchElementException`: If no more tokens are available.
    - `IllegalStateException`: If the scanner is closed.
- **Checked Exception:**
    - `NoSuchElementException` and `IllegalStateException` are checked exceptions.
    - `InputMismatchException` is a runtime exception (unchecked).

R 11.19

If a program reads a file containing the values 1, 2, 3, 4 using a `Scanner` and the `nextInt()` method, and the file structure matches the expected format, the program will likely proceed without errors. However, if the file structure does not match the expected format (e.g., if there are non-integer values or the file ends prematurely), the program may throw an `InputMismatchException` or `NoSuchElementException`.

To improve the program and provide a more accurate error report, you can catch these exceptions and handle them appropriately

By catching and handling these exceptions, the program can provide more informative error messages, making it easier to diagnose and address issues with the input .

R 11.20

If the `readFile` method itself is responsible for throwing a `NullPointerException`, it would typically happen within the method's implementation.

R 11.21

In the provided code, the `PrintWriter` is opened for writing to a file, and then an attempt is made to close it within a `try-catch` block. The `close` method is called both in the `try` block and the `catch` block. However, this approach has a potential disadvantage, especially when dealing with exceptions.

If the `PrintWriter` constructor or the `close` method throws an exception, it could lead to a situation where the `close` method in the `catch` block is called on a potentially uninitialized or partially initialized object. This scenario arises because if an exception occurs during the constructor, the `out` reference may not have been assigned a valid `PrintWriter` object.

Chapter 21(advance input/output)

R 21.1

Input Stream:

- An input stream is a sequence of data elements made available over time. It is a general concept that represents the flow of data from a source to a destination. Input streams can be associated with various sources, such as files, network connections, or user input.

Input streams are commonly used in low-level I/O operations, especially in languages like Java or C++. In Java, for instance, the `InputStream` class is part of the Java I/O (Input/Output) package and is used to read data from different sources.

Reader:

A reader is a higher-level abstraction compared to an input stream. It is typically associated with character-oriented data, reading characters and decoding them into a higher-level representation, such as strings. Readers are often used for text-based data.

Readers are frequently employed in programming languages that emphasize character-based input, like Java. In Java, the `Reader` class is part of the Java I/O package, and it is designed for reading character data from different sources. The `FileReader` class, for example, is a specific implementation of a reader that reads character data from a file.

R 21.2

1. **"output1.txt" (UTF-8):**
    - UTF-8 is a variable-width character encoding that can represent every character in the Unicode character set.
    - The text will be encoded using one or more bytes per character, depending on the character's Unicode code point.
    - The file will start with a byte order mark (BOM), which is optional but common for UTF-8.
2. **"output2.txt" (UTF-16):**
    - UTF-16 is a fixed-width character encoding where each character is represented by two bytes.
    - The text will be encoded using two bytes per character.
    - The file will start with a byte order mark (BOM) indicating the endianness of the file. For UTF-16, it can be either big-endian (BE) or little-endian (LE).

In summary, the primary difference is in the way characters are encoded and stored in the files. UTF-8 uses a variable number of bytes per character, while UTF-16 uses a fixed two bytes per character.

R 21.3

You need to open a RandomAccessFile to open a file for both reading and writing. For

example,

RandomAccessFile f = new RandomAccessFile("bank.dat", "rw");

R 21.4

If you write to a file reader, there will be a compile-time error because input files don't support output operations such as print.

In Java, the `FileReader` class is specifically designed for reading character data from a file. If you attempt to write to a `FileReader`, you will encounter a compilation error because the `FileReader` class does not provide methods for writing data to a file. It is a one-way input stream meant for reading characters.

R 21.5

Attempting to write to a `RandomAccessFile` that was opened only for reading in Java will result in a `java.io.IOException`. The `RandomAccessFile` class has modes for specifying whether the file should be opened for "read" (`"r"`), "write" (`"rw"`), "read and write" (`"rws"` or `"rwd"`), or "append" (`"rwa"`).

If you attempt to write to a `RandomAccessFile` opened in "read" mode (`"r"`), you will encounter an `IOException` with the message "Read-only file system." This is because the file is not open for writing.

R 21.6

To break the Caesar cipher, you can try all 25 keys (b...z) to decrypt the message. Look at the decryptions and see which one of them corresponds to English text.

R 21.7

If you attempt to save an object that is not serializable in Java using an `ObjectOutputStream`, a `NotSerializableException` will be thrown at runtime. This exception indicates that the class of the object does not implement the `Serializable` interface, and therefore, it cannot be serialized.

R 21.8

java.lang.Boolean

java.lang.Character

java.lang.Double

java.lang.Integer

java.lang.String

java.lang.Throwable and its subclasses, i.e., all exceptions

java.io.File

R 21.9

If you simply save the entire ArrayList, you do not have to save the number of elements. If you saved a collection manually, you'd first have to write out the length, then all entries. When reading the collection back in, you'd have to read in the length, then allocate sufficient storage, and then read and store all entries. It is much simpler to just write and read the ArrayList and let the serialization mechanism worry about saving and restoring the length and the entries.

R 21.10

Sequential access forces you to read all bytes in a file in order, starting from the first byte and progressing through the last byte. Once a byte has been read, it cannot be read again, except by closing and reopening the file. With random access, you can repeatedly select any position in the file for reading or writing.

R 21.11

The file pointer denotes the current position in a random access file for reading and

writing. You move it with the seek method of the RandomAccessFile class. You get the current position with the getFilePointer method. The position is the number of bytes from the beginning of the file, as a long integer, because files can be longer than 2GB (the longest length representable with an int).

R 21.12

In Java, you can use the `RandomAccessFile` class to move the file pointer to different positions within a file. The `seek()` method of `RandomAccessFile` is used for this purpose.

Move to the First Byte:

```
try (RandomAccessFile file = new RandomAccessFile("yourfile.txt", "rw")) {
    file.seek(0); // Move to the beginning of the file
    // Perform operations at the beginning of the file
} catch (IOException e) {
    e.printStackTrace();
}
```

Move to the Last Byte:

```
try (RandomAccessFile file = new RandomAccessFile("yourfile.txt", "rw")) {
    long fileLength = file.length();
    file.seek(fileLength - 1); // Move to the last byte
    // Perform operations at the end of the file
} catch (IOException e) {
    e.printStackTrace();
}
```

Move to the Exact Middle:

```
try (RandomAccessFile file = new RandomAccessFile("yourfile.txt", "rw")) {
    long fileLength = file.length();
```

```
    long middlePosition = fileLength / 2;
    file.seek(middlePosition); // Move to the middle of the file
    // Perform operations at the middle of the file
} catch (IOException e) {
    e.printStackTrace();
}
```

R 21.13

It is legal to move the file pointer past the end of the file. If you write to that location, the file will be enlarged. If you read from that location, an exception is thrown**.**

R 21.14

No, it's not possible to move the file pointer of `System.in` in Java. The `System.in` stream, which is an instance of `InputStream`, represents the standard input stream, typically connected to the keyboard. Input streams in Java, including `System.in`, are designed for sequential access and do not provide methods for directly moving the file pointer or seeking to specific positions within the input.

R 21.15

In the Java NIO (New I/O) package, the `Path` interface provides methods for working with file and directory paths. The `resolve` method in the `Path` interface is used to resolve one path against another. It creates a new `Path` by appending the given path to the original path.

When the argument passed to the `resolve` method is an absolute path, the method effectively discards the original path and returns the absolute path provided as the argument. In other words, if the given path is absolute, the result of `resolve` is the given absolute path itself.

R 21.16

The relativize method in the Java NIO (New I/O) Path interface is used to construct a relative path between two paths. It essentially calculates a path that, when resolved against the first path, results in the second path. In this sense, it can be considered the opposite of the resolve method.

Example 1: Files

```
import java.nio.file.Path;
import java.nio.file.Paths;

public class RelativizeExampleFiles {
    public static void main(String[] args) {
        Path basePath = Paths.get("/base/path");
        Path absoluteFilePath = Paths.get("/base/path/subfolder/file.txt");

        // Relativize file path against base path
        Path relativeFilePath = basePath.relativize(absoluteFilePath);

        System.out.println("Relative File Path: " + relativeFilePath);
    }
}
```

Example 2: Directories

```
import java.nio.file.Path;
import java.nio.file.Paths;

public class RelativizeExampleDirectories {
    public static void main(String[] args) {
        Path baseDir = Paths.get("/base/directory");
```

```
        Path subDir = Paths.get("/base/directory/subfolder");

        // Relativize subdirectory path against base directory
        Path relativeSubDir = baseDir.relativize(subDir);

        System.out.println("Relative Subdirectory Path: " + relativeSubDir);
    }
}
```

R 21.17

When using Files.walk in Java, the method yields the results in depth-first order. This means that it will traverse downward through the directories until it reaches the bottom of the tree. From there it will handle all siblings at each level as it moves through the tree in a recursive nature. Directory children are listed after their parents. Files will be listed before directories. The listing of each directory will be displayed alphabetically