UNIVERSITY OF MISKOLC
FACULTY OF MECHANICAL ENGINEERING
AND INFORMATICS

Java programming

GEIAL31A-B2a

File handling

Tamás Tompa, PhD
assistant professor
Department of Information Technology

Based on materials prepared by Miklós Szűcs

Miskolc, 2026

File handling methods

- Java provides multiple approaches for file handling, depending on file type, performance needs, and API level

- **1. Traditional I/O (java.io package)**
  - a. `File` representation
    - represents a file or directory path
    - used to create, delete, and query file properties (exists, size, permissions)
  - b. Byte Streams (Binary Data)
    - used for raw binary files (images, audio, video, ZIP, PDF)
    - `FileInputStream` / `FileOutputStream`
    - `BufferedInputStream` / `BufferedOutputStream`
    - use when working with non-text data

File handling methods

- Java provides multiple approaches for file handling, depending on file type, performance needs, and API level

- c. Character Streams (Text Data)
  - used for text files (TXT, CSV, JSON, XML)
  - recommended for large text files
  - `FileReader` / `FileWriter`
  - `BufferedReader` / `BufferedWriter`
    - efficient line-by-line reading and writing (`readLine()`)

- d. Object Streams (Serialization)
  - used for raw binary files (images, audio, video, ZIP, PDF)
  - `FileInputStream` / `FileOutputStream`
  - `BufferedInputStream` / `BufferedOutputStream`
  - use when working with non-text data

File handling methods

- Java provides multiple approaches for file handling, depending on file type, performance needs, and API level

- **2. Scanner (java.util)**
  - a. `Scanner` representation
    - simple API for reading text files
    - can parse tokens, numbers, and strings
    - slower than BufferedReader for large files

File handling methods

- Java provides multiple approaches for file handling, depending on file type, performance needs, and API level

- **3. New I/O (NIO & NIO.2 – java.nio, java.nio.file)**
  - introduced in Java 7+, this is the modern approach
  - a. `Path` and `Paths`
  - b. Files utility class (static methods for common operations)
  - c. Channels and Buffers (`FileChannel`, `ByteBuffer`)
    - used for high-performance and large-file operations

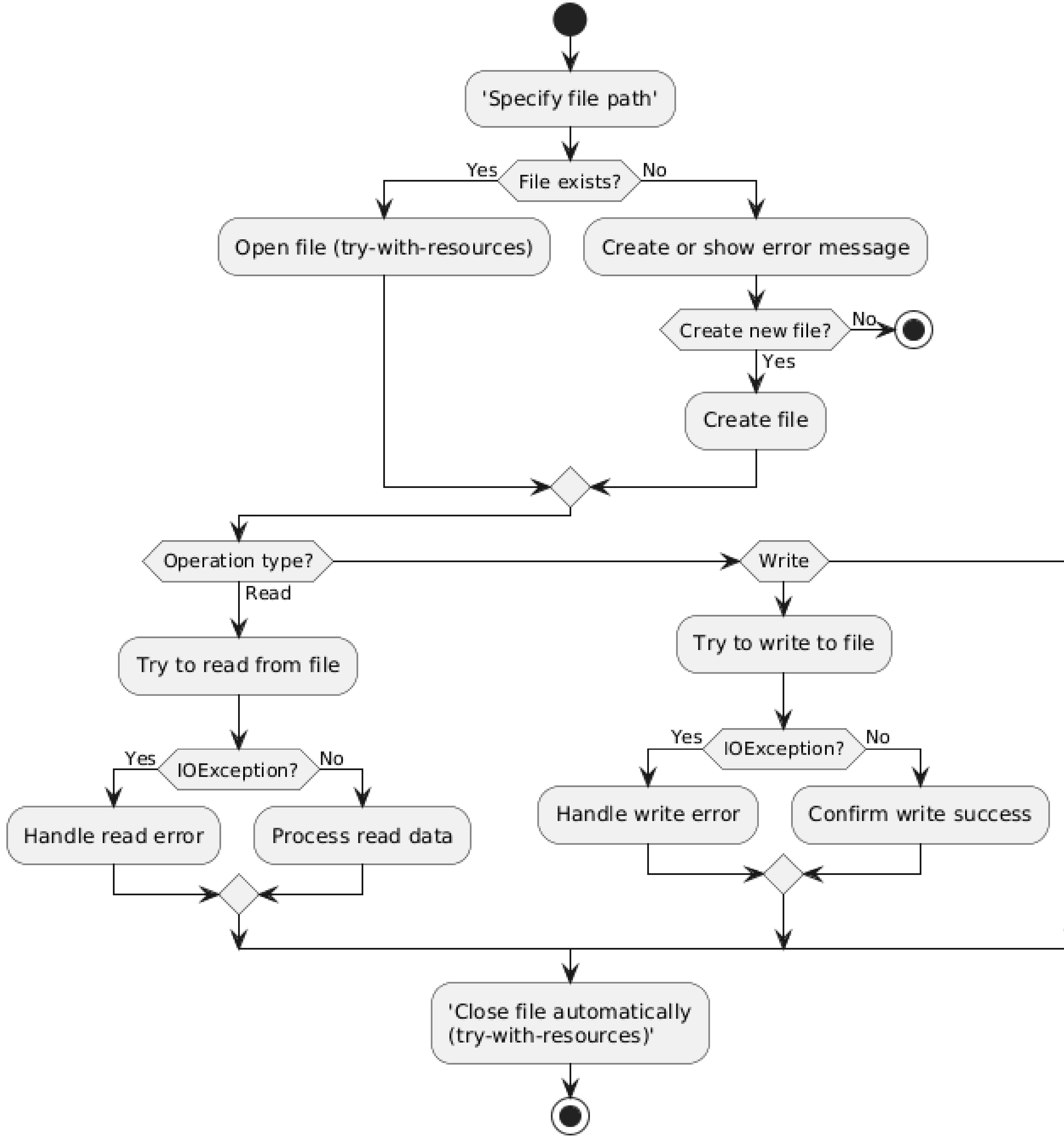
File handling methods

- Java provides multiple approaches for file handling, depending on file type, performance needs, and API level

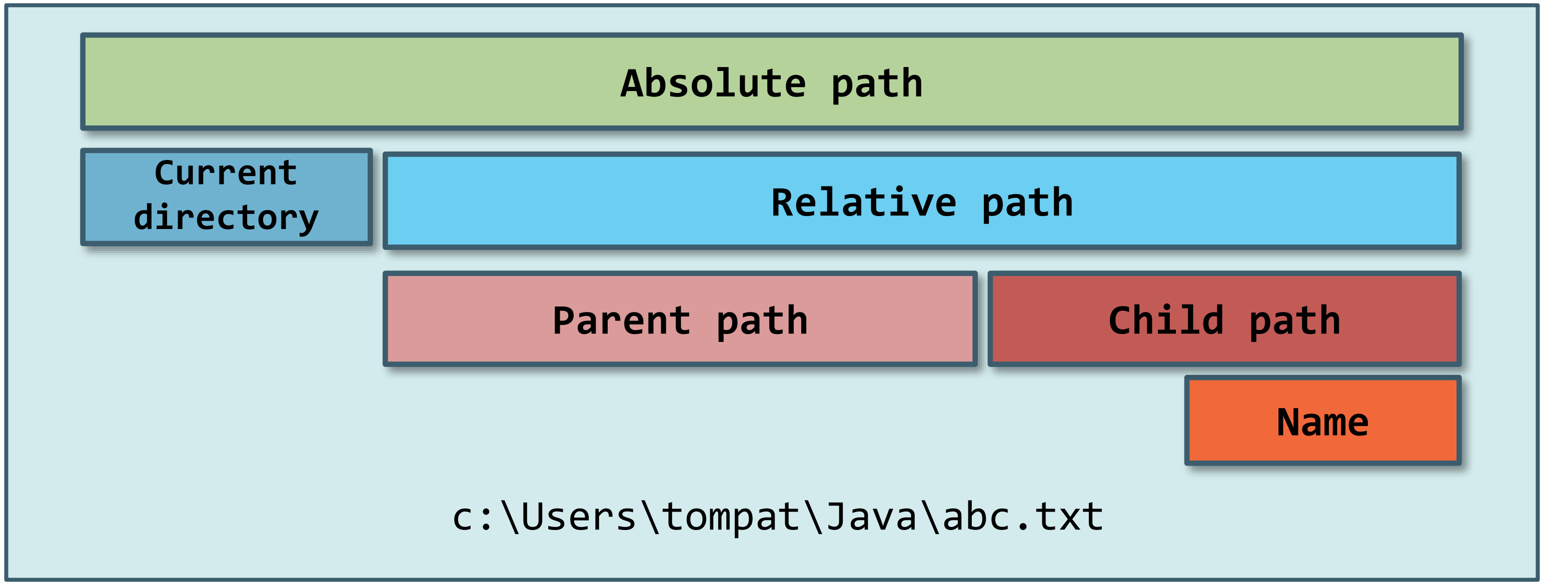
Situation	Recommended API
Small text files	Scanner
Large text files	BufferedReader / BufferedWriter
Binary files	FileInputStream / FileOutputStream
Object persistence	ObjectInputStream / ObjectOutputStream
Modern file operations	Path + Files (NIO.2)

File operations

General File Write/Read Process



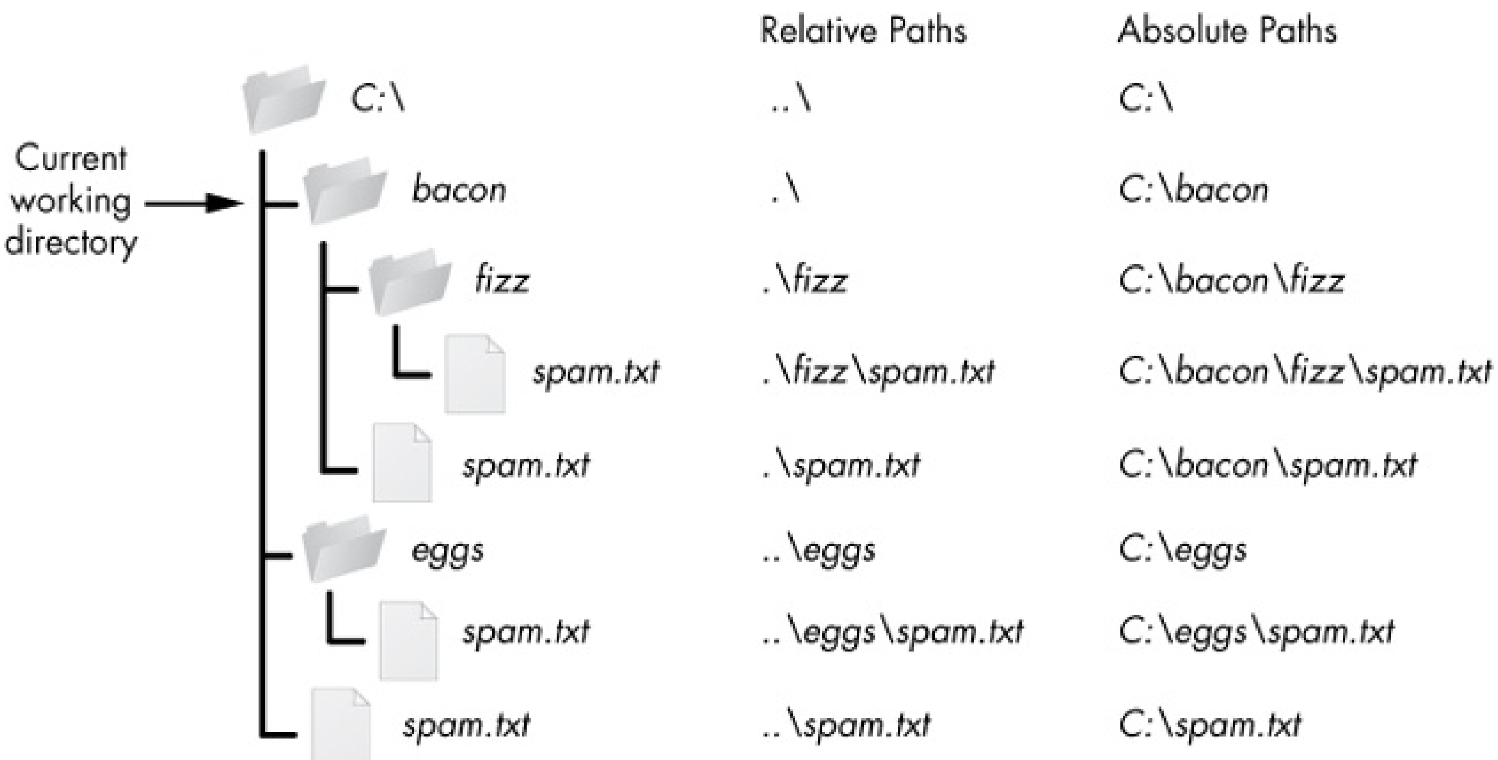
File path types



Absolute path: the complete address of a file or directory in a computer system, starting from the root directory and including all directories leading to the specific file or folder

Relative path: refers to a location that is relative to a current directory

File path types



Traditional I/O (java.io package)

- **1. Traditional I/O (java.io package) – File class**
  - the `File` class from the java.io package, allows us to work with files
  - to use the `File` class, create an object of the class, and specify the filename or directory name:

```
import java.io.File;  // Import the File class

File myObj = new File("filename.txt"); // Specify the filename
```

Traditional I/O (java.io package)

- 1. **Traditional I/O (java.io package) – File class**

Method	Type	Description
canRead()	Boolean	Tests whether the file is readable or not
canWrite()	Boolean	Tests whether the file is writable or not
createNewFile()	Boolean	Creates an empty file
delete()	Boolean	Deletes a file
exists()	Boolean	Tests whether the file exists
getName()	String	Returns the name of the file
getAbsolutePath()	String	Returns the absolute pathname of the file
length()	Long	Returns the size of the file in bytes
list()	String[]	Returns an array of the files in the directory
mkdir()	Boolean	Creates a directory

Traditional I/O (java.io package)

- 1. **Traditional I/O (java.io package) – File class**
  - **Create files**
    - `createNewFile()` method from the `File` class
    - this method returns
      - true: if the file was created successfully
      - false: if the file already exists
      - the method is enclosed in a try...catch block. This is necessary because it throws an IOException if an error occurs
    - create a file in a specific folder
      - to create a file in a specific directory (requires permission), specify the path of the file and use double backslashes to escape the "\" character (for Windows)
      - On Mac and Linux you can just write the path, like: /Users/name/filename.txt

```
File myObj = new File("C:\\Users\\MyName\\filename.txt");
```

Traditional I/O (java.io package)

- 1. Traditional I/O (java.io package) – File class
  - Create files

```
import java.io.File;          // Import the File class
import java.io.IOException; // Import IOException to handle errors

public class CreateFile {
  public static void main(String[] args) {
    try {
      File myObj = new File("filename.txt"); // Create File object
      if (myObj.createNewFile()) {           // Try to create the file
        System.out.println("File created: " + myObj.getName());
      } else {
        System.out.println("File already exists.");
      }
    } catch (IOException e) {
      System.out.println("An error occurred.");
      e.printStackTrace(); // Print error details
    }
  }
}
```

```
File created: filename.txt
```

Traditional I/O (java.io package)

- 1. **Traditional I/O (java.io package) – File class**
  - **Write files**

```
import java.io.FileWriter;   // Import the FileWriter class
import java.io.IOException;  // Import the IOException class

public class WriteToFile {
  public static void main(String[] args) {
    try {
      FileWriter myWriter = new FileWriter("filename.txt");
      myWriter.write("Files in Java might be tricky, but it is fun enough!");
      myWriter.close();  // must close manually
      System.out.println("Successfully wrote to the file.");
    } catch (IOException e) {
      System.out.println("An error occurred.");
      e.printStackTrace();
    }
  }
}
```

```
Successfully wrote to the file.
```

Traditional I/O (java.io package)

- 1. **Traditional I/O (java.io package) – File class**
  - **Read files**

```
import java.io.File;                    // Import the File class
import java.io.FileNotFoundException; // Import this class to handle errors
import java.util.Scanner;              // Import the Scanner class to read text files

public class ReadFile {
  public static void main(String[] args) {
    File myObj = new File("filename.txt");

    // try-with-resources: Scanner will be closed automatically
    try (Scanner myReader = new Scanner(myObj)) {
      while (myReader.hasNextLine()) {
        String data = myReader.nextLine();
        System.out.println(data);
      }
    } catch (FileNotFoundException e) {
      System.out.println("An error occurred.");
      e.printStackTrace();
    }
  }
}
```

Files in Java might be tricky, but it is fun enough!

Traditional I/O (java.io package)

- ## 1. Traditional I/O (java.io package) – File class
  - **Get file information**

```
import java.io.File;  // Import the File class

public class GetFileInfo {
  public static void main(String[] args) {
    File myObj = new File("filename.txt");
    if (myObj.exists()) {
      System.out.println("File name: " + myObj.getName());
      System.out.println("Absolute path: " + myObj.getAbsolutePath());
      System.out.println("Writeable: " + myObj.canWrite());
      System.out.println("Readable " + myObj.canRead());
      System.out.println("File size in bytes " + myObj.length());
    } else {
      System.out.println("The file does not exist.");
    }
  }
}
```

```
File name: filename.txt
Absolute path: C:\Users\MyName\filename.txt
Writeable: true
Readable: true
File size in bytes: 0
```

Traditional I/O (java.io package)

- 1. **Traditional I/O (java.io package) – File class**
  - **Delete files**

```
import java.io.File;  // Import the File class

public class DeleteFile {
  public static void main(String[] args) {
    File myObj = new File("filename.txt");
    if (myObj.delete()) {
      System.out.println("Deleted the file: " + myObj.getName());
    } else {
      System.out.println("Failed to delete the file.");
    }
  }
}
```

```
Deleted the file: filename.txt
```

Traditional I/O (java.io package)

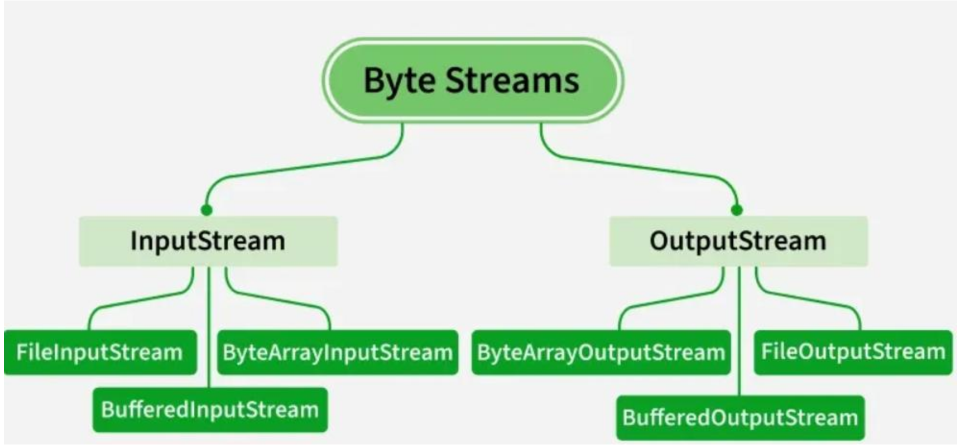
- 1. **Traditional I/O (java.io package) – File class**
  - **Delete folder**
    - it must be empty

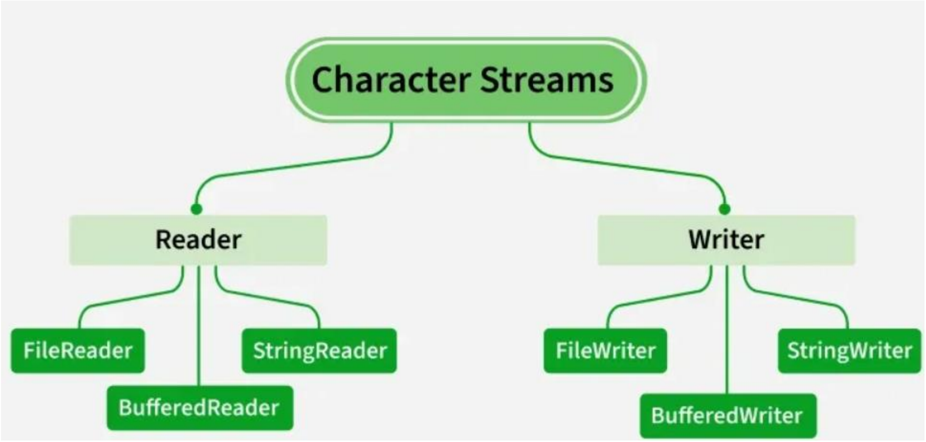
```
import java.io.File;

public class DeleteFolder {
  public static void main(String[] args) {
    File myObj = new File("C:\\Users\\MyName\\Test");
    if (myObj.delete()) {
      System.out.println("Deleted the folder: " + myObj.getName());
    } else {
      System.out.println("Failed to delete the folder.");
    }
  }
}
```

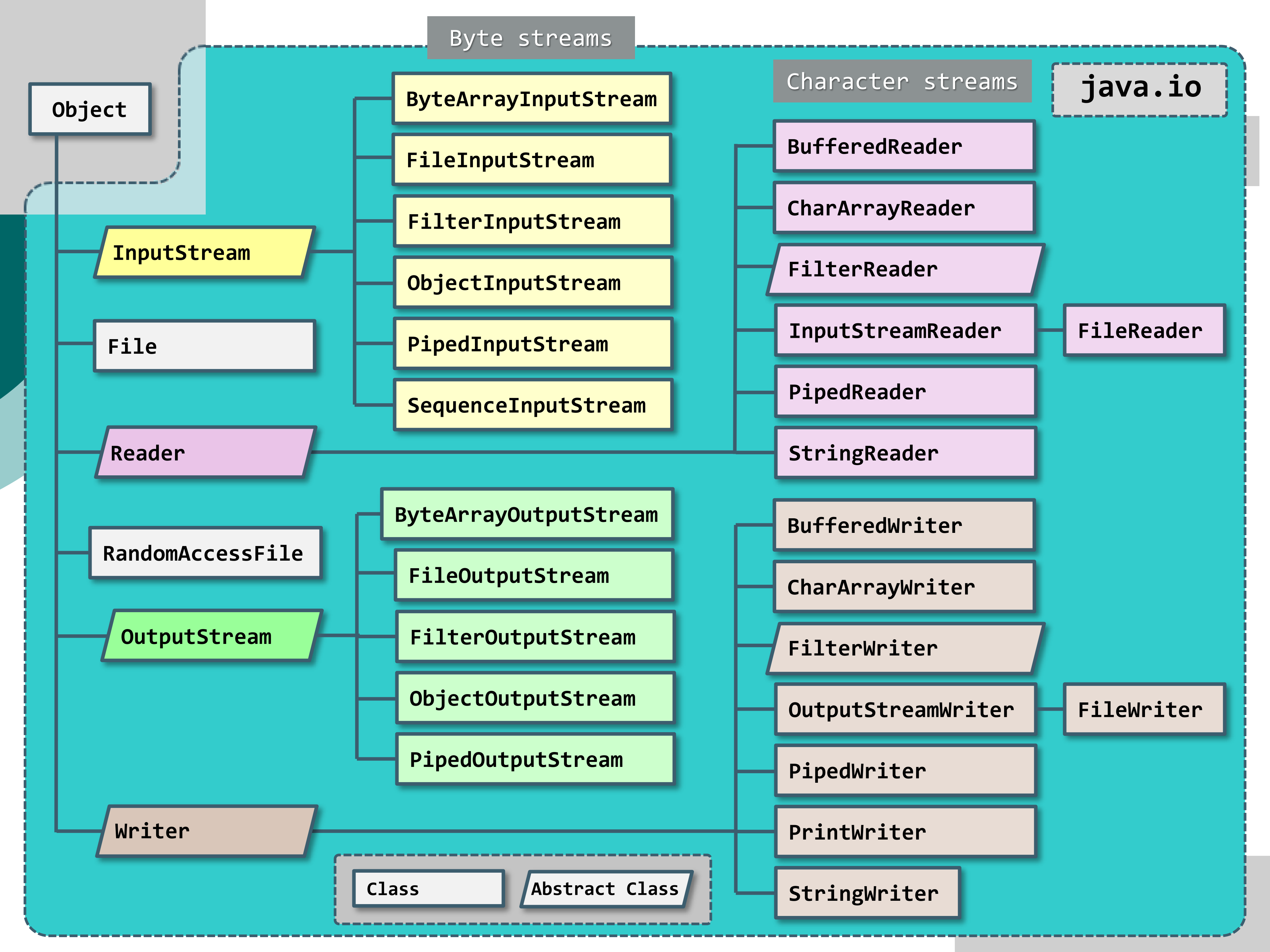
```
Deleted the folder: Test
```

Traditional I/O (java.io package)

- **1. Traditional I/O (java.io package) - I/O Streams**
  - in Java, **I/O streams** are the fundamental mechanism for handling input and output operations
  - I/O Streams are more flexible, because they work with **text and binary data** (like images, audio, PDFs)
  - Java I/O streams are categorized into two main types based on the type of data they handle:






Byte streams
Character streams
java.io
Object
InputStream
File
Reader
RandomAccessFile
OutputStream
Writer
ByteArrayInputStream
FileInputStream
FilterInputStream
ObjectInputStream
PipedInputStream
SequenceInputStream
ByteArrayOutputStream
FileOutputStream
FilterOutputStream
ObjectOutputStream
PipedOutputStream
BufferedReader
CharArrayReader
FilterReader
InputStreamReader
FileReader
PipedReader
StringReader
BufferedWriter
CharArrayWriter
FilterWriter
OutputStreamWriter
FileWriter
PipedWriter
PrintWriter
StringWriter
Class
Abstract Class

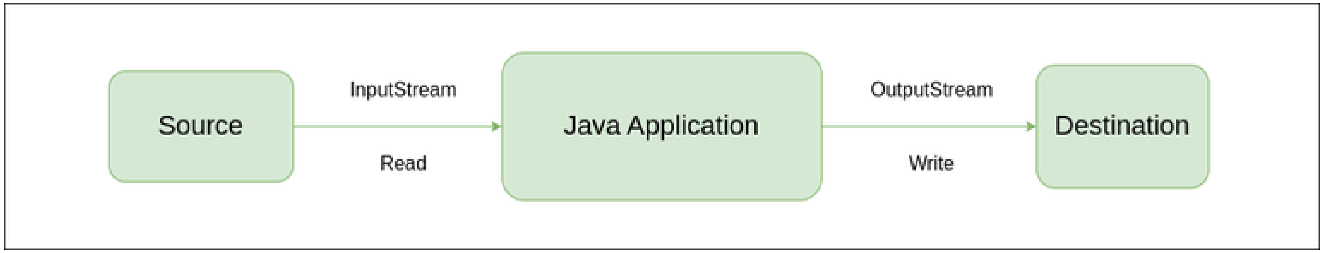
Traditional I/O (java.io package)

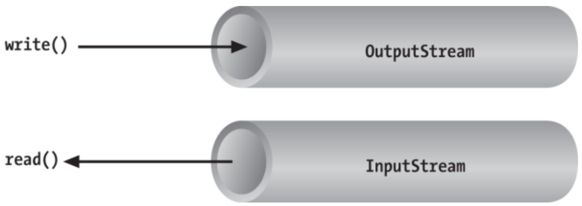
- 1. **Traditional I/O (java.io package) – I/O Streams**
  - **Types of Streams**
    - **Byte Streams**
      - work with raw **binary data** (like images, audio, and PDF files)
      - examples: `FileInputStream`, `FileOutputStream`
    - **Character Streams**
      - work with **text** (characters and strings)
      - these streams automatically handle character encoding
      - examples: `FileReader`, `FileWriter`, `BufferedReader`, `BufferedWriter`

Type	Input	Output
Byte	InputStream	OutputStream
Character	Reader	Wrtier

Traditional I/O (java.io package)

- 1. **Traditional I/O (java.io package) – I/O Streams**
  - **Types of Streams**
    - **OutputStream**
      - this abstract class is the superclass of all classes representing an output stream of bytes
      - an output stream accepts output bytes and sends them to some sink
      - applications that need to define a subclass of OutputStream must always provide at least a method that writes one byte of output
    - **InputStream**
      - a class is the superclass of all the io classes i.e. representing an input stream of bytes
      - it represents an input stream of bytes
      - applications that are defining a subclass of the Java InputStream class must provide a method, that returns the next byte of input





Traditional I/O (java.io package)

- 1. Traditional I/O (java.io package) – I/O Streams
  - **FieInputStream: read a text file**

```
import java.io.FileInputStream;  // Import FileInputStream
import java.io.IOException;      // Import IOException

public class Main {
  public static void main(String[] args) {
    // try-with-resources: FileInputStream will be closed automatically
    try (FileInputStream input = new FileInputStream("filename.txt")) {

      int i;  // variable to store each byte that is read

      // Read one byte at a time until end of file (-1 means "no more data")
      while ((i = input.read()) != -1) {
        // Convert the byte to a character and print it to the console
        System.out.print((char) i);
      }

    } catch (IOException e) {
      // If an error happens (e.g. file not found), print an error message
      System.out.println("Error reading file.");
    }
  }
}
```

```
Some text from filename.txt
```

Traditional I/O (java.io package)

- 1. Traditional I/O (java.io package) – I/O Streams
  - **FieInputStream: copy a binary file**

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

public class CopyFile {
  public static void main(String[] args) {
    // Copy image.jpg into copy.jpg
    try (FileInputStream input = new FileInputStream("image.jpg");
         FileOutputStream output = new FileOutputStream("copy.jpg")) {

      int i;
      while ((i = input.read()) != -1) {
        output.write(i);  // write the raw byte to the new file
      }

      System.out.println("File copied successfully.");

    } catch (IOException e) {
      System.out.println("Error handling file.");
    }
  }
}
```

```
File copied successfully.
```

Traditional I/O (java.io package)

- 1. **Traditional I/O (java.io package) – I/O Streams**
  - **FieOutputStream: write a text file**

```
import java.io.FileOutputStream;
import java.io.IOException;

public class Main {
  public static void main(String[] args) {
    // The text we want to write
    String text = "Hello World!";

    // try-with-resources: stream will be closed automatically
    try (FileOutputStream output = new FileOutputStream("filename.txt")) {
      output.write(text.getBytes());  // convert text to bytes and write
      System.out.println("Successfully wrote to file.");
    } catch (IOException e) {
      System.out.println("Error writing file.");
      e.printStackTrace();
    }
  }
}
```

```
Successfully wrote to file.
```

Traditional I/O (java.io package)

- 1. Traditional I/O (java.io package) – I/O Streams
  - FieOutputStream: append to a file

```
import java.io.FileOutputStream;
import java.io.IOException;

public class Main {
  public static void main(String[] args) {
    String text = "\nAppended text!";

    // true = append mode (keeps existing content)
    try (FileOutputStream output = new FileOutputStream("filename.txt", true)) {
      output.write(text.getBytes());
      System.out.println("Successfully appended to file.");
    } catch (IOException e) {
      System.out.println("Error writing file.");
      e.printStackTrace();
    }
  }
}
```

```
Successfully appended to file.
```

Traditional I/O (java.io package)

1. Traditional I/O (java.io package) – I/O Streams

- **Stream Filtering**
  - in Java, Stream filtering refers to using the `filter()` method to create a filtered stream that contains only the elements that satisfy a given condition
  - Key Points
    - `filter()` takes a Predicate (a boolean-returning function)
    - it returns a new Stream containing only elements for which the predicate is true
    - it does not modify the original collection (streams are immutable)
    - filtering is part of a pipeline (stream → intermediate operations → terminal operation)
    - often used together with map(), sorted(), collect(), etc.
    - example: (*lambda expression*)

```
List<String> names = List.of("Anna", "Bob", "Alex", "Charles");

List<String> result = names.stream()
        .filter(n -> n.startsWith("A"))   // keeps only names starting with A
        .collect(Collectors.toList());

System.out.println(result);  // [Anna, Alex]
```

Traditional I/O (java.io package)

- **1. Traditional I/O (java.io package) – I/O Streams**
  - **Stream Filtering**
    - example: (*lambda expression*)

filter()

```
import java.util.stream.*;
public class FilterExample {
    public static void main(String[] args) {
        Stream.of(1, 2, 3, 4, 5)
            .filter(x -> x % 2 != 0)
            .forEach(System.out::println);
    }
}
```

Traditional I/O (java.io package)

- **1. Traditional I/O (java.io package) – BufferedReader and BufferedWriter**
  - BufferedReader reads text line by line with `readLine()`
  - BufferedWriter writes text efficiently and add new lines with `newLine`()
  - These classes are usually combined with `FileReader` and `FileWriter`, which handle opening or creating the file
  - The buffered classes then make reading/writing faster by using a memory buffer

Traditional I/O (java.io package)

- 1. **Traditional I/O (java.io package) – BufferedReader and BufferedWriter**
  - **BufferedReader: read a text file (line by line)**

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

public class Main {
  public static void main(String[] args) {
    try (BufferedReader br = new BufferedReader(new FileReader("filename.txt")))
{
      String line;
      while ((line = br.readLine()) != null) {
        System.out.println(line);
      }
    } catch (IOException e) {
      System.out.println("Error reading file.");
    }
  }
}
```

```
Some text from filename.txt
```

Traditional I/O (java.io package)

- 1. **Traditional I/O (java.io package) – BufferedReader and BufferedWriter**
  - **BufferedWriter: write to a text file**

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

public class Main {
  public static void main(String[] args) {
    try (BufferedWriter bw = new BufferedWriter(new FileWriter("filename.txt")))
{
      bw.write("First line");
      bw.newLine();  // add line break
      bw.write("Second line");
      System.out.println("Successfully wrote to the file.");
    } catch (IOException e) {
      System.out.println("Error writing file.");
    }
  }
}
```

```
Successfully wrote to the file.
```

Traditional I/O (java.io package)

- **1. Traditional I/O (java.io package) – BufferedReader and BufferedWriter**
  - **BufferedWriter: append to a text file**

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

public class Main {
  public static void main(String[] args) {
    // true = append mode
    try (BufferedWriter bw = new BufferedWriter(new FileWriter("filename.txt", true))) {
      bw.newLine();                          // move to a new line
      bw.write("Appended line");             // add new text at the end
      System.out.println("Successfully appended to the file.");
    } catch (IOException e) {
      System.out.println("Error writing file.");
    }
  }
}
```

```
Successfully appended to the file.
```

Scanner class (java.util package)

- **2. Scanner class (java.util package)**
  - the Scanner class (in java.util) is a simple and convenient tool for **reading text input in Java**
  - it can read data from various sources such as:
    - keyboard input (System.in)
    - text files (new Scanner(file))
    - PDF
    - Strings
  - Scanner breaks the input into tokens (pieces of text) and provides easy methods for reading:
    - `next()`: reads the next token (word)
    - `nextLine()`: reads an entire line
    - `nextInt()`, `nextDouble`(), etc.: read numbers
    - `hasNext()`, `hasNextLine`(): check if more input is available

Scanner class (java.util package)

- **2. Scanner class (java.util package)**
  - **reading from the consolse example**

```
import java.util.Scanner;

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

        Scanner sc = new Scanner(System.in);
        System.out.println("Enter your name:");
        String name = sc.nextLine();

        System.out.println("Hello, " + name + " Welcome to the Java course.");

    }
}
```

Scanner class (java.util package)

- **2. Scanner class (java.util package)**
  - the Scanner class can **read data** not only from the keyboard (System.in), but also **from text files**
  - it is often used for simple file input, such as reading lines, words, or numbers from a file
  - Scanner can read lines, words, or numbers from a file
    - easy to use, ideal for simple text file input
    - common methods:
      - `nextLine(), next(), nextInt(), hasNextLine(), hasNextInt()`
    - needs a `try-catch` for `FileNotFoundException`
    - can be use custom delimiters for parsing

Scanner class (java.util package)

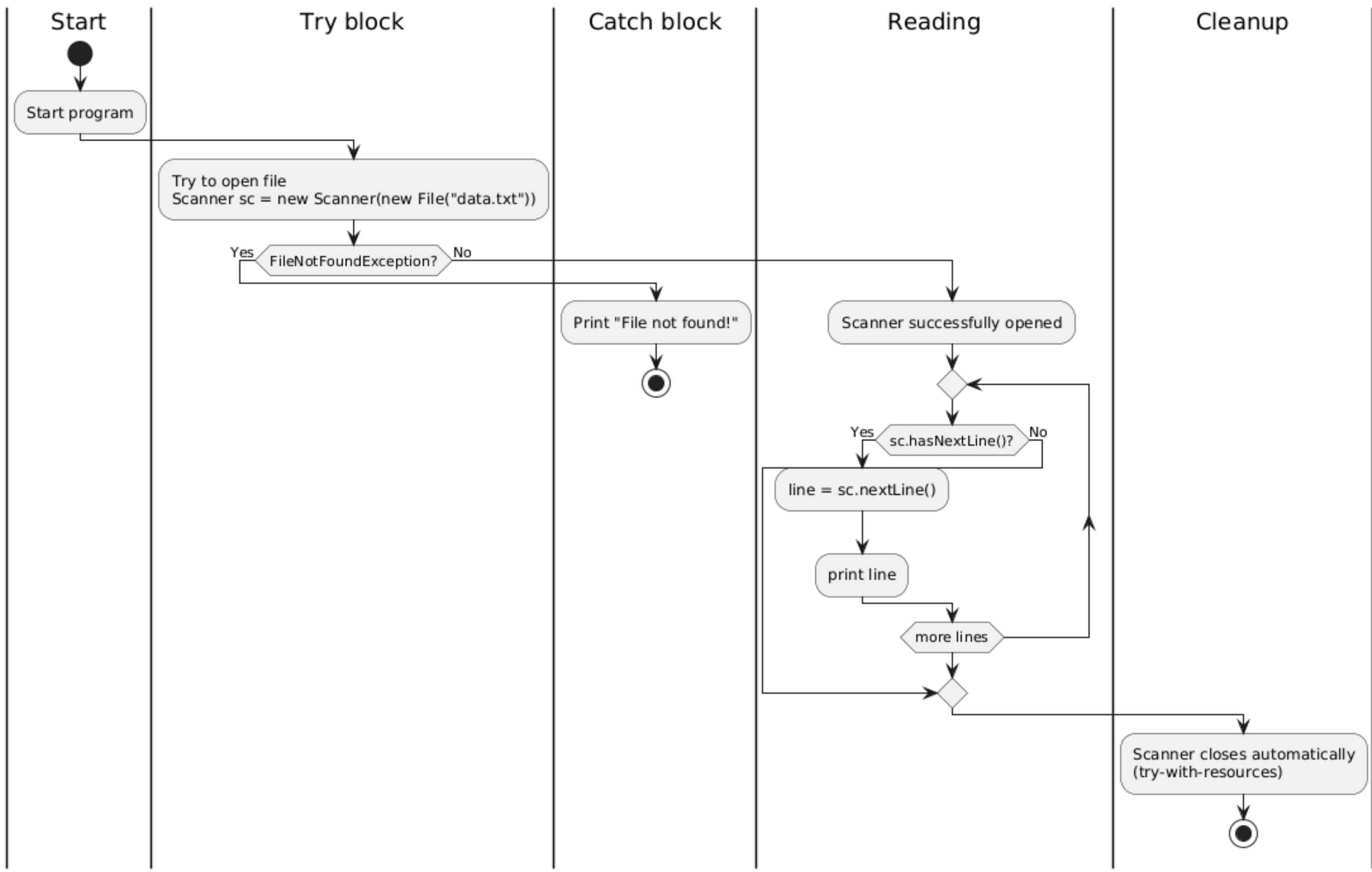
- **2. Scanner class (java.util package)**
  - reading a file line by line:

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class ReadFileExample {
    public static void main(String[] args) {
        try (Scanner sc = new Scanner(new File("data.txt"))) {
            while (sc.hasNextLine()) {
                String line = sc.nextLine();   // read full line
                System.out.println(line);
            }
        } catch (FileNotFoundException e) {
            System.out.println("File not found!");
        }
    }
}
```

Scanner class

- **2. Scanner class (java.util package)**

Scanner-based file reading line by line



Scanner class (java.util package)

- 2. **Scanner class (java.util package)**
  - reading numbers from a file:

```
Scanner sc = new Scanner(new File("numbers.txt"));

while (sc.hasNextInt()) {
    int n = sc.nextInt();
    System.out.println("Number: " + n);
}

sc.close();
```

  - reading words / tokens from a file:

```
Scanner sc = new Scanner(new File("words.txt"));

while (sc.hasNext()) {
    String word = sc.next();
    System.out.println(word);
}
```

New I/O (NIO & NIO.2 – java.nio, java.nio.file)

- **3. New I/O (NIO & NIO.2 – java.nio, java.nio.file)**
  - NIO (New Input/Output) and NIO.2 are **modern Java APIs** designed to provide **faster, more flexible, and more scalable file and I/O operations** compared to the traditional java.io package
  - NIO (java.nio)
    - introduced to support buffer- and channel-based I/O, enabling more efficient data handling and non-blocking operations
  - NIO.2 (java.nio.file)
    - introduced in Java 7, it significantly improved file system handling with a cleaner and more powerful API

New I/O (NIO & NIO.2 – java.nio, java.nio.file)

- **3. New I/O (NIO & NIO.2 – java.nio, java.nio.file)**
  - Core Classes and Interfaces
    - `Path`: represents a file or directory path (modern replacement for File)
    - `Paths`: utility class for creating Path objects
    - `Files`: provides static methods for common file operations
    - Channels and Buffers (`FileChannel`, `ByteBuffer`): used for high-performance I/O and large file processing
  - Advantages
    - cleaner and more readable API
    - better exception handling
    - platform-independent path handling
    - support for atomic file operations
    - suitable for large files and high-performance applications

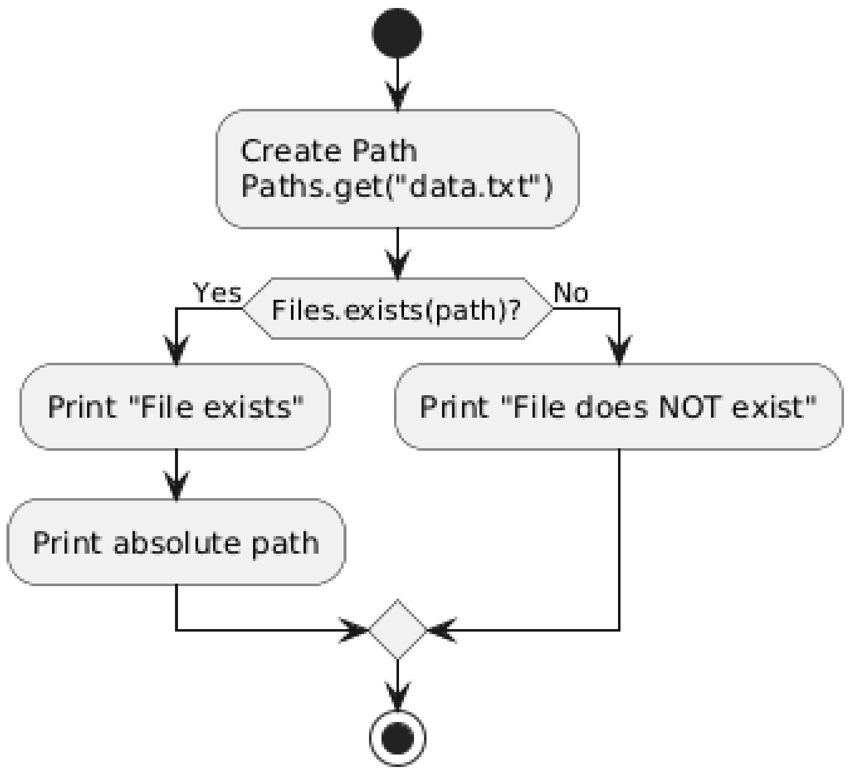
New I/O (NIO & NIO.2 – java.nio, java.nio.file)

- **3. New I/O (NIO & NIO.2 – java.nio, java.nio.file)**
  - creating Path + checking existence

```
import java.nio.file.*;


public class NioExistsExample {
    public static void main(String[] args) {
        Path path = Paths.get("data.txt");

        if (Files.exists(path)) {
            System.out.println("File exists: " + path.toAbsolutePath());
        } else {
            System.out.println("File does NOT exist.");
        }
    }
}
```

New I/O (NIO & NIO.2 – java.nio, java.nio.file)

- **3. New I/O (NIO & NIO.2 – java.nio, java.nio.file)**
  - creating Path + checking existence



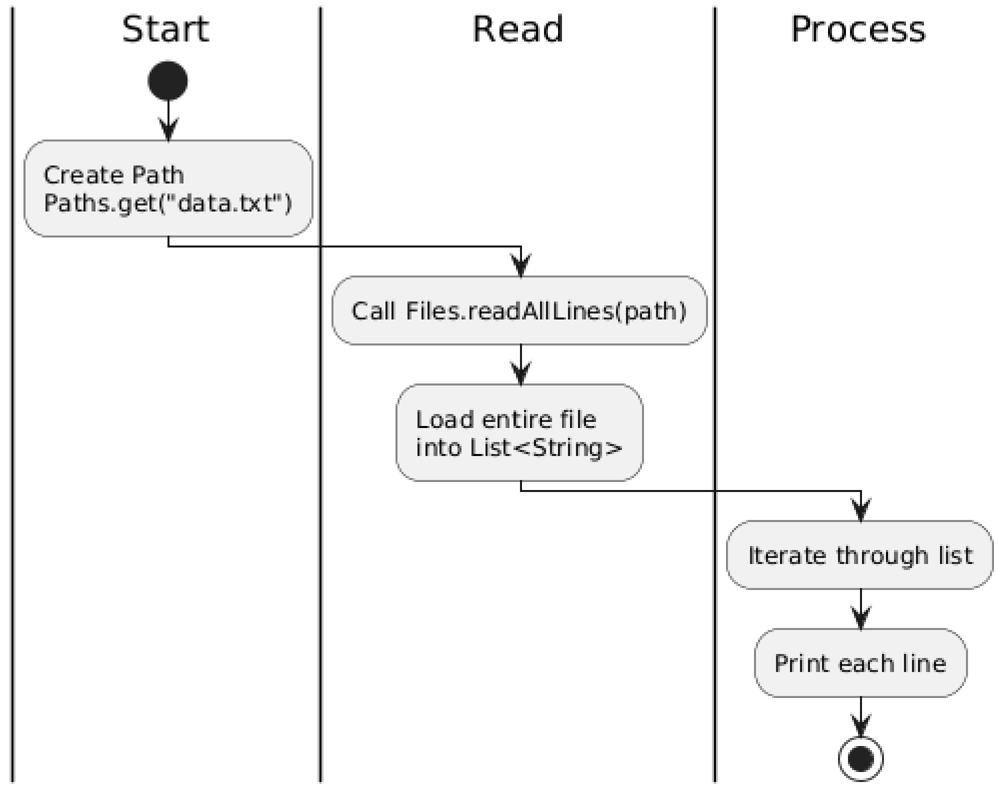
New I/O (NIO & NIO.2 – java.nio, java.nio.file)

- **3. New I/O (NIO & NIO.2 – java.nio, java.nio.file)**
  - read entire file line by line (for small files)

```
import java.io.IOException;
import java.nio.file.*;
import java.util.List;


public class ReadAllLinesExample {
    public static void main(String[] args) throws IOException {
        Path path = Paths.get("data.txt");
        List<String> lines = Files.readAllLines(path);
        lines.forEach(System.out::println);
    }
}
```

New I/O (NIO & NIO.2 – java.nio, java.nio.file)

- **3. New I/O (NIO & NIO.2 – java.nio, java.nio.file)**
  - read entire file line by line (for small files)



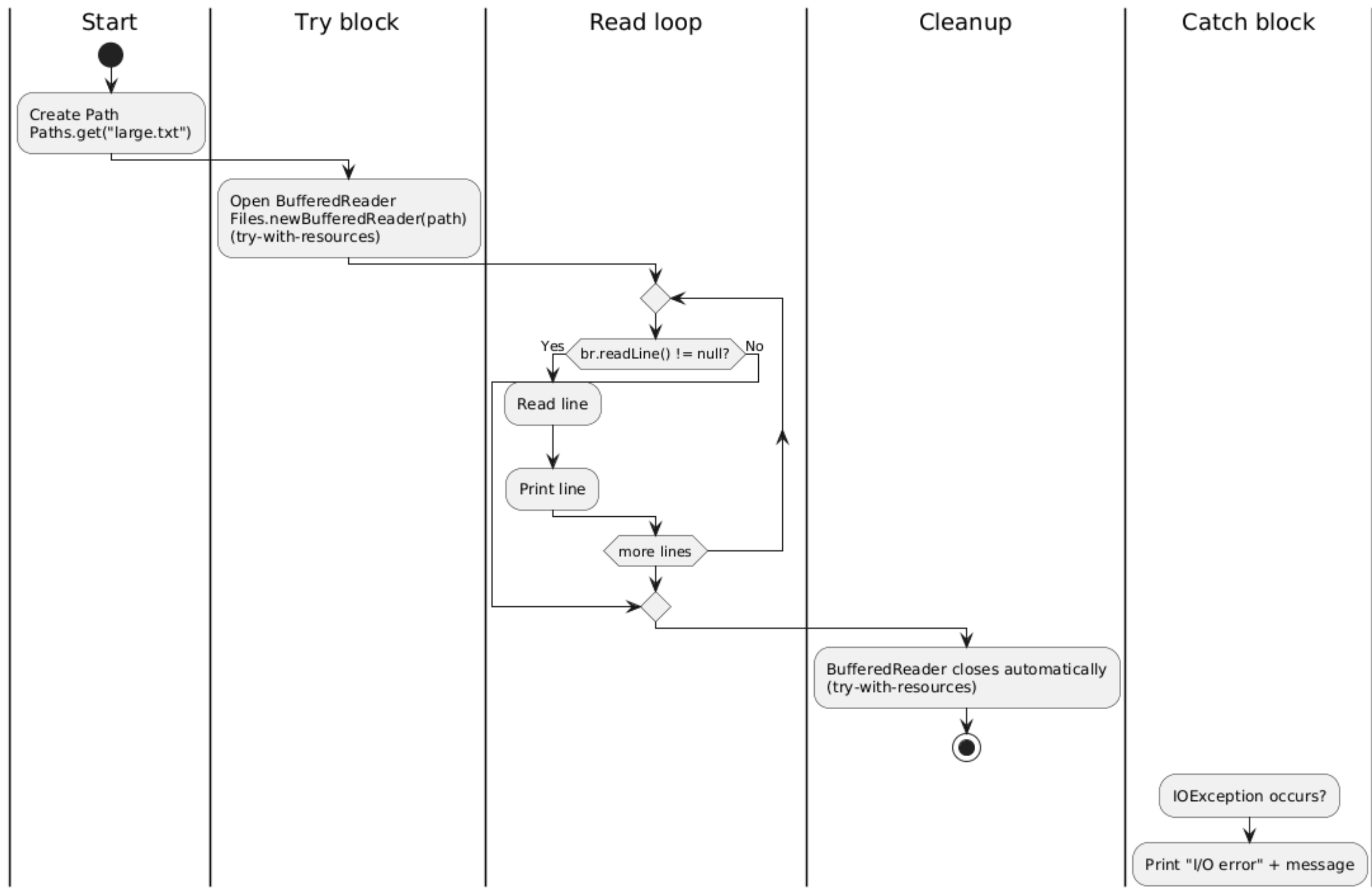
New I/O (NIO & NIO.2 – java.nio, java.nio.file)

- 3. **New I/O (NIO & NIO.2 – java.nio, java.nio.file)**
  - read entire file by BufferedReader (for huge files)

```
import java.io.BufferedReader;
import java.io.IOException;
import java.nio.file.*;

public class BufferedReaderNioExample {
    public static void main(String[] args) {
        Path path = Paths.get("large.txt");

        try (BufferedReader br = Files.newBufferedReader(path)) {
            String line;
            while ((line = br.readLine()) != null) {
                System.out.println(line);
            }
        } catch (IOException e) {
            System.err.println("I/O error: " + e.getMessage());
        }
    }
}
```

New I/O (NIO & NIO.2 – java.nio, java.nio.file)

- 3. **New I/O (NIO & NIO.2 – java.nio, java.nio.file)**
  - read entire file BufferedReader (for huge files)



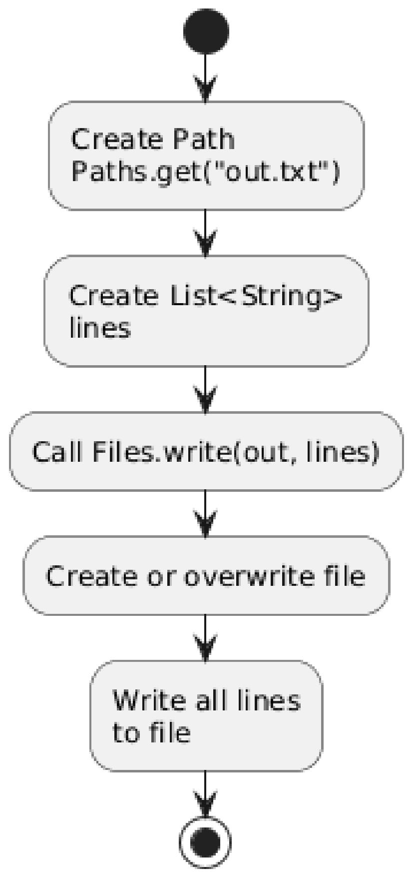
New I/O (NIO & NIO.2 – java.nio, java.nio.file)

- 3. **New I/O (NIO & NIO.2 – java.nio, java.nio.file)**
  - writing texts

```
import java.io.IOException;
import java.nio.file.*;
import java.util.List;

public class WriteLinesExample {
    public static void main(String[] args) throws IOException {
        Path out = Paths.get("out.txt");
        List<String> lines = List.of("Hello", "from", "NIO.2");

        Files.write(out, lines);  // creates or overwrites by default
    }
}
```

New I/O (NIO & NIO.2 – java.nio, java.nio.file)

- **3. New I/O (NIO & NIO.2 – java.nio, java.nio.file)**
  - writing texts



New I/O (NIO & NIO.2 – java.nio, java.nio.file)

- 3. New I/O (NIO & NIO.2 – java.nio, java.nio.file)
  - copy / move / delete

```
import java.io.IOException;
import java.nio.file.*;
import static java.nio.file.StandardCopyOption.*;

public class CopyMoveDeleteExample {
    public static void main(String[] args) throws IOException {
        Path src  = Paths.get("data.txt");
        Path copy = Paths.get("backup", "data_copy.txt");
        Path moved= Paths.get("backup", "data_moved.txt");

        Files.createDirectories(copy.getParent());

        Files.copy(src, copy, REPLACE_EXISTING);   // copy
        Files.move(copy, moved, REPLACE_EXISTING); // move (rename)
        Files.deleteIfExists(moved);               // delete safely
    }
}
```

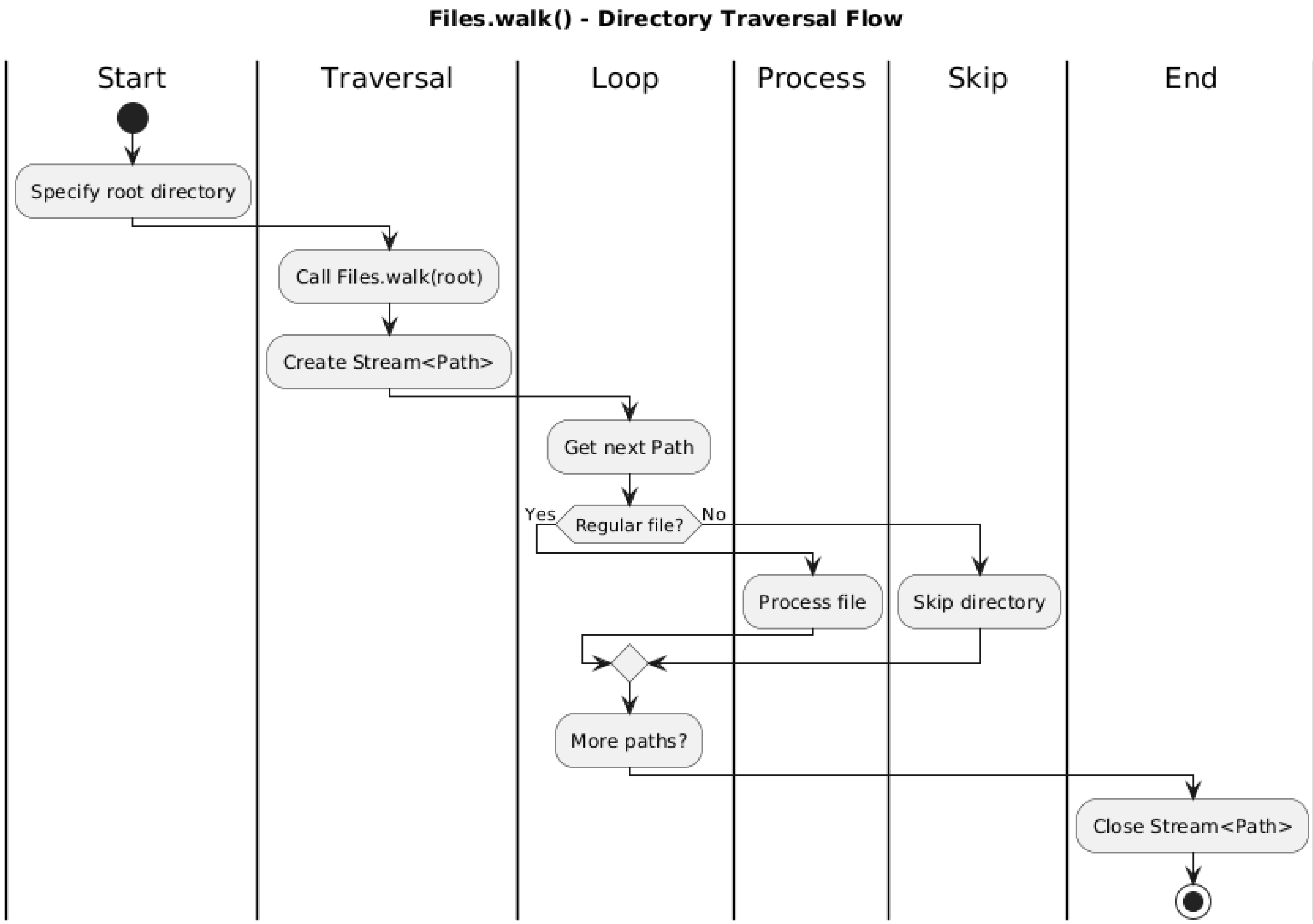
New I/O (NIO & NIO.2 – java.nio, java.nio.file)

- **3. New I/O (NIO & NIO.2 – java.nio, java.nio.file)**
  - iterate over a folder
  - `Files.walk()` recursively traverses a directory tree and returns a `Stream<Path>`, which must be closed after use

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.stream.Stream;

public class WalkExample {
    public static void main(String[] args) throws IOException {
        Path root = Paths.get("src");

        try (Stream<Path> paths = Files.walk(root)) {
            paths.filter(Files::isRegularFile)
                 .forEach(System.out::println);
        }
    }
}
```

New I/O (NIO & NIO.2 – java.nio, java.nio.file)

- 3. New I/O (NIO & NIO.2 – java.nio, java.nio.file)



New I/O (NIO & NIO.2 – java.nio, java.nio.file)

- **3. New I/O (NIO & NIO.2 – java.nio, java.nio.file)**
  - fast read/write

```
import java.io.IOException;
import java.nio.file.*;

public class ReadWriteStringExample {
    public static void main(String[] args) throws IOException {
        Path p = Path.of("data.txt");
        String content = Files.readString(p);
        Files.writeString(Path.of("out.txt"), content.toUpperCase());
    }
}
```

Thank you for your attention!