Topic-1: Java I/O API

The Java I/O (Input/Output) API provides a wide range of classes and interfaces for reading and writing data to various sources, including files, networks, and console. Here's an overview of the Java I/O API:

Java I/O Packages:

1. `java.io`: Contains classes for file I/O, console I/O, and stream I/O.
2. `java.nio`: Provides classes for non-blocking I/O operations and buffer management.

Java I/O Classes and Interfaces:

Input Streams:

1. `InputStream`: Abstract class for reading bytes.
2. `Reader`: Abstract class for reading characters.
3. `BufferedInputStream`: Reads bytes from a stream, buffering data for efficiency.
4. `BufferedReader`: Reads characters from a reader, buffering data for efficiency.
5. `FileInputStream`: Reads bytes from a file.
6. `FileReader`: Reads characters from a file.
7. `Scanner`: Parses input from various sources.

Output Streams:

1. `OutputStream`: Abstract class for writing bytes.
2. `Writer`: Abstract class for writing characters.
3. `BufferedOutputStream`: Writes bytes to a stream, buffering data for efficiency.
4. `BufferedWriter`: Writes characters to a writer, buffering data for efficiency.
5. `FileOutputStream`: Writes bytes to a file.
6. `FileWriter`: Writes characters to a file.
7. `PrintStream`: Writes formatted text to a stream.
8. `PrintWriter`: Writes formatted text to a writer.

Other Key Classes:

1. `File`: Represents a file or directory.
2. `RandomAccessFile`: Allows reading and writing to a file at arbitrary positions.
3. `DataInputStream` and `DataOutputStream`: Read and write primitive data types.

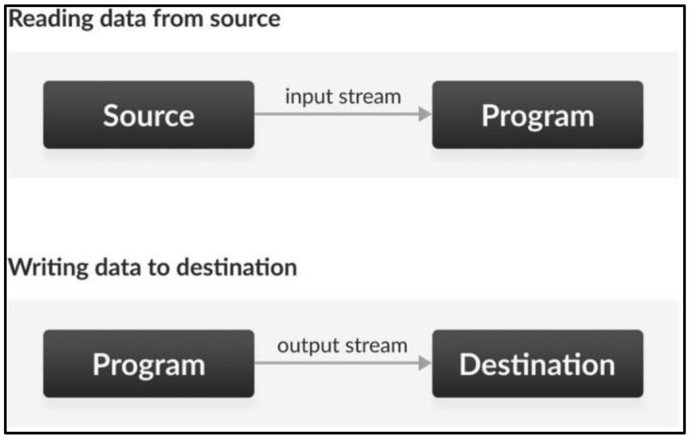
Java NIO Classes:

1. `Channel`: Abstract class for reading and writing data.
2. `Buffer`: Abstract class for storing data.
3. `ByteBuffer`, `CharBuffer`, etc.: Concrete buffer classes.
4. `Selector`: Allows multiplexing multiple channels.

Topic-2: Standard I/O streams

In Java, streams are the sequence of data that are read from the source and written to the destination.

An **input stream** is used to read data from the source. And an **output stream** is used to write data to the destination.



In Java, the standard I/O (Input/Output) streams are used to read from and write to various sources, such as the console, files, and network connections. Here are the standard I/O streams in Java:

Input Streams:

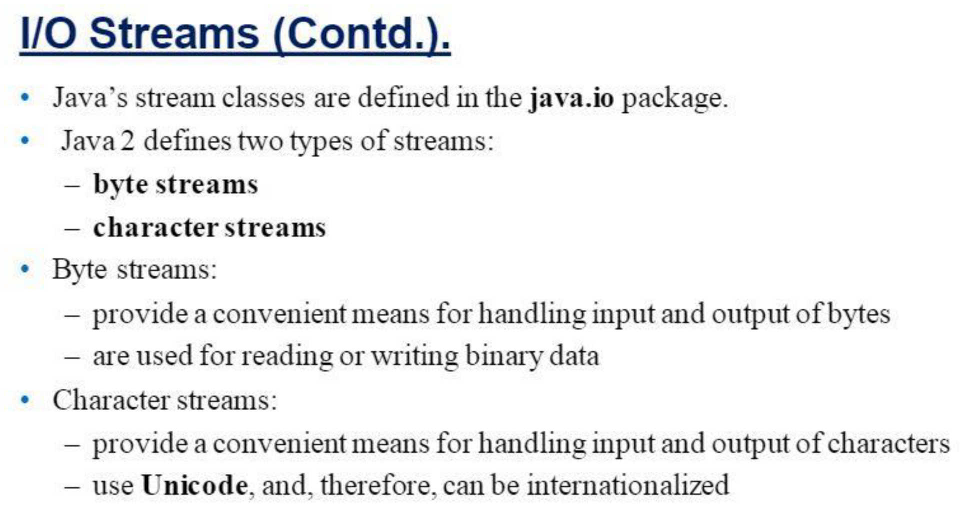
1. **System.in**: The standard input stream, typically the keyboard.
2. **BufferedReader**: Reads text from a character-input stream, buffering characters for efficient reading.

Output Streams:

1. **System.out**: The standard output stream, typically the console.
2. **System.err**: The standard error stream, typically the console.

Topic-3: Types of streams

In Java, streams are classified into two main types based on the kind of data they handle: **byte streams** and **character streams**. Each type has specific classes for input and output operations. Here’s a breakdown of the types of streams in Java:



I/O Streams (Contd.).

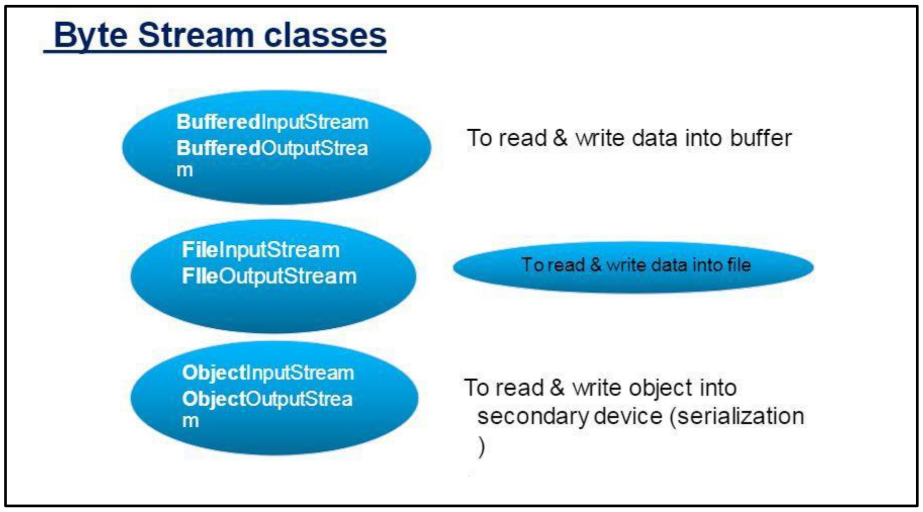
- Java’s stream classes are defined in the **java.io** package.
- Java 2 defines two types of streams:
  - **byte streams**
  - **character streams**
- Byte streams:
  - provide a convenient means for handling input and output of bytes
  - are used for reading or writing binary data
- Character streams:
  - provide a convenient means for handling input and output of characters
  - use **Unicode**, and, therefore, can be internationalized

1. Byte Streams

Byte streams are designed for handling raw binary data. They read and write data in bytes, making them suitable for binary files like images, audio files, and any non-text data.

- **InputStream**: An abstract class for reading byte streams.
  - **FileInputStream**: Reads bytes from a file.
  - **BufferedInputStream**: Buffers input to provide efficient reading of bytes.
  - **DataInputStream**: Allows reading Java primitive data types from an input stream.

- **OutputStream**: An abstract class for writing byte streams.
  - **FileOutputStream**: Writes bytes to a file.
  - **BufferedOutputStream**: Buffers output to provide efficient writing of bytes.
  - **DataOutputStream**: Allows writing Java primitive data types to an output stream.

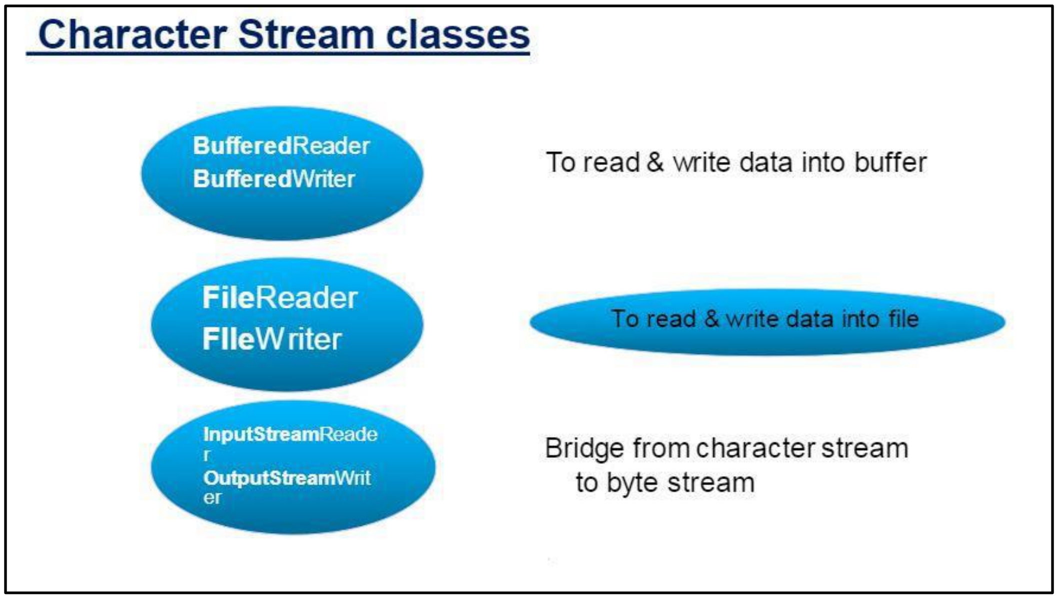


2. Character Streams

Character streams are designed for handling character data. They read and write data in characters, which makes them suitable for text files. Character streams are necessary when dealing with character encoding (e.g., UTF-8, ASCII).

Key Classes:

- **Reader**: An abstract class for reading character streams. Subclasses include:
  - **`FileReader`:** Reads characters from a file.
  - **`BufferedReader`:** Buffers input for improved performance and allows reading of lines.
  - **`CharArrayReader`:** Reads characters from a char array.
- **Writer**: An abstract class for writing character streams. Subclasses include:
  - **`FileWriter`:** Writes characters to a file.
  - **`BufferedWriter`:** Buffers output for improved performance and allows writing of lines.
  - **`CharArrayWriter`:** Writes characters to a char array



Topic-4: Byte streams

Byte streams in Java are designed for handling raw binary data, making them suitable for reading and writing files that contain binary content, such as images or audio files. The core classes for byte streams are `InputStream` for reading and `OutputStream` for writing.

Key Classes

1. **InputStream**: Abstract class for reading byte streams.
   - **Common subclasses:**
     - `FileInputStream`
     - `BufferedInputStream`
     - `DataInputStream`
2. **OutputStream**: Abstract class for writing byte streams.
   - **Common subclasses:**
     - `FileOutputStream`
     - `BufferedOutputStream`
     - `DataOutputStream`

Example: Reading and Writing with Byte Streams

Here's an example that demonstrates how to use `FileInputStream` and `FileOutputStream` to read from and write to a binary file.

Step 1: Writing Data to a Binary File

```
import java.io.*;

public class ByteStreamWriteExample
{
    public static void main(String[] args)
    {
          String data = "Hello, World!";

          // Convert string to byte array
          byte[] byteArray = data.getBytes();

          try
          {
            FileOutputStream fos = new FileOutputStream("example.dat");
            fos.write(byteArray); // Write byte array to the file
            System.out.println("Data written to example.dat");
          }
          catch (IOException e)
          {
            e.printStackTrace();
          }
    }
}
```

Step 2: Reading Data from a Binary File

```
import java.io.*;

public class ByteStreamReadExample
{
    public static void main(String[] args)
    {
        byte[] byteArray = new byte[100]; // Buffer to hold the data

        try
        {
          FileInputStream fis = new FileInputStream("example.dat");

          // Read bytes into the buffer
          int bytesRead = fis.read(byteArray);

          // Convert byte array to string
          String data = new String(byteArray, 0, bytesRead);
          System.out.println("Data read from example.dat: "+data);
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }
    }
}
```

Topic-4: Character streams

Character streams in Java are designed for handling text data, which allows for the reading and writing of characters. They automatically handle the conversion between bytes and characters based on the character encoding.

Key Classes

1. **Reader**: Abstract class for reading character streams.
   - **Common subclasses:**
     - `FileReader`
     - `BufferedReader`
     - `CharArrayReader`
2. **Writer**: Abstract class for writing character streams.
   - **Common subclasses:**
     - `FileWriter`
     - `BufferedWriter`
     - `PrintWriter`

Example: Reading and Writing with Character Streams

Here's an example that demonstrates how to use `FileReader` and `FileWriter` to read from and write to a text file.

Step 1: Writing Data to a Text File

```
import java.io.*;

public class CharacterStreamWriteExample
{
    public static void main(String[] args)
    {
        String data = "Hello, World!\nWelcome to Java char streams.";

        try
        {
            BufferedWriter writer = new BufferedWriter(new
                                    FileWriter("example.txt"));

            // Write string data to the file
            writer.write(data);
            System.out.println("Data written to example.txt");
            Writer.close();
        }
        catch (IOException e)
        {
          e.printStackTrace();
        }
    }
}
```

Step 2: Reading Data from the Text File

```
import java.io.*;

public class CharacterStreamReadExample
{
    public static void main(String[] args)
    {
        try
        {
            BufferedReader reader = new BufferedReader(new
                                              FileReader("example.txt"));

            String line;
            while ((line = reader.readLine()) != null)
            {
                System.out.println("Data from example.txt: " +line);
            }
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }
    }
}
```

Topic-5: Scanner class

The `Scanner` class in Java can be used with various I/O streams, allowing you to read input not only from the console but also from files and other data sources.

Key Features

- **Input Parsing**: Can parse primitive types and strings using regular expressions.
- **Flexible Input Sources**: Can read from various input sources, such as `InputStream`, `File`, and `String`.
- **Delimiter Support**: Allows you to specify custom delimiters for input.

Common Methods

- `next()`: Scans the next token of input as a string.
- `nextLine()`: Advances the scanner past the current line and returns the input that was skipped.
- `nextInt()`: Scans the next token as an `int`.
- `nextDouble()`: Scans the next token as a `double`.
- `hasNext()`: Returns `true` if the scanner has another token.
- `close()`: Closes the scanner and releases any associated resources.

Example-1: Using the Scanner Class to read input from the console

```
import java.util.Scanner;

public class ScannerExample
{
    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);

        // Reading a string
        System.out.print("Enter your name: ");
        String name = sc.nextLine();

        // Reading an integer
        System.out.print("Enter your age: ");
        int age = sc.nextInt();

        // Reading a double
        System.out.print("Enter your height in meters: ");
        double height = sc.nextDouble();

        // Output the collected information
        System.out.println("Hello, " + name + "!");
        System.out.println("You are " + age + " years old.");
        System.out.println("Your height is " + height + " meters.");

        // Close the scanner
        sc.close();
    }
}
```

Example-2: Using the Scanner Class with file input

In this example, we'll read data from a text file using the Scanner class. First, ensure you have a text file (e.g., data.txt) with the following content:

data.txt:

```
Prem 30

Hemika 25

Srujan 18
```

ScannerFileExample.java

```
import java.io.*;
import java.util.Scanner;

public class ScannerFileExample
{
    public static void main(String[] args)
    {
        File file = new File("data.txt"); // Specify the file path

        try

        {

            Scanner sc = new Scanner(file);

            while (sc.hasNextLine())

            {
                // Read the entire line
                String line = sc.nextLine();

                // Split line into parts
                String[] parts = line.split(" ");

                String name = parts[0];

                // Convert age from String to int
                int age = Integer.parseInt(parts[1]);

                System.out.println("Name: " + name + ", Age: " +age);
            }
        }
        catch (FileNotFoundException e)
        {
          System.out.println(e);
        }

    }
}
```

Topic-6: File class

Java provides various classes and methods for working with files and directories. Here's an overview:

File Class:

The `File class` in Java, located in the `java.io` package, provides an abstraction for file and directory pathnames. It enables you to create, delete, and manipulate files and directories in a platform-independent manner.

a) File class Constructors:

1. **`File(String pathname)`:** Creates a `File` object with the specified pathname.
2. **`File(String parent, String child)`:** Creates a `File` object with the specified parent directory and child file.
3. **`File(File parent, String child)`:** Creates a `File` object with the specified parent directory and child file.

b) File Properties

- **`getName()`:** Returns the file name.
- **`getPath()`:** Returns the file path.
- **`getAbsolutePath()`:** Returns the absolute file path.
- **`getParent()`:** Returns the parent directory.
- **`exists()`:** Checks if the file exists.
- **`isFile()`:** Checks if the file is a regular file.
- **`isDirectory()`:** Checks if the file is a directory.
- **`lastModified()`:** Returns the last modified time.
- **`length()`:** Returns the file size.

c) File Operations

- **`createNewFile()`:** Creates a new empty file.
- **`delete()`:** Deletes the file.
- **`renameTo(File dest)`:** Renames the file.
- **`mkdir()`:** Creates a new directory.
- **`listFiles()`:** Returns an array of files in the directory.

d) File Permissions

- **`canRead()`:** Checks if the file can be read.
- **`canWrite()`:** Checks if the file can be written.
- **`canExecute()`:** Checks if the file can be executed.

e) FileInputStream and FileOutputStream:

1. **`FileInputStream(File file)`:** Creates a file input stream.
2. **`FileOutputStream(File file)`:** Creates a file output stream.

f) FileReader and FileWriter:

1. **`FileReader(File file)`:** Creates a file reader.
2. **`FileWriter(File file)`:** Creates a file writer.

g) BufferedReader and BufferedWriter:

1. **`BufferedReader(Reader reader)`:** Creates a buffered reader.
2. **`BufferedWriter(Writer writer)`:** Creates a buffered writer.

Topic-7: Different File Operations

1. Creating a File:

Creating a new file in Java can be accomplished using the `File` class from the `java.io` package. Below is a step-by-step guide on how to create a new file, including example code.

Steps to Create a New File

1. **Import the Required Classes**: You need to import `java.io.File` and `java.io.IOException`.
2. **Create a File Object**: Instantiate a `File` object with the desired file name (and path if necessary).
3. **Use the `createNewFile()` Method**: Call this method to create the file. It returns `true` if the file was created successfully and `false` if the file already exists.
4. **Handle Exceptions**: Use a try-catch block to handle `IOException`, which can occur if an I/O error happens or if the file cannot be created.

Example:

```
import java.io.*;

public class CreateFileExample
{
    public static void main(String[] args)
    {
        // Specify the file name
        File file = new File("example.txt");

        // Attempt to create the new file
        try {
            // createNewFile() returns true if the file was created
            if (file.createNewFile())
            {
                System.out.println("File created: " + file.getName());
            }
            else
            {
                System.out.println("File already exists.");
            }
        }
        catch (IOException e)
        {
         System.out.println(e);
        }
    }
}
```

Output:

```
File created: example.txt
```

2. Writing to a File:

Writing to a file in Java can be accomplished using various classes in the `java.io` package. The most commonly used classes for writing to files are `FileWriter` and `BufferedWriter`. Below is a step-by-step guide along with example code for writing text to a file.

Steps to Write to a File

1. **Import Required Classes**: You need to import `java.io.BufferedWriter`, `java.io.FileWriter`, and `java.io.IOException`.
2. **Create a BufferedWriter Object**: Use `BufferedWriter` in conjunction with `FileWriter` to write to the file.
3. **Write to the File**: Use methods like `write()` and `newLine()` to write text and create new lines.
4. **Handle Exceptions**: Use a try-catch block to handle `IOException`, which may occur during file operations.
5. **Close the Resources**: Always close the `BufferedWriter` to free up system resources.

Example:

```
import java.io.*;
public class WriteFileExample
{
    public static void main(String[] args)
    {
        // Specify the file name
        String fileName = "example.txt";
        // Create a BufferedWriter to write to the file
        try
        {
          BufferedWriter writer = new BufferedWriter(new
                                             FileWriter(fileName));
          // Write text to the file
          writer.write("Hello, World!");
          writer.newLine(); // Write a new line
          writer.write("An example of writing to a file in Java.");
          writer.newLine();
          writer.write("Writing multiple lines is easy.");
          System.out.println("Data written to file: " + fileName);
          writer.close();
        }
        catch (IOException e)
        {
          System.out.println(e);
  }}}
```

3. Reading from a File:

Reading from a file in Java can be accomplished using various classes in the `java.io` package, with `BufferedReader` and `FileReader` being among the most commonly used. Below, is step-by-step guide and example code for reading text from a file.

Steps to Read from a File

1. **Import Required Classes**: You need to import `java.io.BufferedReader`, `java.io.FileReader`, and `java.io.IOException`.
2. **Create a BufferedReader Object**: Use `BufferedReader` in conjunction with `FileReader` to read the file.
3. **Read the File**: Use methods like `readLine()` to read the content line by line.
4. **Handle Exceptions**: Use a try-catch block to handle `IOException`, which may occur during file operations.
5. **Close the Resources**: Always close the `BufferedReader` to free up system resources.

Example:

```
import java.io.*;

public class ReadFileExample
{
    public static void main(String[] args)
    {
        // Specify the file name
        String fileName = "example.txt";

        // Create a BufferedReader to read the file
        try
        {
          BufferedReader reader = new BufferedReader(new
                                              FileReader(fileName));
            String line;

            // Read the file line by line
            while ((line = reader.readLine()) != null) {
                System.out.println("Line: " + line);
            }
           reader.close();
        }
        catch (IOException e)
        {
            System.out.println(e);
        }
    }
}
```