Chapter-4 Java Streams and File I/O

Presentation · January 2022

CITATIONS
0

READS
4,231

1 author:

Naol Getachew
Mattu University
27 PUBLICATIONS **0** CITATIONS

SEE PROFILE

METTU UNIVERSITY

FACULTY OF ENGINEERING & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

Java programming course

Chapter-4

Java Streams and File I/O

Introduction

- What is Stream.
  - ***Streams***: is connection (logical) that allows a sequence of data moving from program to file or vice versa.
- Why IO stream
  - In java, when we store data inside variables, they kept inside RAM (i.e. lost when power off).
    - EX.

```
class TemporaryDataStoring{
    public static void main(String[] args) {
        double balance = 200.45;
        int rolno = 1234;
    }
}
```

Both, **200.45** and **1234** data is used only @execution time.

- Then the data will be removed after program execution completed.

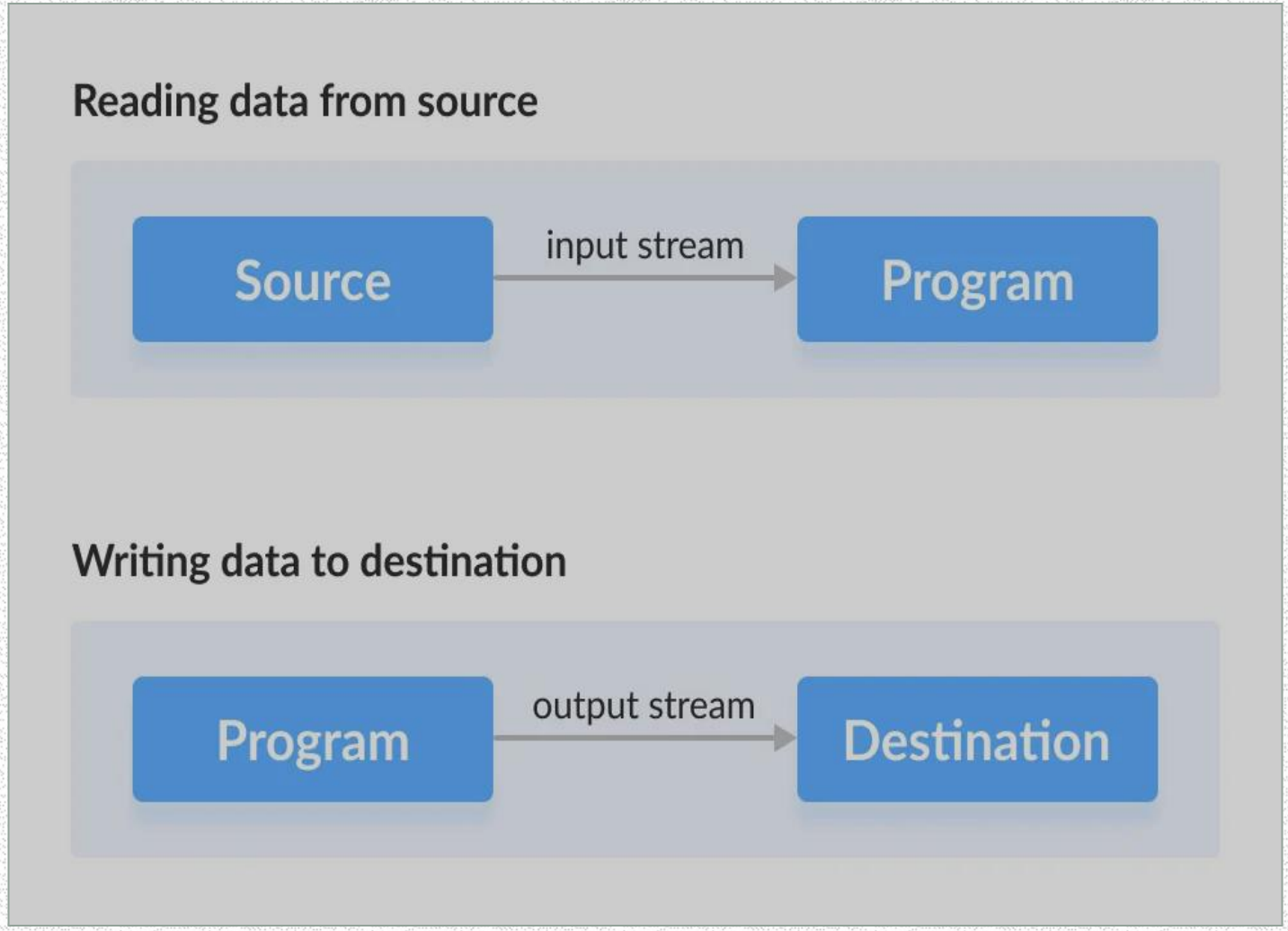
  - Advantage of IO stream:
    - ***Permanent data storage*** (On the **disk**, not on RAM)

- In Java, streams are the sequence of data that are read from the source and written to the destination.
  - An **input stream** allow java program to read data from the source (ex. File).
  - An **output stream** allow java program to write data to the destination (ex. File).

```
class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

Note:
In Hello World example, we have used **System.out** to print a string.

Here, the ***System.out*** is a type of ***output stream.***


Reading data from source
Source
input stream
Program
Writing data to destination
Program
output stream
Destination

Working with File Class

- Before doing operations (reading/writing) on file, we need to learn how to create file first.
- The **java.io.File** is the central class that works with files and directories.
- The instance of this class represents
  - the *name of a file* or *directory* on the host file system.
- When a File object is created, the system doesn't check to the existence of a corresponding file/directory.
- If the file exists, a program can examine its attributes and perform various operations on the file, such as renaming it, deleting it, reading from or writing to it.

```
File f = new File("abc.txt");
```

- This Line won't Create any Physical File, First it will Check is there any Physical File Already Available with abc.txt Name OR Not.
- If it is Already Available then f pointing to that File.
- If it is Not Already Available this Line won't Create any Physical File and just it Creates a Java File Object to Represent the Name abc.txt.

File Class Constructors:

1) File f = new File(String name);

Creates a Java File Object to Represent Name of specified File OR Directory Present in Current Working Directory.

2) File f = new File(String subdir, String name);

Creates a Java File Object to Represent Name of the File OR Directory Present in specifie Directory.

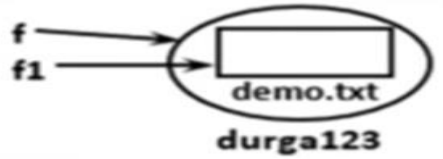
3) File f = new File(File subdir, String name);

Write a Code to Create a File Named with abc.txt in Current Working Directory

```
File f = new File("abc.txt");
f.createNewFile();
```

Write a Code to Create a Directory Named with durga123 in Current Working Directory and Create a File Named with demo.txt in that Directory

```
File f = new File("durga123");
f.mkdir();
//File f1 = new File("durga123", "demo.txt"); OR
File f1 = new File(f, "demo.txt");
f1.createNewFile();
```



Write a Code to Create a File Named with demo.txt in E:\\xyz Directory

```
File f = new File("E:\\xyz", "demo.txt");
f.createNewFile();
```

Note:- Assume that E:\\xyz Folder is Already Available in Our System.

Important Methods of File Class:

1) **boolean exists();** Returns true whether specified Physical File OR Directory exists in our system.

2) **boolean createNewFile();**
   - First this Method will Check whether the Physical File is Already Available OR Not. If it is Already Available then this Method Returns false without creating any Physical File. If it is Not Already Available, then this Method Creates New File and Returns true.

3) **boolean mkdir();**

4) **boolean isFile();** Returns true if the File Object pointing to Physical File.

5) **boolean isDirectory();** Returns true if the File Object Represents Directory.

6) **String[] list();** Returns the Names of All Files and Directories Present in specified Directory.

7) **long length();** Returns the Number of Characters Present in the specified File.

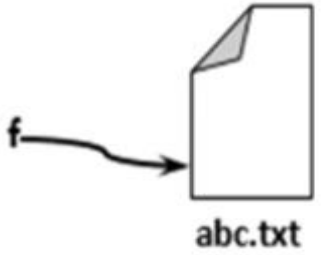
8) **boolean delete();** To Delete specified File OR Directory.

More…

- getAbsolutePath() : String
  - getAbsolutePath() returns the complete, non-relative path to the file.
- boolean f.delete() throws SecurityException
  - Deletes the file.
- boolean f.renameTo(File f2) throws SecurityException
  - Renames f to File f2. Returns true if successful.
-

```
File f = new File("abc.txt");
System.out.println(f.exists()); //false
f.createNewFile();
System.out.println(f.exists()); //true
```

First Run	Second Run
false	true
true	true



- We can Use Java File Object to Represent Directory Also.

```
File f = new File("durga123");
System.out.println(f.exists()); //false
f.mkdir();
System.out.println(f.exists()); //true
```

Note:

- **Java File IO Concept is implemented based on UNIX Operating System and in UNIX Everything is treated as a File. Hence we can Use Java File Object to Represent Both Files and Directories.**

EXAMPLE: CREATE NEW FILE

```
import java.io.*;
public class CreateFileDemo{
  public static void main(String[] args) throws IOException{
        File f;
        f=new File("myfile.txt");
        if(!f.exists()){
         f.createNewFile();
          System.out.println( "New file \"myfile.txt\" has been created " + "to the current directory");
        }
    }
}
```

Using Path Names

- ***Path name***—gives name of file and tells which directory the file is in
- ***Relative path name***—gives the path starting with the directory that the program is in.
- ***Absoulte path name***—gives the path starting with the root directory (**C:**) that the program is in.

  - **Typical UNIX path name:**
    - `/user/smith/home.work/java/FileClassDemo.java`
  - **Typical Windows path name:**
    - `D:\Work\Java\Programs\FileClassDemo.java`
    - When a **backslash** is used in a **quoted string** it must be **written as two backslashes** since backslash is the **escape character**:
      - `"D:\\Work\\Java\\Programs\\FileClassDemo.java"`
  - Java will accept path names in UNIX or Windows format, regardless of which operating system it is actually running on.

Types of Streams

- Depending upon *the* ***type of data flow within the stream***, it (stream) can be classified into:
  - Byte Stream
  - Character Stream

- Based on *the* ***direction of data flow***, it can be classified into:
  - InputStream and OutputStream (Byte Stream type) or
  - Reader and Writer (Character Stream type)

  - Note: both inputstream and Reader utilities for reading data from source to program.
  - both outputstream and Writer utilities for writing data to file from program.

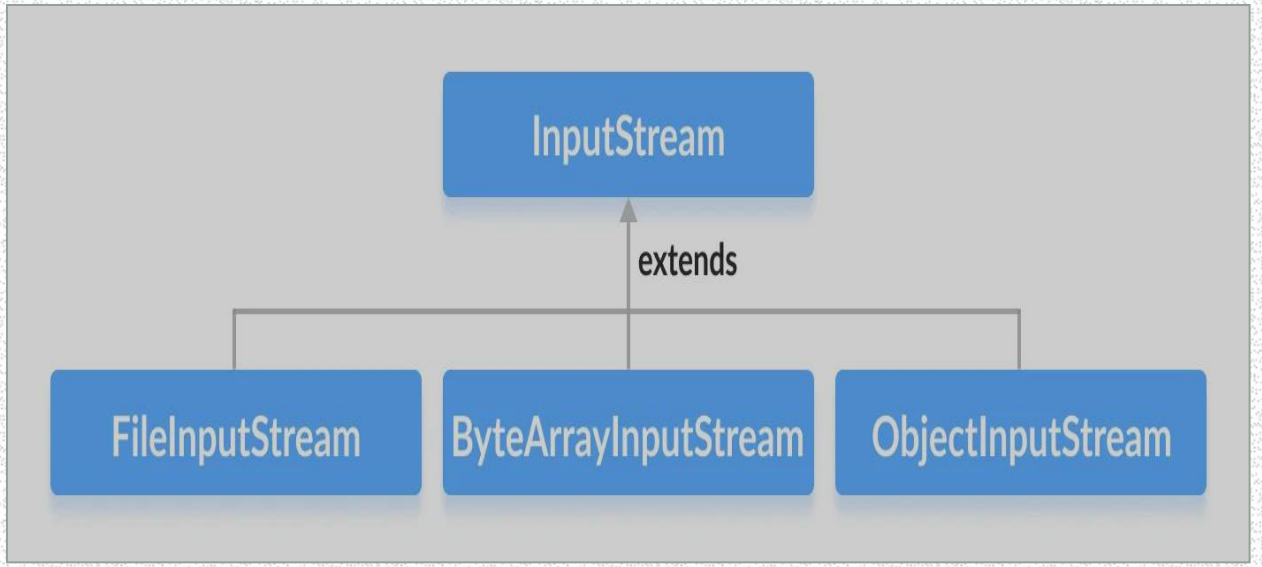
1) Byte Stream

- Byte stream is used to read and write a single byte (8 bits) of data.
- All byte stream classes are derived from base abstract classes called
  - **InputStream** (to read) and **OutputStream** (to write).

Java InputStream Class

- First, inputstream the way of reading data from source by java program.
- The InputStream class is part of the **java.io** package
  - It is an abstract superclass that represents an input stream of bytes.
- Since InputStream is an abstract class, it is not useful by itself.
  - However, its subclasses can be used to read data.

- **Subclasses of InputStream**
  - In order to use the functionality of InputStream, we can use its ***subclasses***.
  - Some of them are:
    - FileInputStream
    - ByteArrayInputStream
    - ObjectInputStream



- **Create an InputStream**
  - In order to create an InputStream, we must import the ***java.io.InputStream*** package first.
  - Once we import the package, here is how we can create the input stream.

```
// Creates an InputStream
InputStream object1 = new FileInputStream();
```

  - Here, we have created an input stream using FileInputStream. It is because InputStream is an abstract class.
    - Hence we cannot create an object of InputStream.
  - **Note**: We can also create an input stream from other subclasses of InputStream.

Methods of InputStream

- The InputStream class provides different methods that are implemented by its subclasses.
- Here are some of the commonly used methods:

- **read()** - reads one byte of data from the input stream
- **read(byte[] array)** - reads bytes from the stream and stores in the specified array
- **available()** - returns the number of bytes available in the input stream
- **mark()** - marks the position in the input stream up to which data has been read
- **reset()** - returns the control to the point in the stream where the mark was set
- **markSupported()** - checks if the mark() and reset() method is supported in the stream
- **skips()** - skips and discards the specified number of bytes from the input stream
- **close()** - closes the input stream

Example: InputStream Using FileInputStream

Suppose we have a file named ***input.txt*** with the following content.
"This is a line of text inside the input.txt file."

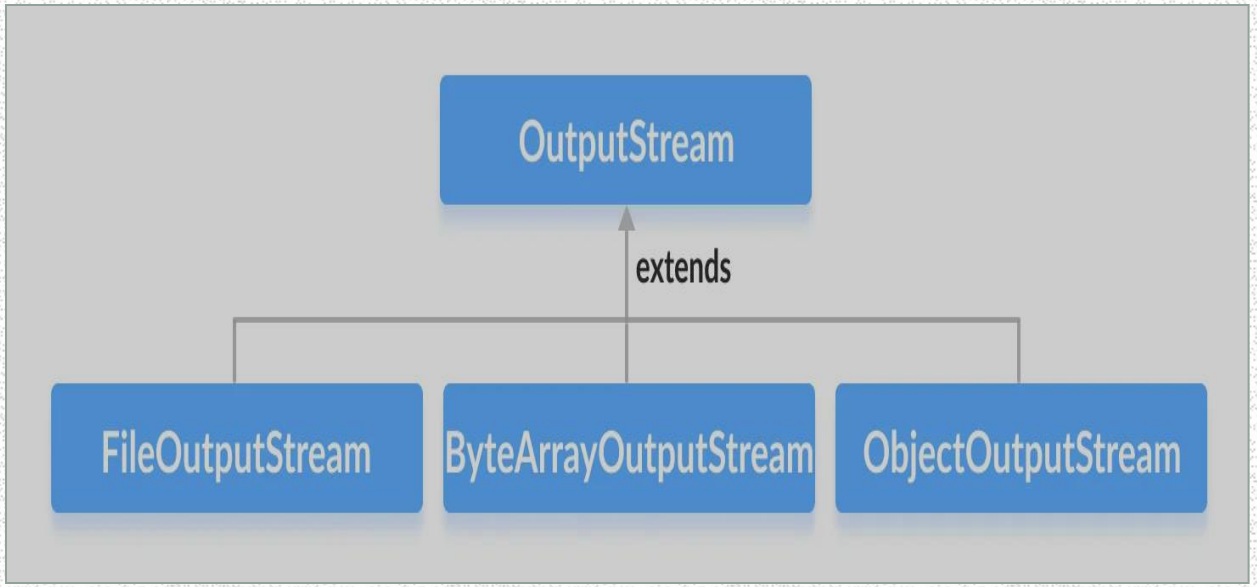
Output
Available bytes in the file is: 49
Data read from the file:
This is a line of text inside the input.txt file.

```
import java.io.FileInputStream;
import java.io.InputStream;

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

    byte[] array = new byte[100];

    try {
      InputStream input = new FileInputStream("input.txt");

      System.out.println("Available bytes in the file: " + input.available());

      // Read byte from the input stream
      input.read(array);
      System.out.println("Data read from the file: ");

      // Convert byte array into string
      String data = new String(array);
      System.out.println(data);

      // Close the input stream
      input.close();
    } catch (Exception e) {
      e.getStackTrace();
    }
  }
}
```

Java OutputStream Class

- First, outputstream the way of writing data to destination from java program.
- The OutputStream class is part of the **java.io** package
  - It is an abstract superclass that represents an output stream of bytes.
- Since OutputStream is an abstract class, it is not useful by itself.
  - However, its subclasses can be used to write data.

- **Subclasses of OutputStream**
  - In order to use the functionality of OutputStream, we can use its subclasses. Some of them are:
    - ***FileOutputStream***
    - ***ByteArrayOutputStream***
    - ***ObjectOutputStream***



- **Create an OutputStream**
  - In order to create OutputStream, we must import **java.io.OutputStream** package first.
  - Once we import the package, here is how we can create the output stream.

```
// Creates an OutputStream
OutputStream object = new FileOutputStream();
```

  - Here, we have created an object of output stream using FileOutputStream.
  - Because OutputStream is an abstract class, we cannot create an object of OutputStream.

- **Methods of OutputStream**
  - The OutputStream class provides different methods that are implemented by its subclasses. Here are some of the methods:

❑ **write()** - writes the specified byte to the output stream
❑ **write(byte[] array)** - writes the bytes from the specified array to the output stream
❑ **flush()** - forces to write all data present in output stream to the destination
❑ **close()** - closes the output stream

Example: OutputStream Using FileOutputStream

Here, we have created an output stream using the **FileOutputStream** class.

The output stream is now linked with the file **output.txt.**

```
import java.io.FileOutputStream;
import java.io.OutputStream;

public class Main {

  public static void main(String args[]) {
    String data = "This is a line of text inside the file.";

    try {
      OutputStream out = new FileOutputStream("output.txt");

      // Converts the string into bytes
      byte[] dataBytes = data.getBytes();

      // Writes data to the output stream
      out.write(dataBytes);
      System.out.println("Data is written to the file.");

      // Closes the output stream
      out.close();
    }

    catch (Exception e) {
      e.getStackTrace();
    }
  }
}
```

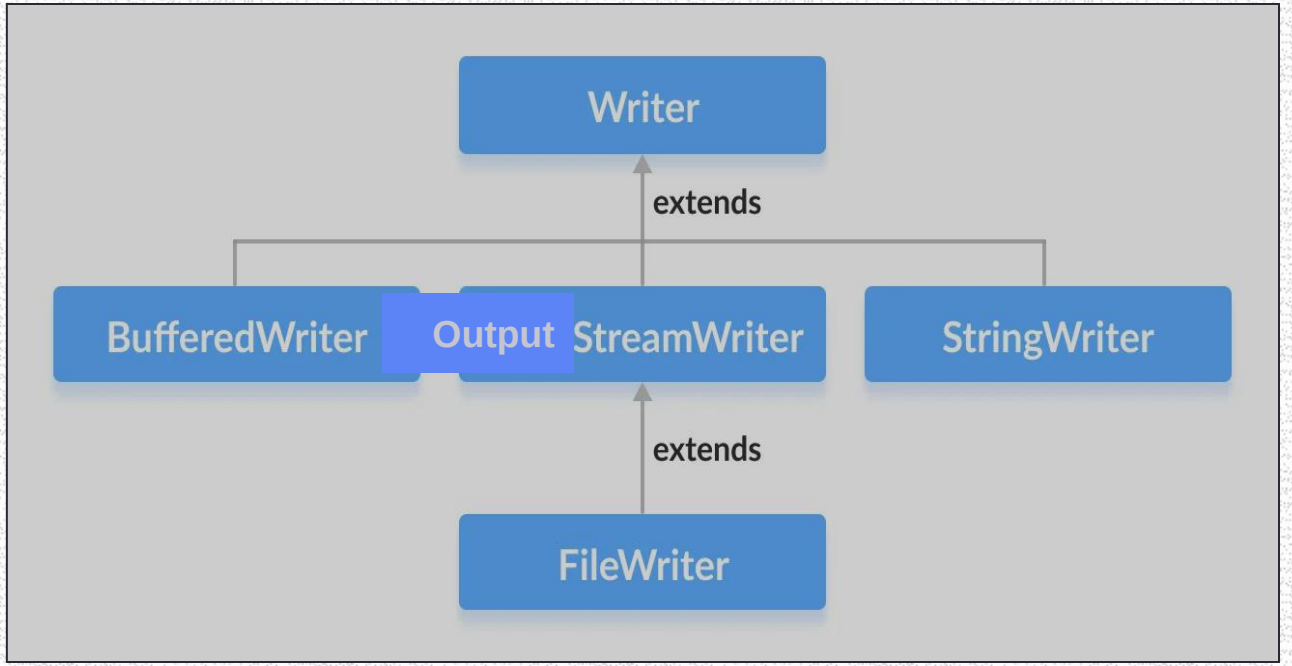
2) Character Stream

- Character stream is used to read and write a single character of data.
- All the character stream classes are derived from base abstract classes:
  - **Writer** and
  - **Reader**

Java Writer Class

- The Writer class of the **java.io** package.
  - It is an abstract superclass that represents a stream of characters.
- Since Writer is an abstract class, it is not useful by itself.
  - However, its subclasses can be used to write data.

- **Subclasses of Writer**
  - **In order to use the functionality of the Writer, we can use its subclasses. Some of them are:**
    - ***BufferedWriter***
    - ***OutputStreamWriter***
    - ***FileWriter***
    - ***StringWriter***



- **Create a Writer**
  - In order to create a Writer, we must import the **java.io.Write**r package first.
  - Once we import the package, here is how we can create the writer.

```
// Creates a Writer
Writer output = new FileWriter();
```

  - Here, we have created a writer named output using the FileWriter class.
  - It is because the Writer is an abstract class.
  - Hence we cannot create an object of Writer.

FileWriter

We can Use FileWriter Object to write Character Data(text data) to the File.

Constructors:

1) FileWriter fw = new FileWriter(String name);
2) FileWriter fw = new FileWriter(File f);

The Above 2 Constructors meant for Overriding Existing Data.
Instead of Overriding, If we want to Perform Append Operation then we have to Use the following 2 Constructors.

1) FileWriter fw = new FileWriter(String name, boolean append)
2) FileWriter fw = new FileWriter(File f, boolean append)

Note: If the specified File is Not Already Available then All the Above Constructors will Create that File.

Methods of FileWriter class:

1) **write(int ch);** To write a Single Character to the File.

2) **write(char[] ch);** To write an Array of Characters to the File.

3) **write(String s);** To write a String to the File.

4) **flush();** To give the Guarantee that Total Data including Last Character written properly to the File.

5) **close();**

Demo Program to demonistrate FileWriter:

```
import java.io.FileWriter;
import java.io.IOException;
class  FileWriterDemo {
        public static void main(String[] args) throws IOException {
                FileWriter fw = new FileWriter("abc.txt");
                fw.write(100);//Adding a Single Character
                fw.write("urga\nSoftware Solutions");
                fw.write("\n");
                char[] ch1 = {'a', 'b', 'c'};
                fw.write(ch1);
                fw.write("\n");
                fw.flush();
                fw.close();
  }
}
```

durga
Software Solutions
abc

abc.txt

- In the Above Program FileWriter Always Overrides existing Data.
- Instead of Overriding if we want Append we have to Create FileWriter Object as

```
FileWriter fw = new FileWriter("abc.txt", true);
```

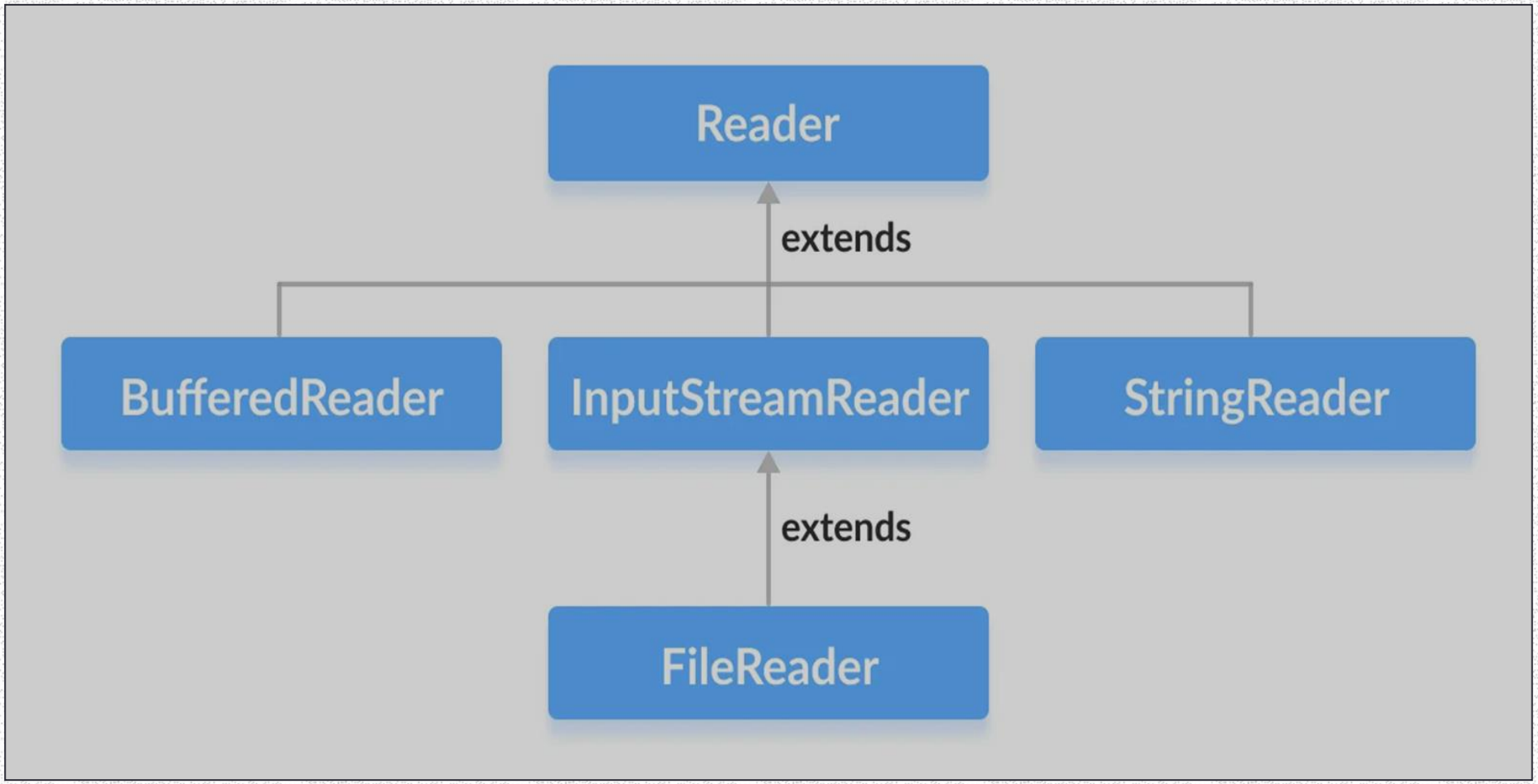
Example2: Writer Using FileWriter

```
import java.io.FileWriter;
import java.io.Writer;

public class Main {

   public static void main(String args[]) {

      String data = "This is the data in the output file";

      try {
         // Creates a Writer using FileWriter
         Writer output = new FileWriter("output.txt");


         // Writes string to the file
         output.write(data);

         // Closes the writer
         output.close();
      }

      catch (Exception e) {
         e.getStackTrace();
      }
   }
```

Java Reader Class

- The Reader class is part of the **java.io** package.
  - It is an abstract superclass that represents a stream of characters.
  - Since Reader is an abstract class, it is not useful by itself.
- However, its subclasses can be used to read data.

- **Subclasses of Reader**
  - In order to use the functionality of Reader, we can use its subclasses.
  - Some of them are:
    - **BufferedReader**
    - **InputStreamReader**
    - FileReader
    - **StringReader**


Reader
extends
BufferedReader
InputStreamReader
StringReader
extends
FileReader

- **Create a Reader**
  - In order to create a Reader, we must import the java.io.Reader package first.
  - Once we import the package, here is how we can create the reader.

```
// Creates a Reader
Reader input = new FileReader();
```

  - Here, we have created a reader using the FileReader class.
  - It is because Reader is an abstract class.
    - Hence we cannot create an object of Reader.

FileReader

We can Use FileReader to Read Character Data from the File.

Constructors:

1) FileReader fr = new FileReader(String fName);
2) FileReader fr = new FileReader(File f);

Methods:

1) int read();

- It Attempts to Read Next Character from the File and Returns its Unicode Value.
- If there is No Next Character then we will get -1.
- As this Method Returns Unicode Value Compulsory at the Time of Printing we should Perform Type - Casting.

```
FileReader fr = new FileReader("abc.txt");
int i = fr.read();
while (i != -1) {
      System.out.println( (char)i );
      i = fr.read();
}
```

Example: Reader Using FileReader

Suppose we have a file named **input.txt** with the following content.

"This is a line of text inside the file."

Output

Is there data in the stream? ***true***
Data in the stream:
This is a line of text inside the file.

```
import java.io.Reader;
import java.io.FileReader;

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

      // Creates an array of character
      char[] array = new char[100];

      try {
         // Creates a reader using the FileReader
         Reader input = new FileReader("input.txt");

         // Checks if reader is ready
         System.out.println("Is there data in the stream?  " + input.ready());

         // Reads characters
         input.read(array);
         System.out.println("Data in the stream:");
         System.out.println(array);

         // Closes the reader
         input.close();
      }

      catch(Exception e) {
         e.getStackTrace();
      }
   }
```

- **Read more on:**
  - **Subclasses of InputStream/OutputStream**
  - **Subclasses of Writer/Reader**
  - **FilterClass**

End