303.10.1 -
Java Files and Java IO

Lesson 1
Java's IO package and File Class

Learning Objectives:

In this lesson, we will demonstrate how to work with files in Java.

By the end of lesson, learners will be able to:

- Understand the Java IO Package.
- Describe how to use the File API to access a file using absolute and relative paths
- Describe the difference between getPath() and getAbsolutePath() methods provided by the API.
- Demonstrate how to access a directory
- Create a file, read a file, and write content in a file.

Outline

TEKsystems
Own change

Topic 1: Overview of Java IO Package

- The **java.io** package is primarily focused on **input** and **output** to ***files, network streams, internal memory buffers*** etc.
- The **java.io** package contains many classes that your programs can use to read data from a source and write data to a destination.
- The most typical sources and destinations of data are:
    - Files.
    - Pipes.
    - Network Connections.
    - In-memory Buffers (e.g., arrays).
    - System.in, System.out, and System.error.

Topic 2: Overview of File Class

- The **File** class of the **java.io** package is used to perform various operations on files and directories. For example, if you need to check whether a file has the appropriate directory name, you need the file class.
- We can access all the characteristics of the file from our system through the File Class.
- **FileNotFoundException exception** occurs while we try to access a file, because of following reasons:
  - If the given file is not available in the given location, this **exception** will occur.
  - If the given file is inaccessible, such as as read-only file, you can read the file but not modify the file. If we try to modify it, an **exception** will occur. If the file that you are trying to access for the read/write operation is opened by another program, this error will occur.

Topic 2a: Why use Files?

- While variables and arrays can be stored in memory, the storage gets wiped out when your program is closed. In order to permanently store data, you will need to put it inside of a file.
- Files are relatively insecure when compared to a database. Likewise, obtaining data from a file can take much more time than obtaining data from a database because no indexes exist.
- Data that must be secured (such as user information) or stored in large quantities should be placed in a database rather than a file. For anything else (e.g., properties or basic storage), files can be used.

Topic 2b: File Class Constructors

To use a file, declare a **File** variable using one of its constructors:

```
public File(String pathString)
public File(String parent, String child)
public File(File parent, String child)
// Constructs a File instance based on the given path string.

public File(URI uri)
 // Constructs a File instance by converting from the given file-URI "file://...."
```

Examples:

```
File file = new File("in.txt");     // A file relative to the current working directory
File file = new File("d:/myproject/java/Hello.java");  // A file with absolute path
File dir  = new File("c:/temp");   // A directory
```

Topic 2c: Create File Class Object

- To create an Object of File class, we need to import the **java.io.File** package first.
- Instances of a File class are immutable; once created, the abstract pathname represented by the File object will never change.
- The basic syntax for creating a file object is:

```
File <objectname> = new File(<Directory>)
```

Example: Here is how we can create objects of file class.

```
String location = “C:\\FolderName\\Subfolders_Name\\fileName.txt”;
File myObj = new File("filename.txt");
```

Note that you need to use 2 back-slashes instead of one.

Topic 2d: Absolute Path and Relative Path

In general, a path is a way to refer to a particular file or directory in a file system. There are two types of paths: ***absolute*** and ***relative***. In order to successfully locate a file within a program, we need to understand the difference between the two.

- **Absolute Path**
  - A path is absolute if it starts with the root element of the file system. In **windows os**, the root element is a drive e.g. `C:\\`, `D:\\`, while in **unix** os, it is denoted by the **"/"** character.
  - An absolute path is complete in that no other information is required to locate the file. It usually holds the complete directory list, starting from the root node of the file system until it reaches the file or directory it denotes.
- **Relative Path**
  - A relative path is a path that does not start with the root element of the file system. It is simply the path needed in order to locate the file from within the **current directory** of your program. It is not complete and needs to be combined with the current directory path in order to reach the requested file.
  - In order to construct a rigid and platform-independent program, it is a common to use a relative path when locating a file inside your program.

Topic 2e: Methods for Absolute and Relative Path

Checking if a path is relative:

- **isAbsolute():** The `isAbsolute()` method is a part of File class. This method returns whether the abstract pathname is absolute or not.

  Examples:

  ```
  File f1 = new File("file.doc");
  System.out.println(f1.isAbsolute());    //output →  false

  File f2 = new File("c:\\temp");
  System.out.println(f2.isAbsolute());  //output →  true
  ```

- Another useful method is `getAbsolutePath()`. It returns the absolute path to an instance of the file.

  Examples:

  ```
  File f1 = new File("\\directory\\file.xml");
  System.out.println(f1.getAbsolutePath());      //output →  C:\directory\file.xml

  File f2 = new File("c:\\directory\\file.xml");
  System.out.println(f2.getAbsolutePath());  //output →  c:\directory\file.xml
  ```

Topic 2f: File Class Useful Method

The File class has many useful methods for creating and getting information about files. For example:

methods	Description
List()	Returns an array of the files in the directory
getParentFile()	Returns a file pointing to the directory that contains the current file or directory.
getAbsoluteFile()	Returns another instance of file with an absolute path.
toURI()	Returns a URI (*Universal Resource Identifier*) that begins with file, which is useful to network operations.
isFile() and isDirectory()	Tells if a file points to a file or directory, respectively.
exists()	Tells if the file or directory really exists in the file system.
canRead and canWrite	Tells if you can read or write to the file, respectively.
createNewFile()	To create a new file, we can use the createNewFile() method. It returns: ● *true* if a new file is created. ● *false* if the file already exists in the specified location.
delete()	Removes the file or directory (if empty).

TEKsystems Own change

Topic 2f: File Class Useful Method (continued)

methods	Description
length()	Returns the file size in bytes.
mkdir() and mkdirs()	Creates a new directory if the file is a directory. The latter also creates parent directories if needed.
getFreeSpace()	Returns the available space in the device to where file is pointing to.
createTempFile()	Static method that returns a unique temporary file to be used by the application. The method deleteOnExit will delete the file at the termination of the program.
isHidden()	Returns *true* if a file or directory is hidden.
long lastModified()	Returns the date when the file was last modified (in milliseconds); this value can be converted into the dd-MM-yyyy HH:mm:sss format using SimpleDateFormat class.
String[] list()	Return contents of the directory in a String-array.
File[] listFiles()	Return contents of the directory in a File-array.

Example One: File Class - Accessing Directory and Files

The below example will return the List of Files in a specified Directory.

```
1.  public class FileExampleOne {
2.     public static void main(String[] args) {
3.         String path = "C:/Users/Downloads/TestingFolder";
4.         File dir = new File(path);
5.         System.out.println(dir.isDirectory());
6.
7.         File[] FilewithPath =  dir.listFiles(); // return an  Array of Files
8.         System.out.println(Arrays.toString(FilewithPath));
9.    }
10.  }
```

Note: Change directory path at line# 3 to match your PC.

Example Two: File Class - Demonstration to Methods

In this example, you will demonstrate how to use methods of file class.

```java
import java.io.File;
public class myMain {
  public static void main(String args[]) {
    String path = "C:/Users/Downloads/testingFile.txt";
File f = new File(path);
System.out.println("File Name: " + f1.getName());
System.out.println("File Path: " + f1.getPath());
System.out.println("Is path absolute: " + f.isAbsolute());
System.out.println("Return Absolute path of the File " + f.getAbsolutePath());
System.out.println("Return Parent folder of the given File " + f.getParent());
System.out.println("File is Exist " + f.exists());
System.out.println(f1.exists() ? "Yes file exists" : "file does not exist");
if(f.exists())  // return true or False
    {
   System.out.println("File is Found");
   System.out.println("IS file Readable " +  f.canRead());
   System.out.println("IS file Writeable " +  f.canWrite());
   System.out.println("File size in a bytes " + f.length()); // file size in byte
    }
  }
}
```

Click here to Download Dummy file

Output

```
File Name: TestingFolder
Path: C:\Users\Downloads\TestingFolder
Abs Path: C:\Users\Downloads\TestingFolder
Parent: C:\Users\Downloads
exists
is writeable
is readable
is
might be a named pipe
is absolute
File last modified: 1661278597935
File size: 8192 Bytes
```

TEKsystems Own change

Example: Creating a New File

In this program, you will demonstrate how to create a new file by using File class.

```
import java.io. * ;
public class FileClassDemo {
 public static void main(String[] args) {
   try {
    File f = new File("./Datafile.txt");
     if (f.createNewFile()) {
       System.out.println("New File created!");
     } else {
       System.out.println("The file already exists.");
     }
   } catch(IOException e) {
     e.printStackTrace();
   }
 }
}
```

Explanation

- ❖ The ***createNewFile()*** method creates a new, empty file if the file with the specified name does not already exist.
- ❖ It returns true if the file is created successfully; otherwise, it returns false.
- ❖ The method throws an IOException if an I/O error occurs.

Output:

```
New File created!
```

Topic 2g: Creating Directories/Folders

- You can use the *mkdir*() or *mkdirs*() methods to create a new directory.
- The *mkdir*() method creates a directory only if the parent directory specified in the pathname already exists.
- The *mkdir*() method creates a directory returning *true* on success and *false* on failure.
- For example, if you want to create a new directory called "TestingFoler" in the users directory in the **C: drive** on Windows, you construct the File object representing this pathname:

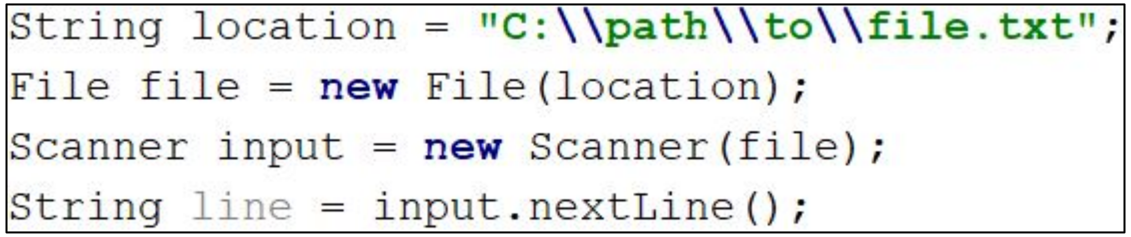
```
File newDir = new File("C:/Users/Folder");
```

Example Three: Creating Directory/Folder using **File** Class

```
import java.io.File;
public class Main {
   public static void main(String args[]) {
      File f1 = new File("C:/Downloads/TestingFolder");
      f1.mkdir();
        }
 }
```

Topic 2h: Reading File Content

- One approach to reading a file is to make use of a **Scanner** Class. A Scanner Class supports tokens for all of the Java language primitive types (except for char).
- Usually, we pass ***System.in*** to the Scanner in order to read user Input.
  - Now, instead, we are going to pass in our ***file location.***
  - The Scanner class provides methods like **hasNextLine()** and **nextLine()**, which can be used to read a file line-by-line until an "End of File" (EOF) character is reached.
  - To read a single character, we can use **`next().charAt(0)`**.
    - The **`next()`** method returns the next token/word in the input as a string and **`charAt(0)`** method returns the first character in that string.
- The code to read a line using a Scanner would look like this:



```
String location = "C:\\path\\to\\file.txt";
File file = new File(location);
Scanner input = new Scanner(file);
String line = input.nextLine();
```

Example: Reading a File Content Using File Class and Scanner Class

Click here - Download the Dummy file (testingFile.txt).

Create, a class named **`ScannerExample`** or give any name to the class.
Below is the complete code example of using `Scanner` to read text files in Java.

```
 1.import java.io.File;
 2.import java.io.FileNotFoundException;
 3.import java.util.Scanner;
 4.public class ScannerExample {
 5.   public static void main(String[] args) {
 6.       // TODO Auto-generated method stub
 7.       try {
      //------- change the location of the file-----.
 8.    File file = new File file = new File("C:/Users/testingFile.txt");
 9.           Scanner sc = new Scanner(file);
10.           String data = "";
11.           while (sc.hasNextLine()) {
12.               data = sc.nextLine();
13.               System.out.println(data);  }
14.       }
15.       catch (FileNotFoundException e)
16.       {
17.System.out.println("Either file is not found or file is not able to access");
18.           System.out.println(e.getStackTrace());
19.           e.printStackTrace();
20.       }
21.   }
22.}
```

Output: The file's content will appear in the console. If the path is incorrect, you will receive the error message: *"Either file is not found or file is not able to access."*

In this example, we will use java.util.Scanner class to read the file, line-by-line in Java. First, create a File instance to represent a text file in Java, and then we will pass this File instance to `java.util.Scanner` for scanning.

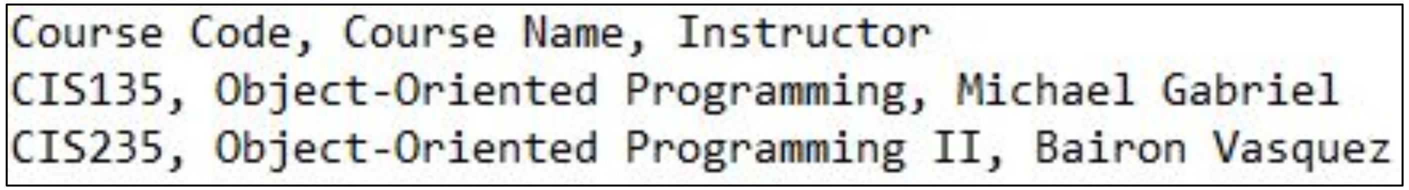
`The scanner` provides methods like `hasNextLine()` and `readNextLine()`, which can be used to read the file line-by-line. It is advised to check for the next line before reading it to avoid "FileNotFoundexception."

The **hasNext()** method verifies whether the file has another line, and the nextLine() method reads and returns the next line in the file.

Note: Do not forget to change the path or location at line number 8.

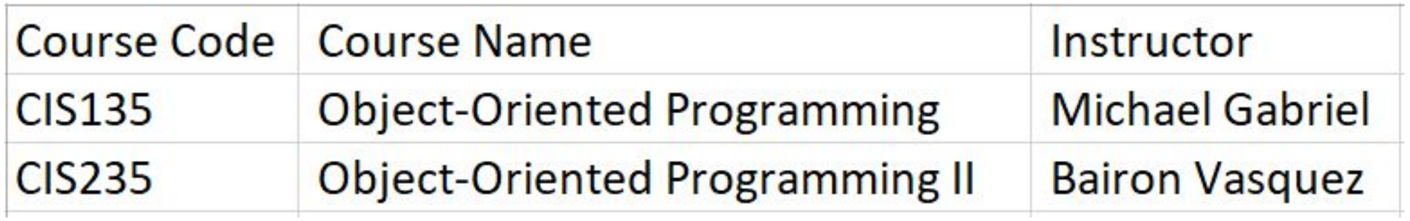
Topic 2i: Reading a Delimited File

- A delimited file is a file that has been separated into columns and rows (e.g., a table). The character separating each column is known as the "***delimiter***," and each row is on a new line.
- For instance, assume that the content below is a file separated by commas (also known as a CSV or Comma Separated Values file):



```
Course Code, Course Name, Instructor
CIS135, Object-Oriented Programming, Michael Gabriel
CIS235, Object-Oriented Programming II, Bairon Vasquez
```

- The CSV file can be read using something similar to Excel® in order to display the data as a table.



Course Code	Course Name	Instructor
CIS135	Object-Oriented Programming	Michael Gabriel
CIS235	Object-Oriented Programming II	Bairon Vasquez

- So, let's think for a moment about how we could make use of this.

Reading a Delimited File (continued)

- For separating a Delimited file, we can use:
  - String class - has a **split()** method to identify the comma delimiter and split the row into fields.
  - Scanner class - has a **useDelimiter()** to identify the comma delimiter and split the row into fields.

Hands-On Lab

Find Lab - 303.10.1- Reading a Delimited File on Canvas under the **Guided Lab section.**

If you have any technical questions while performing the lab activity, you can ask the instructors for assistance.

Topic 2j: Deleting File Using File Class

- If your application needs to create temporary files for some application logic or unit testing, you would need to make sure that these temporary files are deleted when they are not needed.
- Use the ***delete()*** method of the File class to delete a file/directory. This method returns *true* if the file/directory is deleted; otherwise, it returns *false*.
- You can also delay the deletion of a file until the JVM terminates by using the ***deleteOnExit()*** method.

```
File demoFile = new File("demo.txt");

// To delete the demo.txt file immediately
demoFile.delete();

// To delete the demo.txt file when the JVM terminates
 demoFile.deleteOnExit();
```

Knowledge Check

1. What package holds the File class?
2. Is the following path a relative path or an absolute path?

   C:\MyFiles\Programs\Examples\someFile.txt
3. What method of File class is used to test if a file or directory exists?
4. What File class method creates a new disk directory?

Lesson 2

Java IO Streams

Learning Objectives:

In this lesson, we will explore how to work with Java IO Stream classes.

By the end of lesson, learners will be able to:

- Describe Java IO Stream.
- Describe byte-based Stream and Character-based Stream.
- Explain Java IO - Streams Supper Classes and classes hierarchy.
- Demonstrate how to use the Java IO Streams classes.

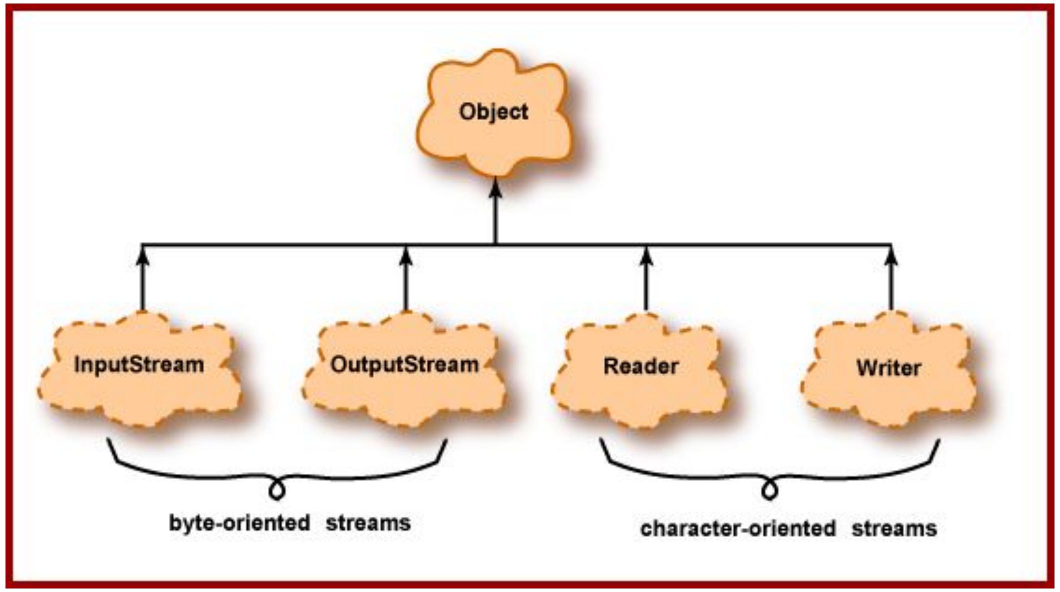
Outline/Agenda

TEKsystems Own change

Topic 1
Java I/O - Stream

Topic 1a: Java I/O - Stream

- **Java I/O Streams:** a core concept in Java IO. A stream is a conceptually **endless flow of data**. You can either **read from a stream** or **write to a stream**. A stream is connected to a data source or a data destination.
- Streams in Java IO can be either **byte-based** (reading and writing bytes) or **character-based** (reading and writing characters).
- Streams do **not** store elements; they simply convey elements from a source such as a data structure, an array, or an I/O channel through a pipeline of computational operations.
- Data that flows through a given stream has one direction only, input or output.



Source: programiz.com

For more information about I/O Stream, visit the Wiki document.

Topic 1b: Java I/O - Stream Classes - Overview

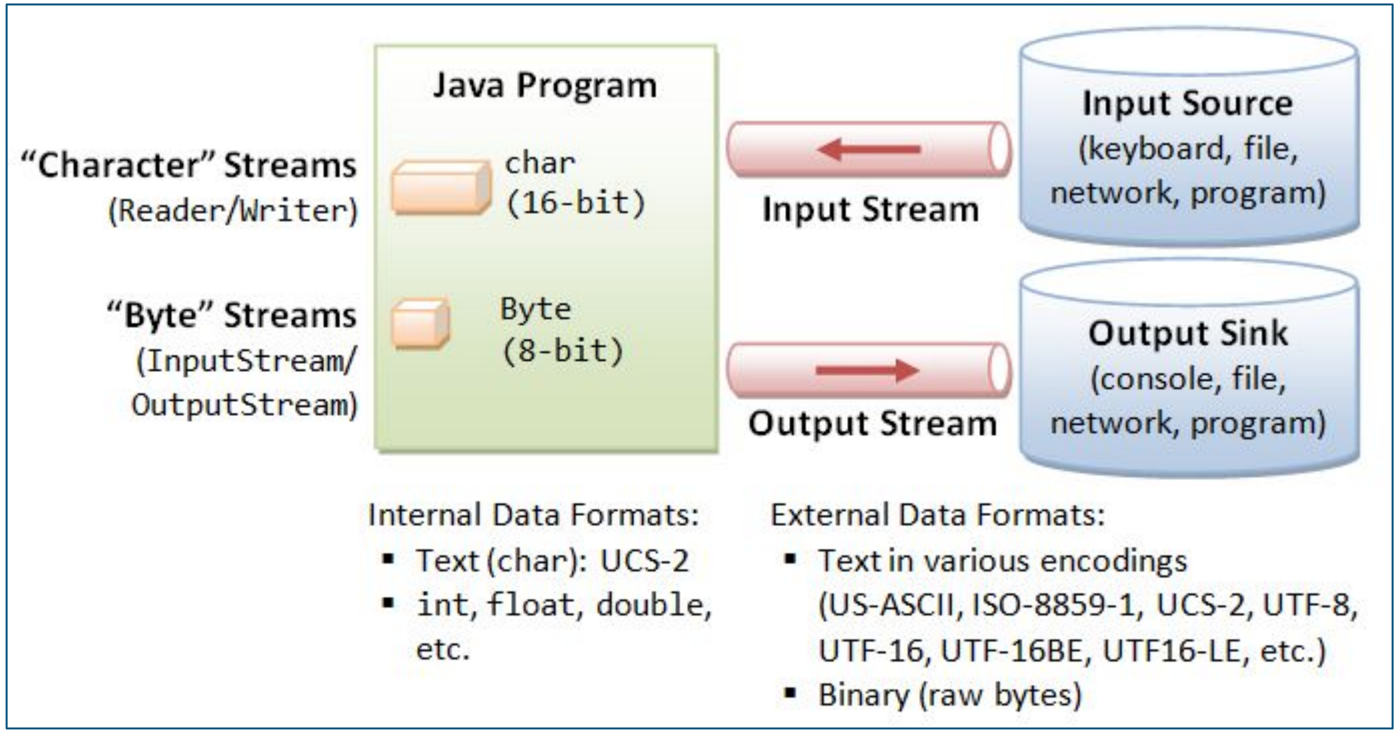
The `java.io` package contains the Java I/O stream classes. These classes are divided by **input** and **output**, as they are **byte-based Stream** and **Character-based Stream.**

- **For Byte-Based Stream:** Byte-based stream is defined by using two abstract classes at the top of the hierarchy: **InputStream** and **OutputStream** (shown in upcoming slides). These two abstract classes have several concrete classes that handle various devices such as disk files and network connections, etc.
  - **InPutStream** – The InputStream is used to read data **byte-by-byte** from a source file.
  - **OutPutStream** – The OutputStream is used for writing data to a file **byte-by-byte** to a destination.

- **For Character-Based Stream:** Character-based stream is also defined by using two abstract class at the top of the hierarchy: **Reader** and **Writer.** These two abstract classes have several concrete classes that handle unicode characters.
  - **Reader -** The Reader is used to read data **character-by-character.**
  - **Writer -** is used to write data to a file **character-by-character.**

Topic 1: Java I/O - Stream Classes - Overview

continued



Internal Data Formats:

- Text (char): UCS-2
- int, float, double, etc.

External Data Formats:

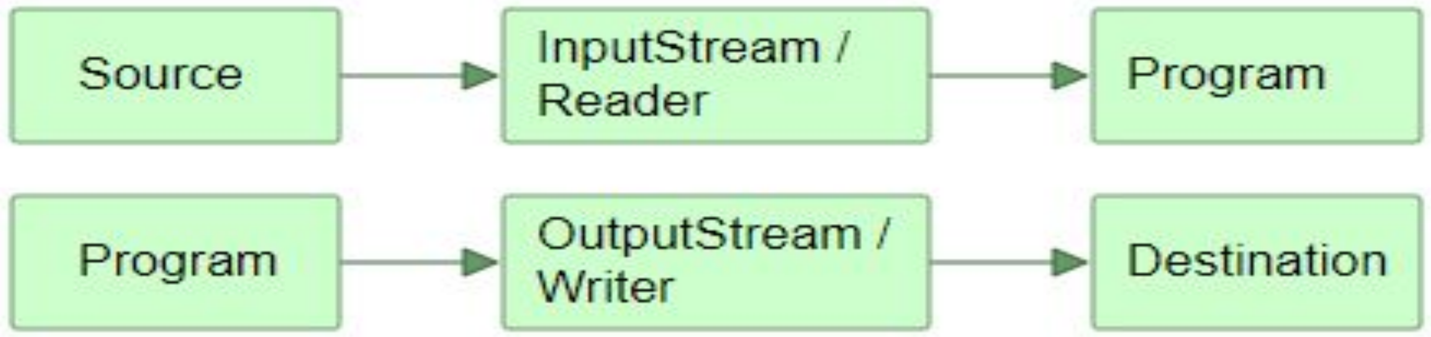
- Text in various encodings (US-ASCII, ISO-8859-1, UCS-2, UTF-8, UTF-16, UTF-16BE, UTF16-LE, etc.)
- Binary (raw bytes)

Source: math.hws.edu

Topic 2
Java I/O - Stream Super Classes

Topic 2: Java IO - Streams Super Classes - Overview

- For **byte-based Stream**, the Super classes are **InputStream** and **OutputStream.**
- For **Character-based Stream** the Super classes are **Reader** and **Writer.**
- A program that needs to read data from some source needs an **InputStream** or a **Reader,** both of which are super class. A program that needs to write data to some destination needs an **OutputStream** or a **Writer,** both of which are Super class. This is illustrated in the diagram below:
- An **InputStream** or **Reader** is linked to a source of data. An **OutputStream** or **Writer** is linked to a destination of data.



Topic 3
File Handling Using
Java IO / Character-Based Stream Classes

Topic 3a: Fundamental of Java IO Character-Based Streams

- The **Character-Based Streams** can read/write characters that are *Unicode*-based text data.
- I/O with character-based streams is no more complicated than I/O with byte streams. Input and output performed with stream classes automatically translates to and from the local character set. A program that uses character streams in place of byte streams automatically adapts to the local character set and is ready for internationalization — all without extra effort by the programmer.

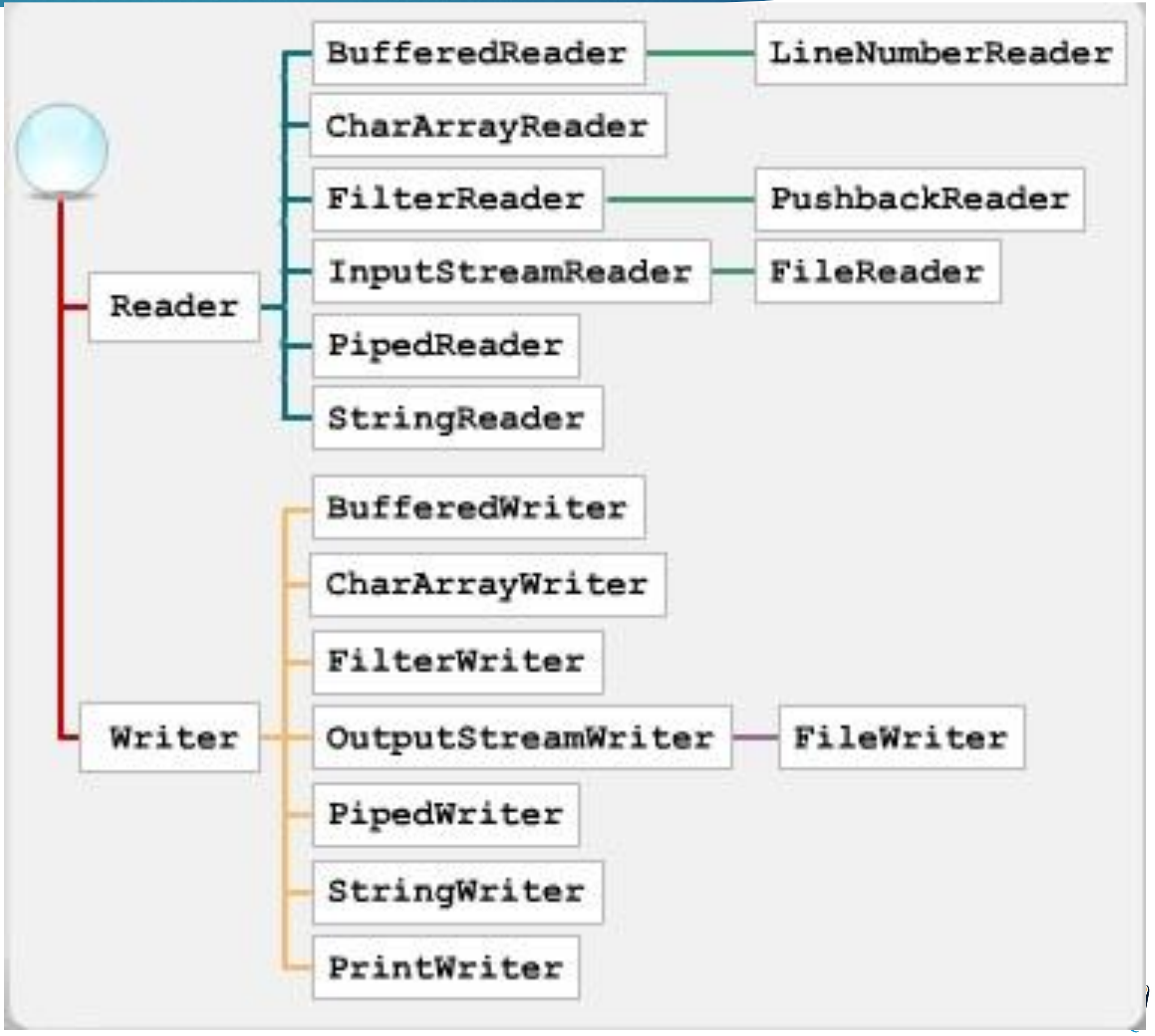
- **Why use character streams?**
  - ➢ The primary advantage of character streams is that they make it easy to write programs that are not dependent upon a specific character encoding; therefore, they are easy to internationalize.

Character-Based Stream Classes

Character-Based Stream Classes Hierarchy:

Because of limited time, we will specifically explore and demonstrate only the **FileReader() Stream class**, the **FileWriter() Stream class**, and the **PrintWriter() class.**

For more information about **Character-Based Stream Classes**, visit the **Wiki** document.



Topic 3b: Overview of FileReader() Class

The **FileReader class** is used to read data from the file, and inherits from the InputStreamReader class.

`java.lang.Object → java.io.Writer → java.io.OutputStreamWriter → java.io.FileWriter`

❖ **Constructors of FileReader class:**

FileReader(String file): Gets the filename in string. It opens the given file in read mode. If file does not exist, it throws FileNotFoundException.

❖ **Methods of FileReader class:**

- **read():** This method Reads a single character, and will block until a character is available, an I/O error occurs, or the end of the stream is reached. It is used to return a character in ASCII form, and returns -1 at the end of file.
- **close():** This method is used used to close the FileReader class.

Example: FileReader() Class Character-Based Stream

Read One Character:

Part One: In below example, we are reading the data from the text file file.txt using **FileReader** class, but we will get only one character at a time.

```
public class FileReaderExampleOne {
  public static void main(String[] args) {
   try {
    FileReader fr=new FileReader("./testingFile.txt");
           System.out.println(fr.read());
           int data = fr.read();
           System.out.println( (char)data ); // typeCasting: because
return a character in ASCII form
           fr.close();
       }
       catch (Exception e) {
           System.out.println("input file is not available");
       }}
}
```

Read All Characters:

Part Two: Let's put "**System.*out*.println(fr.read())**" in whileloop to read all characters from the file at one time.

```
public class FileReaderExampleTwo {
   public static void main(String[] args) {
       try {
           FileReader fr = new FileReader("./testingFile.txt");
           int i = 0;
           while ((i = fr.read()) != -1)   /* read() reading a single character.
return a character in ASCII form.  It returns -1 at the end of file. */ {
            //   System.out.print(i);
               System.out.print((char) i);
           }
       } catch (Exception e) {
           System.out.println("Cannot read the file");
           e.printStackTrace();
       }
   }
   }
```

Topic 3c: Overview of FileWriter() Class

`java.lang.Object → java.io.Writer → java.io.OutputStreamWriter → java.io.FileWriter`

- **FileWriter** Class is used to write character-oriented data to a file. It is a character-oriented class, which is used for file handling in Java. In FileWriter Class, we do *NOT* need to convert string into a byte array because it provides a method to write string directly.

  When instantiating a **FileWriter Class**, you need to pass it in two parameters:

  - The **File object** you are writing to, and
  - A **Boolean,**which determines if you should write over or append to a file.
    - True = Append.
    - False = Write over.
- FileWriter will create the file if it does not already exist.
- The Java **FileWriter** class also has a **`write(char[])`** method, which can write an array of characters to the destination the FileWriter is connected to. The **`write(char[])`** method returns the number of characters actually written to the FileWriter.

Topic 3c: Overview of FileWriter() Class

continued

- Unfortunately, the ***write()*** method will not automatically make a new line.
- If you would like to make a new line at the end of each line that you write(), you will need to include the new line character “\n”:

```
writer.write("Hello\n");
writer.write("World\n");
```

- The new line character “\n” will give the following result:

```
Hello
World
```

Note: You will not see the new line in Notepad. Try using your IDE!

Topic 3c: Overview of FileWriter() Class
continued

Constructor of FileWriter Class:

FileWriter(String fileName): Creates a FileWriter object using specified fileName. It throws an IOException if the named file exists.

FileWriter(String fileName, boolean append): Creates a FileWriter object using specified fileName with a boolean, indicating whether or not to **append** the data written. If the second argument is true, the data will be written to the end of the file rather than the beginning. It throws an IOException if the named file exists.

Methods of FileWriter Class:

- **write(int c):** We can send an ASCII character, which will convert into a single character and write a single character to a file.
- **write(String):** It writes a string.
- **write (char [] c) throws IOException:** It writes an array of characters.
- **fileWriter.close():** Closing a FileWriter is done by calling its close() method.

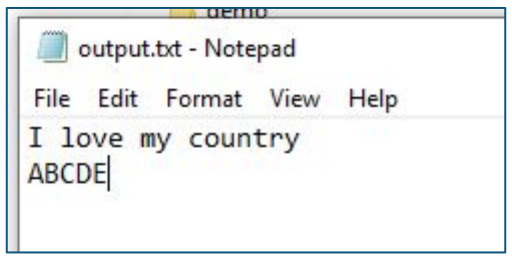
TEKsystems
Own change

Example - FileWriter() Class

In this example, we are writing the data to the text file **output.txt** using Java FileWriter class.

```java
import java.io.IOException;
import java.io.*;
public class WriteDatainFile {
   public static void main(String[] args) throws IOException {
      // Change below path as per your PC
       FileWriter fileWriter = new FileWriter("./output.txt");
       char[] chars = new char[]{'A', 'B', 'C', 'D', 'E'};
       String content = "I love my country \n";
       fileWriter.write(content);
       fileWriter.write(chars);
       fileWriter.close();
   }
}
```

Output



```
I love my country
ABCDE
```

1. Create an object of type *FileWriter*, and set it to your file path.
2. Specify your data or location of the data. In our case, just for demonstration, we are going to write simple content for data.
3. Write to the file using the *write()* method.
4. Close the writer using the *close()* method.
5. To write to a file multiple times, simply use the *write()* method, and make sure that you close the Filewriter object using the *close() method* **after** you have written **all** of your text.

Example: Read and Write Using FileReader() and FileWriter()

In the following example, we will reads data from a file using FileReader and writes the same data to another file using FileWriter classes.

```
import java.io.*;
public class ReadandWriteCharacter {
   public static void main(String args[]) throws IOException {
       FileReader in = null;
       FileWriter out = null;
       try {
   // change below file path as per your file path
           in = new FileReader("C:/Users/Downloads/testingFile.txt");
           out = new FileWriter("C:/Users/Downloads/sampleOutput3.txt");
           int c;
           while ((c = in.read()) != -1) {
               out.write(c); // writing data in file
           }
           System.out.println("Reading and Writing in a file is done!!!");
       }
       catch(Exception e) {
           System.out.println(e);
       }
       finally {
           if (in != null) {
               in.close();
           }
           if (out != null) {
               out.close();    }
       }
   }}
```

read() method:

- Reads a single character. This method will block until a character is available, an I/O error occurs, or the end of the stream is reached.
- It uses an int variable, "c," to read to and write from. The int variable holds a character value. Using an int as the return type allows read() to use -1 to indicate that it has reached the end of the stream.

For a result, the file named "sampleOutput3" should be created at specified location.

TEKsystems
Own change

Topic 3d: Overview of PrintWriter() Class

- **PrintWriter class** gives Prints formatted representations of objects to a text-output stream. It implements all of the print methods found in **System.out.println(PrintStream class)**. It does not contain methods for writing raw bytes for which a program should use unencoded byte streams.
- Unlike the **System.out.println(PrintStream class)**, if automatic flushing is enabled, it will be done only when one of the println, printf, or format methods is invoked rather than whenever a newline character happens to be output. These methods use the platform's own notion of line separator rather than the newline character.

Methods of PrintWriter

The **PrintWriter class** provides various methods that allow us to print data to the output.

print() Method

- print() - prints the specified data to the writer.
- println() - prints the data to the writer along with a new line character at the end.

Example: PrintWriter() Class

```
import java.io.PrintWriter;

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

    String data = "This is a text inside the file.";

    try {
      PrintWriter output = new
PrintWriter("C:\\folder\\output.txt");
      output.print(data);
      output.close();
    }
    catch(Exception e) {
      e.getStackTrace();
    }
  }
 }
```

In the above example, we created a PrintWriter named output. This PrintWriter is linked with the file output.txt. To print data to the file, we used the print() method. When we run the program, the output.txt file is filled with the following content.

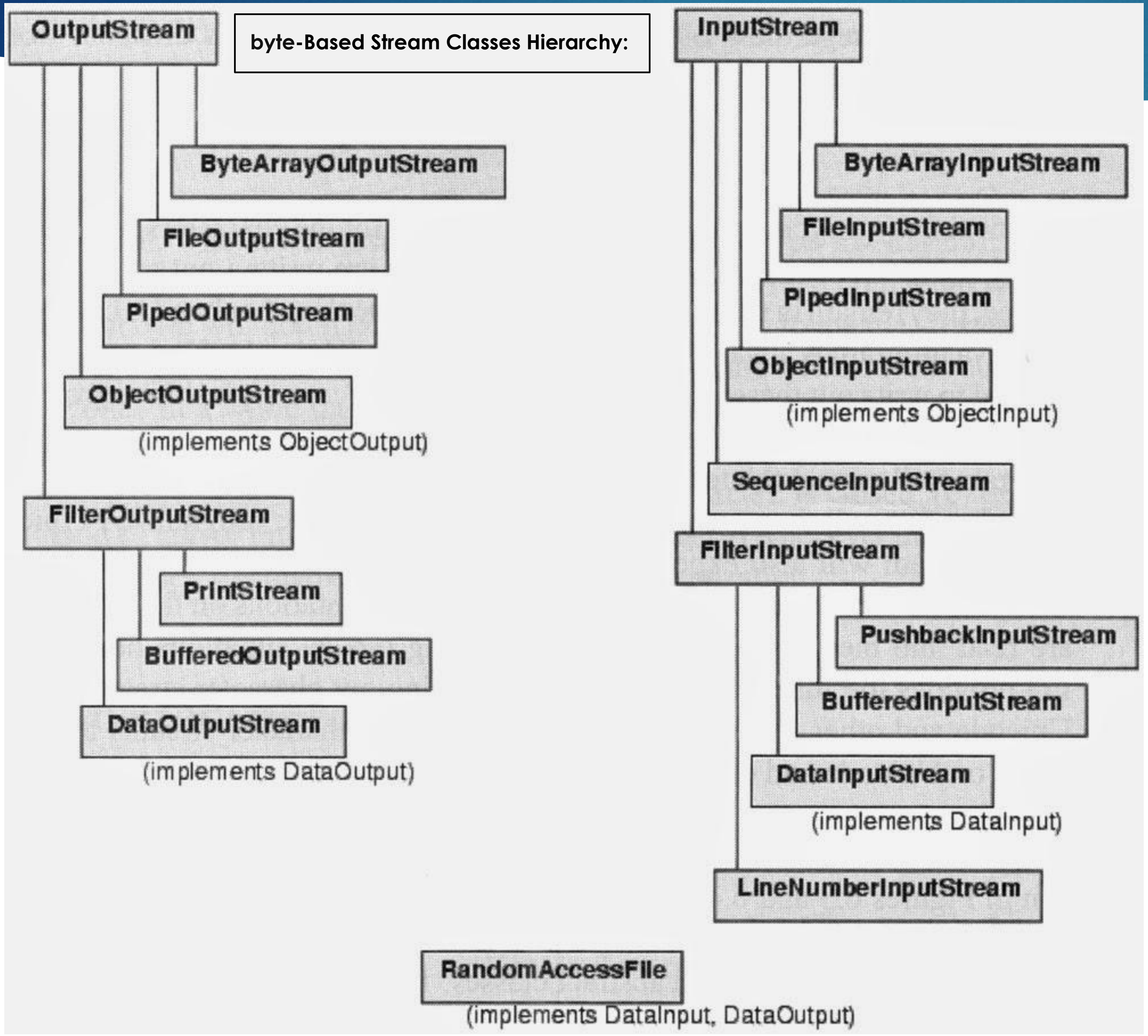
```
This is a text inside the file.
```

Topic 4
File Handling Using
Java I/O
Byte-Based Stream Classes

Topic 4a: Overview to Java I/O Byte-Based Streams

- A **Byte-Based Streams** accesses the file byte-by-byte.
- Java programs use byte-based streams to perform input and output of 8-bit bytes. It is suitable for any kind of file; however, not quite appropriate for text files. For example, if the file is using a unicode encoding, and a character is represented with two bytes, the byte stream will treat these separately and you will need to do the conversion yourself.
- Byte-oriented streams do not use any encoding scheme while character-oriented streams use character encoding scheme (UNICODE).

Topic 4b: Byte-Based Stream Classes Hierarchy



Byte-Based Stream classes are used to perform reading and writing of 8-bit bytes.

Java further divides byte stream classes into two classes: **InputStream** class and **OutputStrearn** class. The subclasses of **InputStrearn** class contains methods to support input, and the subclasses of **OutputStrearn** class contain output related methods.

Because of limited time, we will specifically explore and demonstrate:
FileInputStream() class, and
FileOutputStream() class.

Take time to read this article on your own time:
http://ecomputernotes.com/java/stream/byte-stream-classes

Java IO - For Input Byte-Based Classes Hierarchy

Byte-Based Classes Hierarchy for Input

InputStream Class: Is the Superclass of all classes representing an input stream of bytes. All of the methods of this class throw an IOException.

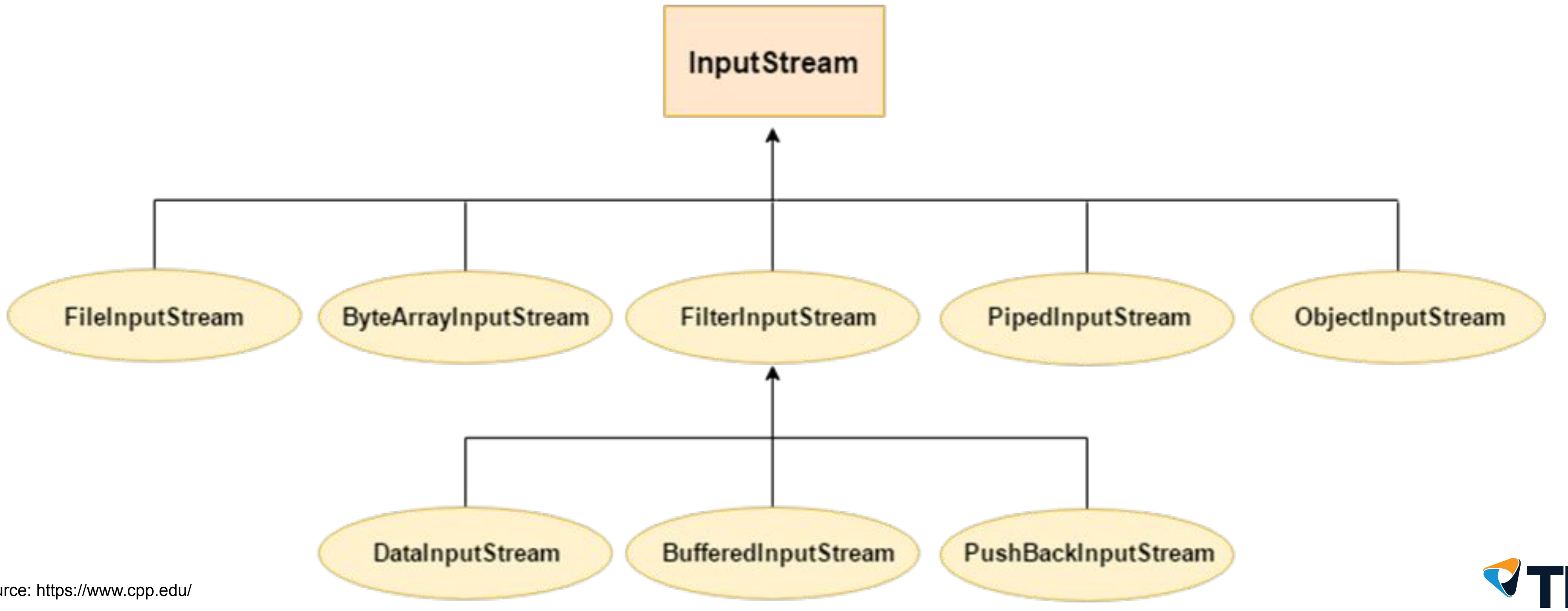


Image source: https://www.cpp.edu/

Java IO - For Output Byte-Based Classes Hierarchy

Byte-Based Classes Hierarchy for Output

OutputStream Class: Is the Superclass of all classes representing an output stream of bytes. An output stream accepts output bytes and sends them to some destination. All of the methods of this class throw an IOException.

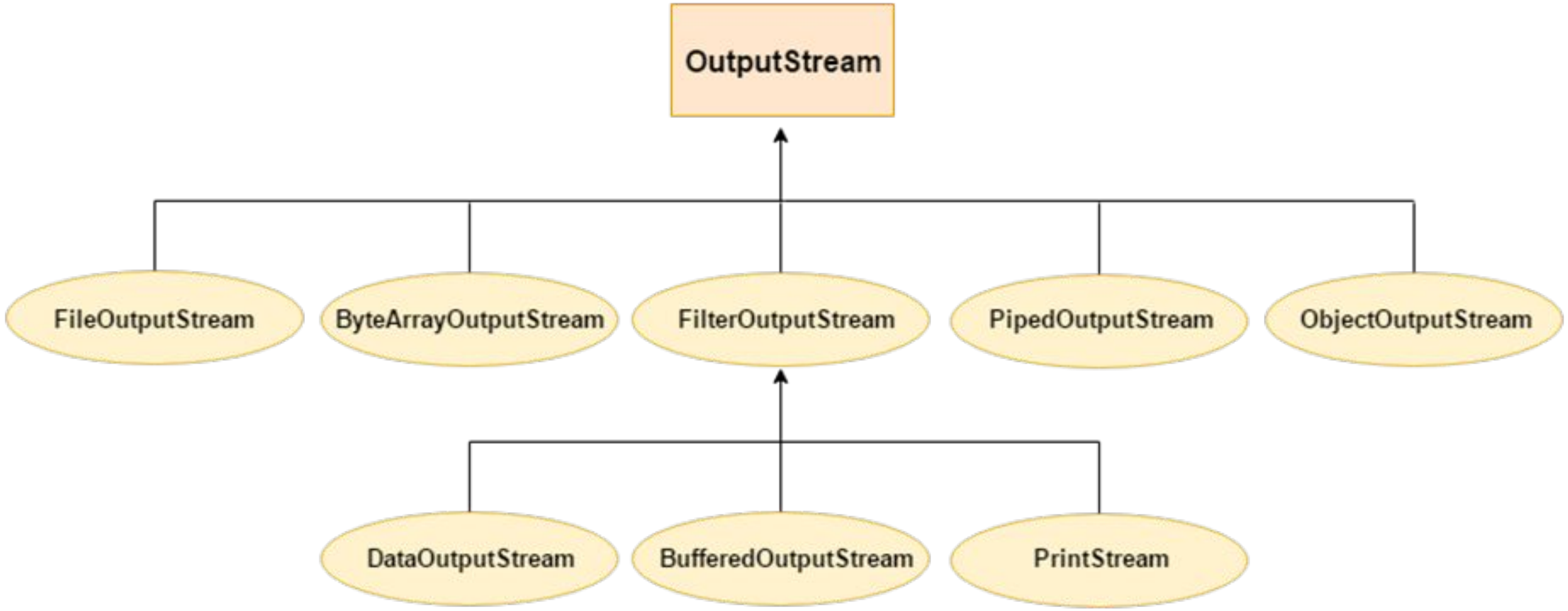


Image source: https://www.cpp.edu/

Topic 4e: Overview of Byte-Based Stream Classes

FileInputStream() - A built-in class in Java that allows reading data from a file. This class has implemented based on the byte-based stream. FileInputStream is meant for reading streams of raw bytes such as image data. The FileInputStream class provides a method **read()** to read data from a file byte-by-byte.

```
java.lang.Object → java.io.InputStream → java.io.FileInputStream
```

FileOutputStream() - A built-in class in Java that allows writing data to a file. This class has implemented based on the byte-based stream. The FileOutputStream class provides a method **write()** to write data to a file byte-by-byte.

```
java.lang.Object →  java.io.OutputStream → java.io.FileOutputStream
```

Example: FileInputStream() Class

Streaming Through the File

Create a simple file named sourcefile.txt and write the code below:

```
import java.io.*;
public class FileinputStreamExample {
   public static void main(String args[]) throws IOException {
       try {
           FileInputStream fin = new
FileInputStream("C:/Downloads/testingFile.txt");
           int i = 0;
           // System.out.println(fin.read());
           while ((i = fin.read()) != -1) {
               System.out.print((char) i);
           }
       } catch (IOException e) {
           System.out.println(e);
       }
   }
}
```

read() return type is int, although it returns a byte value.

It returns-1 if the end of the file is reached, indicating that there are no more bytes to read.

To read all bytes in a Java InputStream, you must keep reading until the value -1 is returned.This value means that there are no more bytes to read from the InputStream.

Example: FileInputStream and FileOutputStream Classes

In this example. We will read the data from a file and write the same data to another file using **FileInputStream** and **FileOutputStream** classes.

```java
import java.io.*;
public class ExampleBytebasedStream {

   public static void main(String[] args)throws IOException {
       FileInputStream in = null;
       FileOutputStream out = null;
       try {
           in = new FileInputStream("C:/Users/Downloads/testingFile.txt");
           out = new FileOutputStream("C:/Users/Downloads/sampleOutput4.txt");
           int c;
           while ((c = in.read()) != -1) { // read byte by byte
               out.write(c); // write byte by byte}
               System.out.println("Reading and Writing has been success!!");
           }
       }
           catch(Exception e){
               System.out.println(e);
           }finally {
               if (in != null) {
                   in.close();
               }
               if (out != null) {
                   out.close();
               }  }
       }
   }
```

The write() method takes an int, which contains the byte value of the byte to write, and writes the single byte to the file output stream.

Topic 4f: Classes
List for Byte-Based and Character-Based Streams

Where we can utilized all of these classes:

	Byte Based Classes		Character Based Classes	
Areas of usage	Input	Output	Input	Output
Basic	InputStream	OutputStream	Reader InputStreamReader	Writer OutputStreamWriter
Arrays	ByteArrayInputStream	ByteArrayOutputStre am	CharArrayReader	CharArrayWriter
Files	FileInputStream RandomAccessFile	FileOutputStream RandomAccessFile	FileReader	FileWriter
Buffering	BufferedInputStream	BufferedOutputStrea m	BufferedReader	BufferedWriter

Topic 4f: Classes List for Byte-Based and Character-Based Streams

continued

	Byte Based Classes		Character Based Classes	
Areas of usage	Input	Output	Input	Output
Filtering	FilterInputStream	FilterOutputStream	FilterReader	FilterWriter
Parsing	PushbackInputStream StreamTokenizer		PushbackReader LineNumberReader	
Strings			StringReader	StringWriter
Data	DataInputStream	DataOutputStream		
Data - Formatted		PrintStream		PrintWriter
Objects	ObjectInputStream	ObjectOutputStream		
Utilities	SequenceInputStream			

TEKsystems
Own change

Knowledge Check

- ❑ Java has two types of streams: Character Streams and Byte Streams. Why?
- ❑ What is the difference between the two types of streams?
- ❑ Can data flow through a given stream in both directions?

Lesson 3

Java IO - New I/O

Learning Objectives:

In this lesson, we will demonstrate how to work with Java IO Stream classes.

By the end of lesson, learners will be able to:

- Describe Java New I/O (NIO).
- Explain the core components of New I/O (NIO).
- Demonstrate how to read and write data in a file using Java New I/O (NIO).

Outline

Topic 1: Overview of Java New I/O

Java has provided a second **I/O system** called ***New I/O (NIO)***.

- It supports a buffer-oriented, channel-based approach for I/O operations.
- NIO was developed to allow Java programmers to implement high-speed I/O without using the custom native code.

For more information about Java **I/O NIO**, visit Wiki document.

Topic 2: Problem with Traditional Java IO Packages

- Java NIO channels are similar to streams but with a few differences. You can both read and write to a **channel,** but Streams are typically one-way (read or write).
- A **stream-oriented I/O** system deals with data one or more bytes at a time. An input stream produces one byte of data, and an output stream consumes one byte of data. The important thing is that bytes are not cached anywhere. Furthermore, we cannot move back and forth in the data in a stream.
- If we need to move back and forth in the data read from a stream, we must cache it in a buffer first. Java NIO's buffer-oriented approach is slightly different. Data is read into a buffer from which it is later processed. You can move back and forth in the buffer as you need to. This gives you a bit more flexibility during processing. However, you also need to check if the buffer contains all of the data you need in order to fully process it. Further, you need to make sure that when reading more data into the buffer, you do not overwrite data in the buffer you have not yet processed.

Topic 2: Problem With Traditional Java IO Packages

continued

The ***java.nio.file.Path*** does everything ***java.io.File*** can, but generally in a better way:

- File Features: The new classes support symlinks, proper file attributes, metadata support (think about: PosixFileAttributes), ACLs, and more.
- Usage: When deleting a file, you get an exception with a meaningful error message (no such file, file locked, etc.), instead of a simple boolean saying *false*.
- Decoupling: Enables support for in-memory file systems.

Topic 3: Core Components of Java NIO

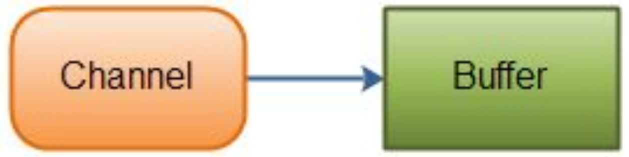
The NIO package is based on some core components used in the reading and writing operation:

1. Channels and Buffers.
2. Selectors (Thread).
3. Non-blocking I/O.

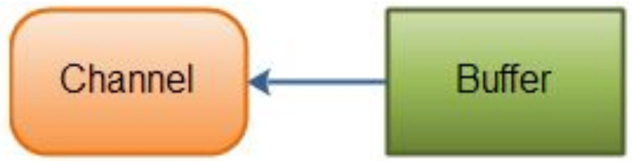
Topic 3a: Overview of Channels and Buffers

- In standard I/O API, the character streams and byte streams are used, but in NIO, we work with **channels** and **buffers**. Data is always written from a **buffer** to a **channel,** and read from a **channel** to a **buffer**.
- A **channel** is like a stream. It represents a connection between a data source and destination.
- A **Buffer** is an object, which holds some data that is to be written to or that has just been read from.
  - **Reading from channel:** We can create a buffer, and then ask a channel to read the data.
  - **Writing from channel:** We can create a buffer, fill it with data, and ask a channel to write the data.

Data reading operation



Data writing operation



For more information about **buffers**, visit the **Wiki** document.

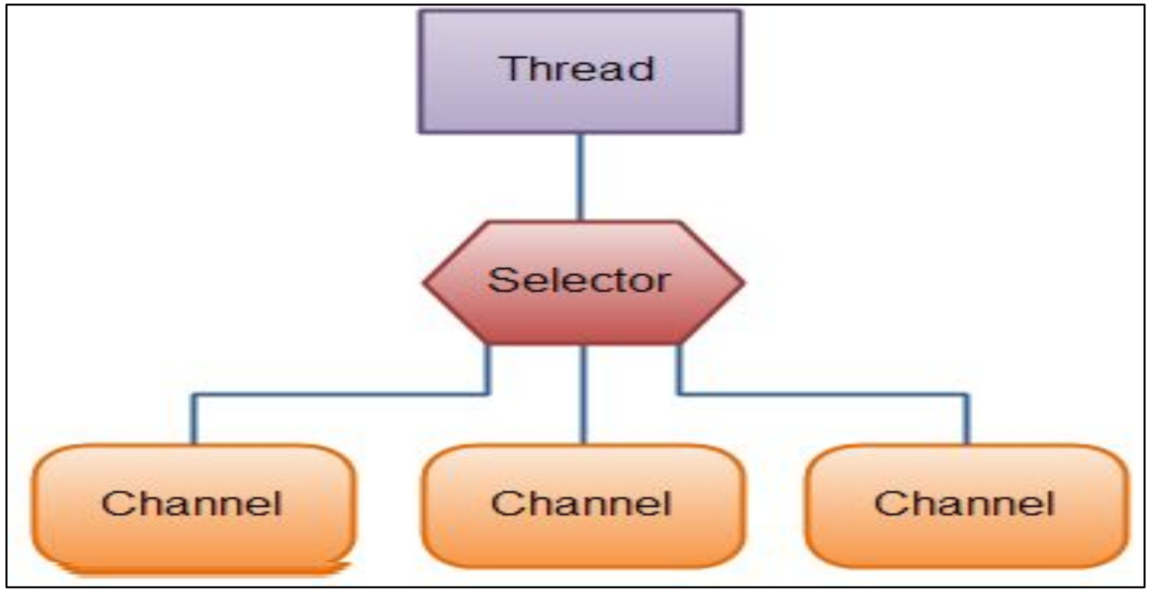
Topic 3a: Overview of Channels and Buffers continued

- In Java NIO, the primary channels used are:
  - The **FileChannel** reads data from and to files.
  - The **DatagramChannel** can read and write data over the network via UDP(User Datagram Protocol).
  - The **SocketChannel** can read and write data over the network via TCP.
  - The **ServerSocketChannel** allows you to listen for incoming TCP connections, similar to what a web server does. For each incoming connection, a **SocketChannel** is created.
- In Java NIO, ByteBuffer is not the only type of buffer. In fact, there is a buffer data type for each of the primitive Java types:
  - ByteBuffer
  - CharBuffer
  - ShortBuffer
  - IntBuffer
  - LongBuffer
  - FloatBuffer
  - DoubleBuffer

The above buffers cover the basic data types that we can send via I/O: characters, double, int, long, byte, short, and float.

Topic 3b: Overview of Selectors

- Java NIO provides the concept of "selectors." It is an object that can be used for monitoring the multiple channels for events such as data arrived, connection opened, etc. Therefore, a single thread can monitor the multiple channels for data.
- Let's see the thread using a Selector to handle the three Channel illustrations shown below:



Topic 3c: Overview of Non-blocking I/O

- Java NIO provides the feature of Non-blocking I/O.
- Non-blocking IO does not wait for the data to be read or write before returning. Java NIO non-blocking mode allows the thread to request writing data to a channel, but not wait for it to be fully written. The thread is allowed to go on and do something else in a meantime.

Topic 4: Java NIO Packages

Java NIO provides a new I/O model based on channels, buffers, and selectors. So, these modules are considered as the core of the API. The following table illustrates the list of Java.nio packages for an NIO system and why they are used:

Package	Purpose
java.nio	The top-level package for NIO system. The various types of buffers are encapsulated by this NIO system.
java.nio.charset	Encapsulates the character sets and also supports the encoder and decoder operations that convert characters to bytes and bytes to characters, respectively.
java.nio.charset.spi	Supports the service provider for character sets.
java.nio.channels	Supports the channels, which are essentially open the I/O connections.
java.nio.channels.spi	Supports the service providers for channels.
java.nio.file	Provides the support for files.
java.nio.file.spi	Supports the service providers for file system.
java.nio.file.attribute	Provides the support for file attributes.

Topic 5: FileChannel Class Implementation

- The file channel is used for reading the data from the files. It's object can be created only by calling the **getChannel()** method. But before you use a FileChannel, you *must* open it. We *cannot* create FileChannel object directly.
- So reading from a file involves three steps:
    1. Obtain a **FileChannel** via an InputStream, OutputStream, File class, Pathclass, etc.
    2. Declare and initialize the Buffer size.
    3. Read from the Channel into the Buffer.

Example: Step 1: Creating and Opening the object of FileChannel.

```
FileInputStream fis = new FileInputStream("C:/FolderName/testfile.txt");
ReadableByteChannel rbc = fis.getChannel();
```

The next step is to create a buffer:

Topic 5: FileChannel Class Implementation

continued

Step 2: Declare and initialize the Buffer size. Allocating a buffer object with size by using the allocate() method for allocating a buffer.

- Example: The allocation of ByteBuffer, with a capacity of 28 bytes:
  - **ByteBuffer buf = ByteBuffer.allocate(28);**
- Example: The allocation of CharBuffer, with space for 2048 characters:
  - **CharBuffer buf = CharBuffer.allocate(2048);**

However, the above buffer size is not a Dynamic; we can make buffer size a dynamic by finding the size of the channel, and then initializing the Buffer size per the size of the File channel, as shown below:

```
long filesize =  FileChannelObject.size();
ByteBuffer buffersize  =  ByteBuffer.allocate((int)filesize);
```

Topic 5: FileChannel Class Implementation

continued

Step 3: Reading from the Channel into the Buffer.

- To read data from a **FileChannel** into the **buffer, w**e can use the **read()** method or put method, as shown here:

```
int bytesRead = inChannel.read(buf);
```

- The **int** returned by the **read()** method tells how many bytes were written into the **Buffer**. If -1 is returned, the end-of-file is reached.

Note: There are two methods for writing the data into a Buffer:

- **read()** is a method of Filechannel class; it is used to write data from channel to buffer.
- **put()** is a method of buffer, which is used to write data in buffer.

Topic 6: Reading Data From a Buffer

- Once we read data from the Channel into the Buffer, we can access or read data from the Buffer.
- We can use two methods for reading the data from a Buffer:
  - Read the data from Buffer by using the **get()** method of **buffer class**. This method is used to read data from buffer and read one byte at a time from buffer.
  - Read the data from Buffer into a Channel by using the **write()** method of **FileChannel class.**
  - Example that reads the data from the Buffer using get() method:
    - **byte aByte = buf.get();**
  - Let's see the example that reads the data from Buffer into a channel:
    - **int bytesWritten = inChannel.write(buf);**

Topic 7: The flip() Method

The **flip()** method switches the mode of the buffer from writing, to reading mode. It also sets the position back to 0 and sets the limit to where the position was at the time of writing.

Example - NIO

```
import java.io.*;
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;
public class ChannelDemoTwo {
   public static void main(String args[]) throws IOException {
       String fname = "C:/Downloads/testingFile.txt";
       FileInputStream file = new FileInputStream(fname);
       FileChannel fileChannel = file.getChannel();
       long filesize =  fileChannel.size();
       // intializing buffer,
       // ByteBuffer buffersize  =  ByteBuffer.allocate(200);
       ByteBuffer byteBuffer  =  ByteBuffer.allocate( (int) filesize);
       // read date from a channel into buffer
       while (fileChannel.read(byteBuffer) > 0) {
           // flip the buffer to prepare for get operation
           byteBuffer.flip();
           while (byteBuffer.hasRemaining()) {
               //get() method used to read data from buffer and
               // read 1 byte at a time from buffer
               System.out.print((char) byteBuffer.get());
           }
       }
       file.close();
       fileChannel.close();
   }
}
```

In this example, we will demonstrate how to read data from a text file, how the channel is created, and how the buffer is declared. Then we will demonstrate how to read data from the buffer and print the content to the console using NIO classes and methods.

Hands-On Lab - NIO

You can find LAB - 303.10.2 - NIO on Canvas under the **Guided Lab section.**

If you have any technical questions while performing this lab activity, ask the instructors for assistance.

Knowledge Check

- ❑ State the difference between Standard IO and NIO.
- ❑ How do channels differ from streams in Java?
- ❑ What is a selector?
- ❑ What is a buffer?

Summary

- The Java program *reads from* or *writes to* a stream — one byte at a time.
- The Java NIO package provides one more utility API named *Files*, which is basically used for manipulating files and directories using its static methods, which mostly work on Path objects.
- Java NIO is a buffer-oriented approach.
- A Buffer is an object, which holds some data that is to be written to or that has just been read from. Buffers are defined inside java.nio package. It defines the core functionality, which is common to all buffers: limit, capacity, and current position.
- NIO transports the time-consuming I/O operations to the buffers, allowing threads to move to other processes; and thus, speeds up the system.
- Components of NIO in Java:
  - Channels.
  - Buffers.
  - Selectors.

References

- https://docs.oracle.com/javase/tutorial/essential/io/bytestreams.html
- https://docs.oracle.com/javase/8/docs/api/java/io/FileOutputStream.html
- https://docs.oracle.com/javase/tutorial/essential/io/charstreams.html
- https://www.ibm.com/developerworks/java/tutorials/j-nio/j-nio.html
- http://tutorials.jenkov.com/java-nio/nio-vs-io.html
- https://blogs.oracle.com/javamagazine/post/java-nio-nio2-buffers-channels-async-future-callback

Questions?

TEKsystems
Own change