IO

Introduction

Java I/O (Input and Output) is used to process the input and produce the output.

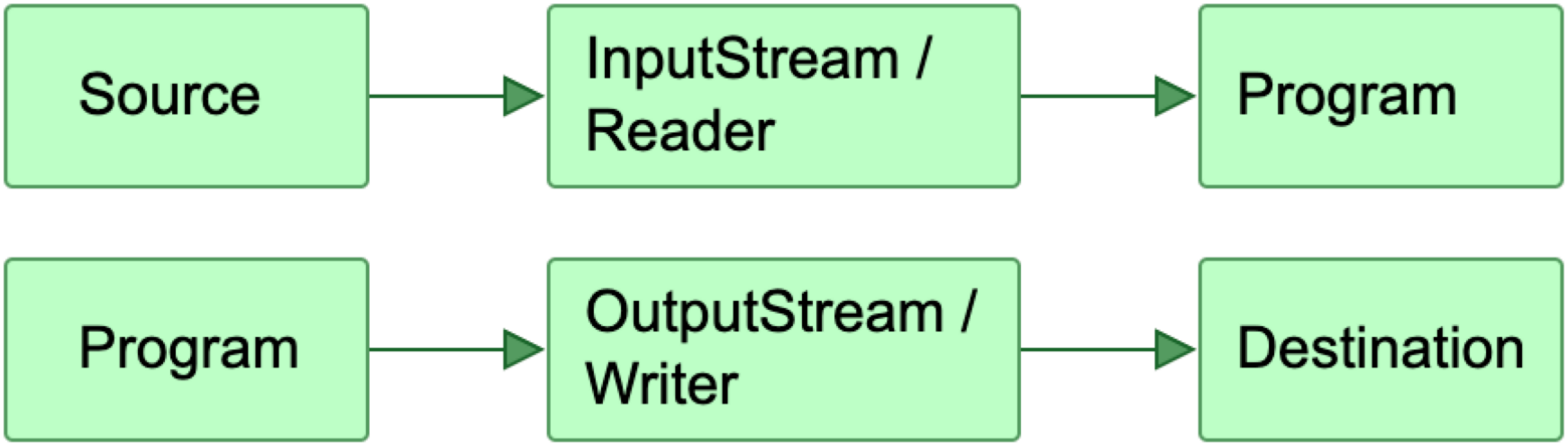
A program that needs to read data from some source needs an InputStream or a Reader. A program that needs to write data to some destination needs an OutputStream or a Writer.

Java's IO package mostly concerns itself with the reading of raw data from a source and writing of raw data to a destination. The most typical sources and destinations of data are these:

- Files
- Pipes
- Network Connections
- In-memory Buffers (e.g. arrays)
- System.in, System.out, System.error

Streams

A stream is a conceptually endless flow of data. You can either read from a stream or write to a stream. Java uses the concept of a stream to make I/O operation fast. The java.io package contains all the classes required for input and output operations. A stream has no concept of an index of the read or written data, like an array does. Nor can you typically move forth and back in a stream, like you can in an array, or in a file using RandomAccessFile. A stream is just a continuous flow of data.



InputStream

Java application uses an input stream to read data from a source; it may be a file, an array, peripheral device or socket.

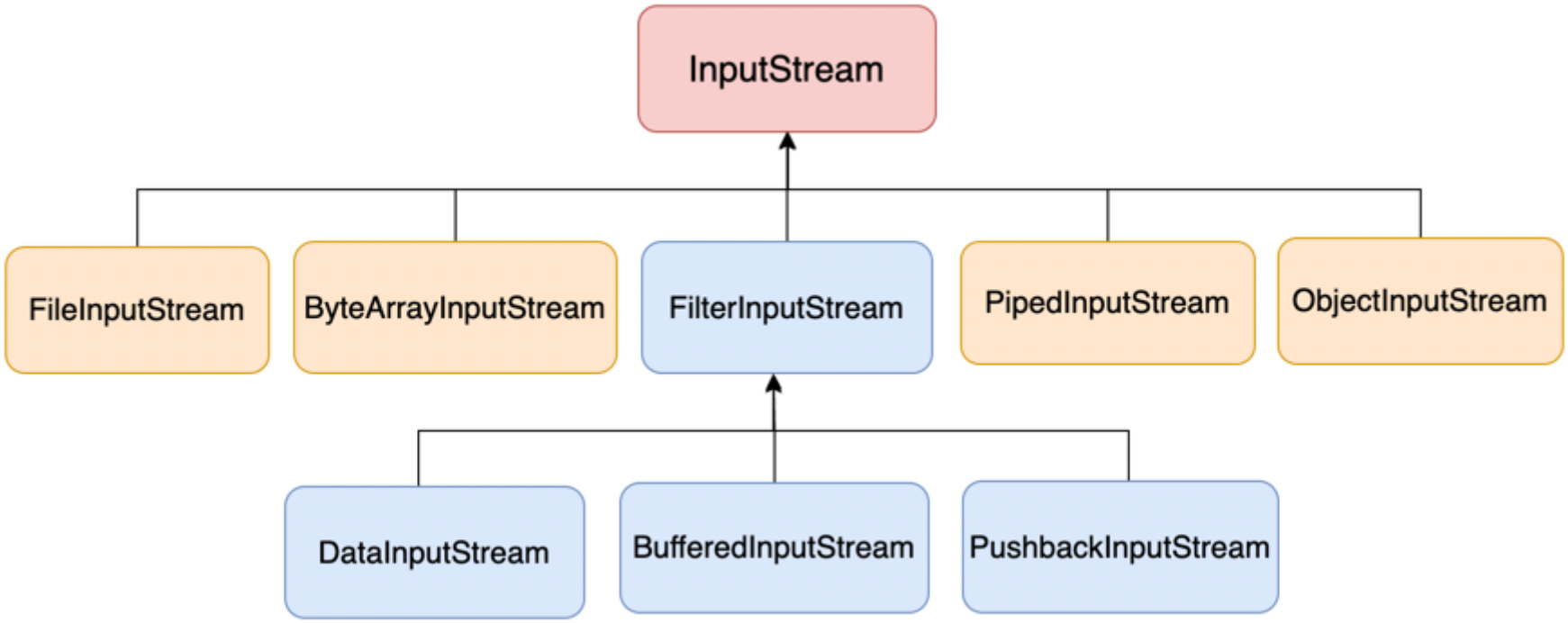
InputStream class is an **abstract** class. Since java.io.InputStream class is abstract class you have to use subclass of it in your application. It has subclasses for specified problems. For example to read from a file use FileInputStream, for in memory streaming use ByteArrayInputStream, to buffer data and read from buffer use BufferedInputStream etc.

```
InputStream fileStream=new FileInputStream("demo.txt");

InputStream arrayStream=new ArrayInputStream("A dummy text".getBytes());

InputStream buffStream=new BufferedInputStream(new FileInputStream("dome.txt"));
```

. . .

In the picture below you will see the subclass of the InputStream class.

FilterInputStream is another subclass of InputStream. It is also not used as alone for streaming. We are using subclass of FilterInputStream in our operations, which are DataInputStream, BufferedInputStream, and PushbackInputStream.



Java Input Stream Classes

Useful methods of InputStream

1. public abstract int read() throws IOException

It returns an int value containing the Byte value. The value returned is between 0 and 255 or -1. If no byte is read, the code returns -1, which indicates the end of the file.

It reads 1 byte on each read. Think that you have a file that contains 10 characters. we know each character is 1 byte. So to read all character, application will do 10 reads.

2. public int available() throws IOException

It returns the number of bytes that can be read from the input stream.

For Example, you have 10 bytes in stream. After each read the number of available byte will decrease. When it is 0 that means there is no byte to read.

```
 // initial 10 bytes data in stream

available()// // is 10
read();
available()// // is 9
.
.// 9 more read()
...
available() // is 0
```

3. public void close() throws IOException

It closes the current input stream and releases any system resources associated with it.

---

4. public int read(byte[ ] b) throws IOException

It reads the bytes from the input stream and stores every byte in the buffer array. It returns the total number of bytes stored in the buffer array. If there is no byte in the input stream, it returns -1 as the stream is at the end of the file.

5. public int read(byte[ ] b , int off , int len) throws IOException

It reads up to len bytes of data from the input stream. It returns the total number of bytes stored in the buffer. Here the “off” is start offset in buffer array b where the data is written, and the “len” represents the maximum number of bytes to read.

6. public void reset() throws IOException

It repositions the stream to the last called mark position. The reset method does nothing for input stream class except throwing an exception.

7. public long skip(long n) throws IOException

It discards n bytes of data from the input stream

OutputStream

Java application uses an output stream to write data to a destination; it may be a file, an array, peripheral device or socket.

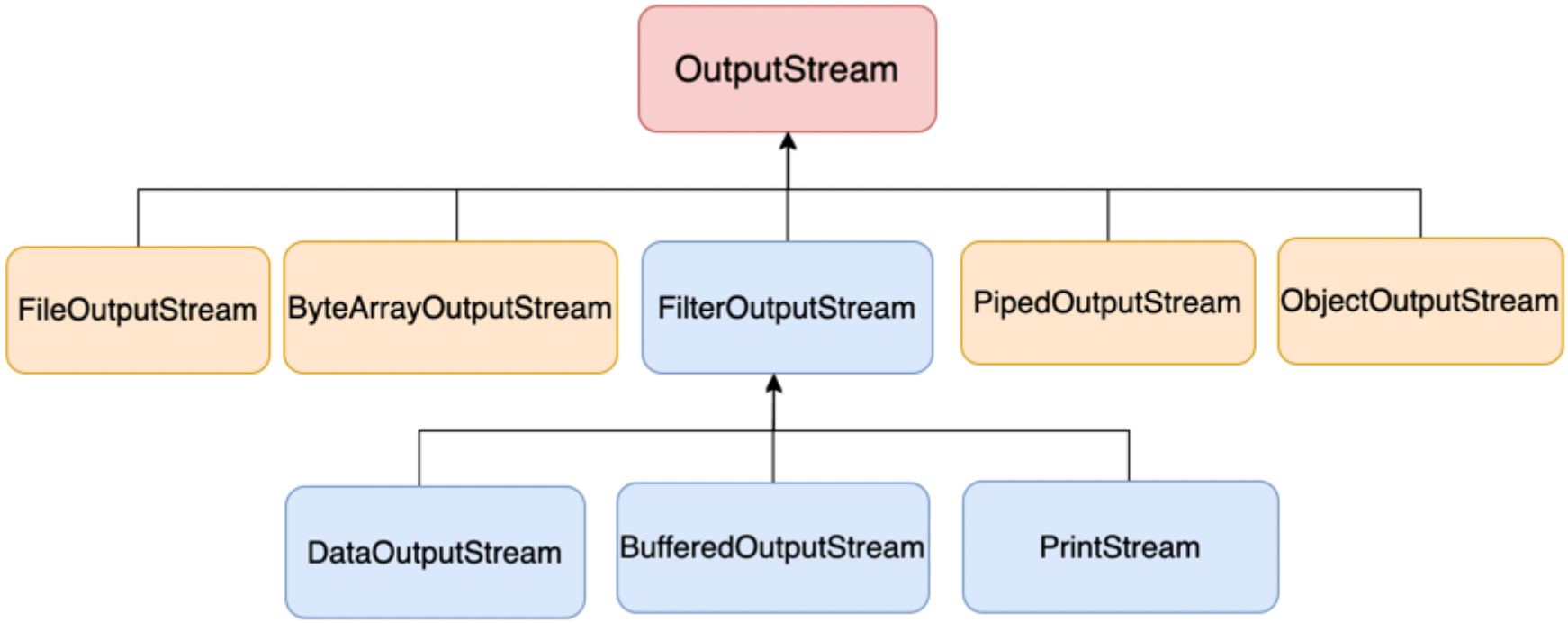
OutputStream class is an **abstract** class. Since java.io.OutputStream class is abstract class you have to use subclass of it in your application. It has subclasses for specified problems. For example; to write data to a file, use FileOutputStream, for in memory streaming use ByteArrayOutputStream, to buffer data use BufferedOutStream etc.

```
OutputStream fileStream=new FileOutputStream("demo.txt");

OutputStream arrayStream=new ByteArrayOutputStream();
```

```
OutputStream buffStream=new BufferedOutputStream(new FileOutputStream("dome.txt"));

...
```

In the picture below you will see the subclass of the OutputStream class.

FilterOutputStream is another subclass of OutputStream. It is also not used as alone for streaming. We are using subclass of FilterOutputStream in our operations, which are DataOutputStream, BufferedOutputStream and PrintStream.



Java Output Stream Classes

Useful methods of OutputStream

1. public void close() throws IOException

It closes the current output stream and releases any system resources associated with it. The closed stream cannot be reopened and operations cannot be performed within it.

2. public void flush() throws IOException

It flushes the current output stream and forces any buffered output to be written out.

3. Public void write(byte[ ] b) throws IOException

It writes the b.length bytes from the specified byte array to the output stream.

4. Public void write(byte[ ] b ,int off ,int len) throws IOException

It writes upto len bytes of data to the input stream. Here the “off” is the start offset in buffer array b, and the “len” represents the maximum number of bytes to be written in the output stream.

5. Public abstract void write(int b) throws IOException

It writes the specific byte to the output stream.

```
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.OutputStream;

OutputStream outputStream = new ByteArrayOutputStream();
try {
    outputStream.write("Input Values".getBytes());
```

```
    outputStream.close();
    System.out.println(outputStream);
} catch (IOException e) {
    throw new RuntimeException(e);
}
```

Close Stream

Streams and Readers / Writers class need to be closed properly when you are done using them. This is done by calling the close() method.

Since file handles are scarce, finite resources. You can run out of them if you don’t clean them up properly, just like database connections. So when you finished with File Stream you must close them properly.

```
InputStream input = null;

try{
  input = new FileInputStream("./input-text.txt");

  int data = input.read();
  while(data != -1) {
      //do something with data...
      doSomethingWithData(data);

      data = input.read();
  }
}catch(IOException e){
  //do something with e... log, perhaps rethrow etc.
} finally {
  try{
    if(input != null) input.close();
  } catch(IOException e){
    //do something, or ignore.
```

```
  }
}
```

try-with-resources

Java 7 introduced new statement to improve this implementation. It ensures that every resource will be close at the end of the statement.

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

  int data = input.read();
  while(data != -1) {
      //do something with data...
      doSomethingWithData(data);

      data = input.read();
  }
} catch(IOException e){
  //do something with e... log, perhaps rethrow etc.
}
```

IOException

```
 `IOException` is a general class for all I/O exceptions.
There are lots of subclasses of `IOException` in core java.
```

- `EOFException` for signaling that the reader reached the End Of the File.
- `SocketException` happens when there is a problem on creating or accessing Socket.

- `CharacterCodingException` When the problem is on Encoding or decoding.
- `FileNotFoundException` throws when there is a problem with opening the file does not exist, access problem)

File Input / Output Stream

Java FileOutputStream Class

Java FileOutputStream is an output stream used for writing data to a file.

If you have to write primitive values into a file, use FileOutputStream class. You can write byte-oriented as well as character-oriented data through FileOutputStream class. But, for character-oriented data, it is preferred to use FileWriter than FileOutputStream.

Example

```
   private static void writeDummyText() throws IOException {
        OutputStream fr=new FileOutputStream("demo.log");

        try(fr){
            fr.write(
                    """
                        I am dummy string, Just for test purpose.
                        You will use me for the next exercises.
                    """.getBytes()
                    );
            }
    }
```

Assignment

Create a program to writeDummyText exercise by using FileOutputStream class.

Java FileInputStream Class

Java FileInputStream class obtains input bytes from a file. It is used for reading byte-oriented data (streams of raw bytes) such as image data, audio, video etc. You can also read character-stream data. But, for reading streams of characters, it is recommended to use **FileReader** class.

```
import java.io.FileInputStream;
public class DataStreamExample {
     public static void main(String args[]){
          try{
              // be sure that testout.txt is exist
            InputStream fin=new FileInputStream("demo.log");
            int i=fin.read();
            System.out.print((char)i);

            fin.close();
          }catch(Exception e){System.out.println(e);}
         }
        }
```

use read(byte[]) while reading large file.

You have already known that read() method only read 1 byte in a time. To read 10K byte from the disk that mean you need to access 10K time to disk, and it costs time.

Using read(byte[]) will save you accessing disk times.

For example.

For 10k input file and with 1k byte array size we will access to disk 10 times to read the data. On each read we will read 1k data from the disk. And it will be almost 10 times faster than reading with `read()` single byte per time methods.

```
class InputStreamDemo{

private static void main(String[] args) {
        Instant start=Instant.now();

        byte[] bytes = new byte[10];

        try  ( InputStream inputStream=new FileInputStream("demo.log");){
            int numberOfBytes;
            do {
                numberOfBytes = inputStream.read(bytes);

                System.out.println(new String(Arrays.copyOf(bytes,numberOfBytes)));
            } while (inputStream.available()>0);
        }
        Instant end=    Instant.now();
        long elapsed= Duration.between(start,end).toMillis();

        System.out.printf("Elapsed time to read file is %d",elapsed);
    }
}
```

Assignment

Create a program to ready text file that you created on previous exercises. Use read() and read(byte) to measure performance.

HINT: To see the performance difference you may need to have a large file. So you can modify your writeDummyText method to write same lines in

10_000_000 times.

Buffered Input and Output Stream

Reading(Writing) **each byte** one at a time from(to) a disk is slow and it is not applicable for the large files. With BufferedStream you can buffer your stream data to decrease disk access.

BufferedInputStream

It is faster to read data from memory. Buffering data to the memory is for this goal. When we buffer data we get amount of data to the memory and make read operation from the memory.

IMPORTANT	By default, BufferedInputStream has `8192`(8k) byte buffer size. If you dont override the buffer size, it will try to buffer 8k byte on each disk access.

For Example

You have 12k byte in your input stream. With buffered input stream you will be buffering 8192 (8k) byte on first disk access on second access you will buffer 4696(4k) byte.

For this operation your application will access to input stream source 2 times.

NOTE	BufferedInputStream constructor except Input Stream. To use BufferedInputStream you must provide input stream within constructor.

```
BufferedInputStream bf= new BufferedInputStream(new FileInputStream("demo.log"));
```

Assignment

- 1- Modify your writeDummyText method to write 1M line of same text.
- 2- Use BufferedInputStream, FileInputStream to read same file and measure their read performance.

BufferedOutputStream

Java BufferedOutputStream class is used for buffering an output stream. It internally uses buffer to store data. It adds more efficiency than to write data directly into a stream. So, it makes the performance fast.

As its name suggests, BufferOutputStream has an internal buffer (byte[]) to which contents of individual small writes are first copied. They are written to the underlying OutputStream when buffer is full, or the stream is flushed, or the stream is closed. This can make a big difference if there is a (relatively large) fixed overhead for each write operation to the underlying OutputStream, as is the case for FileOutputStream (which must make an operating system call) and many compressed streams.

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

public static void main(String[] args)throws Exception{

     OutputStream fout=new FileOutputStream("demo.log");
     BufferedOutputStream bout=new BufferedOutputStream(fout);
     String s="Hey guys whats up. Now you are seeing how to use buffered output stream";

     byte b[]=s.getBytes();

     try(fout; bout){
         bout.write(b);
         bout.flush();
     }
```

```
      System.out.println("Now you might have understood");
  }
}
```

SequenceInputStream

A SequenceInputStream represents the logical concatenation of other input streams. It starts out with an ordered collection of input streams and reads from the first one until end of file is reached, whereupon it reads from the second one, and so on, until end of file is reached on the last of the contained input streams.

To create SequenceInputStream you must give the input stream in constructor either providing Enumeration of InputStream source

```
SequenceInputStream(Enumeration<? extends InputStream> e)
```

or giving input source as parameter to the constructor. But in this way constructor accept only 2 input sources.

```
SequenceInputStream(InputStream s1, InputStream s2)
```

```
      try (
              InputStream fin1 = new FileInputStream("file1.txt");
              InputStream fin2 = new FileInputStream("file2.txt");
              SequenceInputStream sequenceStream=new SequenceInputStream(fin1,fin2);

        ) {
            int j;
            while((j=sequenceStream.read())!=-1){
               // TODO: do your operation
            }
            System.out.println("Reading all file is success...");
        }catch (IOException exception){
            // handle exception here
```

```
        }
```

Byte Array Input Stream

The ByteArrayInputStream is composed of two words: ByteArray and InputStream. As the name suggests, it can be used to read byte array as input stream. Java ByteArrayInputStream class contains an internal buffer which is used to read byte array as stream. In this stream, the data is read from a byte array.

The buffer of ByteArrayInputStream automatically grows according to data

To create a ByteArrayInputStream you must provide byte array within constructor.

```
import java.io.ByteArrayInputStream;
import java.io.IOException;
import java.io.InputStream;

InputStream input = new ByteArrayInputStream("A LARGE INPUT ".getBytes());
```

Byte Array Output Stream

Java ByteArrayOutputStream class is used to write common data into multiple files. In this stream, the data is written into a byte array which can be written to multiple streams later.

The ByteArrayOutputStream holds a copy of data and forwards it to multiple streams.

The buffer of ByteArrayOutputStream automatically grows according to data.

By using `write` method, we first fill the byte array with the data. Then by using `writeTo(OutputStream out)` we are writing all the content

of ByteArray to the specified output stream.

Example

```
import java.io.*;
public class ByteArrayOutputStreamExample {
public static void main(String args[]){

       try (
                OutputStream fout1 = new FileOutputStream("file1.txt");
                OutputStream fout2 = new FileOutputStream("file2.txt");
                ByteArrayOutputStream  bout = new ByteArrayOutputStream()
        ) {
            bout.write("a character".getBytes());
            bout.write(" or a text".getBytes());
            bout.writeTo(fout1);
            bout.writeTo(fout2);
            bout.flush();
            // file1.txt and file2.txt will have same text inside as 'a character or text'
            System.out.println("Success...");
        }catch (IOException exception){
            // handle exception here
        }
     }
    }
```

IO Pipe

In java Pipe is used for streaming data between threads. The read and write operations are blocking so we need at least 2 separate threads for it.

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

```
import java.io.PipedInputStream;
import java.io.PipedOutputStream;

public class PipeDemo {

    public static void main(String[] args) throws IOException {

        final PipedOutputStream output = new PipedOutputStream();
        final PipedInputStream input = new PipedInputStream(output);
        Thread thread1 = new Thread(() -> {
            try {
                output.write("Hello world, pipe!".getBytes());
            } catch (IOException e) {
            }
        });

        Thread thread2 = new Thread(() -> {
            try {
                int data = input.read();
                while (data != -1) {
                    System.out.print((char) data);
                    data = input.read();
                }
            } catch (IOException e) {
            }
        });

        thread1.start();
        thread2.start();
    }
}
```

NOTE Since 1.5 java has BlockingQue. If the third part library is not forcing you to use IO Pipe, it is better to use BlockingQue to exchange

> data between thread.

PushbackInputStream

The PushbackInputStream is intended to be used when you parse data from an InputStream. Sometimes you need to read ahead a few bytes to see what is coming, before you can determine how to interpret the current byte. The PushbackInputStream allows you to do that. Actually it allows you to push the read bytes back into the stream. These bytes will then be read again the next time you call read().

```
import java.io.*;

public class PushbackStreamExamples {

    public static void main(String[] args) throws IOException {
        InputStream input = new ByteArrayInputStream("input value".getBytes());
        int limit = 10;
        try (PushbackInputStream pushbackInputStream = new PushbackInputStream(input, limit)) {
            // READ
            int data1 = pushbackInputStream.read();
            int data2 = pushbackInputStream.read();
            int data3 = pushbackInputStream.read();
            System.out.println((char) data1);
            System.out.println((char) data2);
            System.out.println((char) data3);
            // PUSH BACK
            pushbackInputStream.unread(data2);
            pushbackInputStream.unread(data3);
            // READ AGAIN
            System.out.println((char) pushbackInputStream.read());
            System.out.println((char) pushbackInputStream.read());
        }
    }
```

```
}
```

Output

```
i
n
p
p
n
```

Java File Class

The File class is an abstract representation of file and directory pathname. A pathname can be either absolute or relative.

The File class have several methods for working with directories and files; such as creating new directories or files, deleting and renaming directories or files, listing the contents of a directory etc.

Modifier and Type	Method	Description
boolean	createNewFile()	It atomically creates a new, empty file named by this abstract pathname if and only if a file with this name does not yet exist.
boolean	delete	It deletes the file
boolean	canWrite()	It tests whether the application can modify the file denoted by this abstract pathname.String[]

Modifier and Type	Method	Description
boolean	canRead()	It tests whether the application can read the file denoted by this abstract pathname.
boolean	isDirectory()	It tests whether the file denoted by this abstract pathname is a directory.
boolean	isFile()	It tests whether the file denoted by this abstract pathname is a normal file.
String	getName()	It returns the name of the file or directory denoted by this abstract pathname.
String	getParent()	It returns the pathname string of this abstract pathname’s parent, or null if this pathname does not name a parent directory.
boolean	mkdir()	It creates the directory named by this abstract pathname.
File[]	listFiles()	It returns an array of abstract pathnames denoting the files in the directory denoted by this abstract pathname
String[]	list()	It returns an array of strings naming the files and directories in the directory denoted by this abstract pathname

Modifier and Type	Method	Description
boolean	mkdirs()	It creates the directory named by this abstract pathname and also all the non-existent parent directories
…	…	…

Examples

```
import java.io.File;
import java.io.IOException;
class FileOperation{

public static void main(String[] args) throws IOException {
        File file=new File("my/dummy/dir/dummyFile.txt");

        if(file.getParentFile().mkdirs()){
            // directory wasnt exist
            System.out.println("directory was created");
        }

        if(file.createNewFile()){
            // file wasnt exist
            System.out.println("file "+file.getName()+" was created");
        }

         if( file.delete()){
           System.out.println("File was deleted");
       }

    }
```

```
}
```

WARNING | To delete a folder you must delete sub file and folder first.

Some Useful Tips

- Use `File.separator` instead of `'/'`. The file separator can change through the operating systems. On UNIX systems the value of this field is '/', on Microsoft Windows systems it is '\\'.
- Use `File.pathSeperator` instead of `';'`. The file separator can change through the operating systems. On UNIX systems the value of this field is ':', on Microsoft Windows systems it is ';'.

file.getAbsolutePath() vs file.getCanonicalPath()

A canonical path is an absolute unique path to the file. A file can have only one canonical path and many absolute paths

Example

C:\projects\sandbox\trunk\file.txt ¬ canonical Path C:\projects\sandbox\trunk\.\file.txt ¬ Absolute Path C:\projects\sandbox\trunk\test\..\.\file.txt ¬ Absolute Path C:\projects\sandbox\trunk\.\test\..\.\file.txt ¬ Absolute Path

Use File with IO operation

To provide stream source you can use File instead of giving file path as well.

```
File file=new File("my/demo/dir/demo.log");

// TODO: create file and directories if not exist
```

```
OutputStream fout = new FileOutputStream(file);

// check file is exist before reading from the file

InputStream fout = new FileInputStream(file);
```

Assignment

Create an application to delete a folder.

HINT: You can use recursive function to delete sub folder and files.

WARNING	Please create a dummy folder for test cases. You can accidentally delete your important folder.

Path

As name suggests `Path` is the particular location of an entity such as file or a directory in a file system so that one can search and access it at that particular location.

Technically in terms of Java, Path is an interface which is introduced in **Java NIO** file package during Java version 7,and it is the representation of location in particular file system. As path interface is in Java NIO package, so it gets its qualified name as java.nio.file.Path.

In order to get the instance of Path we can use static method of `java.nio.file.Paths` class get(). This method converts a path string to a Path object.

Java Path class gives us an alternative way to use Path operation rather than using File class' methods.

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

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

       Path path = Paths.get("my/dummy/./path/dummyFile.txt");

       System.out.println("relative path is " + path);
       System.out.println("absolute path is " + path.toAbsolutePath());
       System.out.println("canonical path is " + path.toRealPath().normalize());

       // equals with file methods of
       File file =new File("my/dummy/./path/dummyFile.txt");

       System.out.println("relative path is " + file.getPath());
       System.out.println("absolute path is " + file.getAbsolutePath());
       System.out.println("canonical path is " + file.getCanonicalPath());

       // output will be
       /*
         relative path is my/dummy/./path/dummyFile.txt
         absolute path is /Users/john/IODemo/my/dummy/./path/dummyFile.txt
         canonical path is /Users/john/IODemo/my/dummy/path/dummyFile.txt
       * */
   }
}
```

To create a new Path you can use

```
Path myDummyTextPath=  Paths.get("my","dummy", "path","dummyFile.txt");

//or

Path myDummyTextPath=  Paths.get("my/dummy", "path","dummyFile.txt");
```

`Path` to `File` and `File` to `Path` is always exist

```
Path myDummyTextPath = Paths.get("my/dummy/./path/dummyFile.txt");

File file = myDummyTextPath.toFile();

//or

File myDummyTextFile = new File("my/dummy/./path/dummyFile.txt");

Path path = myDummyTextFile.toPath();
```

Files

Java introduce `Files` class under `java.nio.file` package since **1.7**

Java Files class contains static methods that work on files and directories. This class is used for basic file operations like create, read, write, copy and delete the files or directories of the file system.

- It is an alternative way to do File operation.
- Path operation of File class given to the Path class
- create, delete, mkDir, etc operation given to the Files class.

```
// old way

File file=new File("myDummyText.log");
file.createNewFile();

// new way
```

```
Path file=Paths.get("myDummyText.log");
Files.createFile();
```

Why did Oracle decide to bring a new API, and which one should I use?

As we know, the java.io package was delivered with the first release of the Java JDK, allowing us to perform I/O actions. Since then, many developers have reported many drawbacks, missing functionality, and problems with some of its capabilities.

Error Handling

The most common problem is poor error handling. Many methods don’t tell us any details about the encountered problem or even throw exceptions at all.

Let’s assume we have a simple program that deletes a file:

```
File file = new File("baeldung/tutorial.txt");
boolean result = file.delete();
```

This code compiles and runs successfully without any error. Of course, we have a result flag containing a false value, but we don’t know the reason for this failure. The file might not exist, or the program may not have permission to delete it.

We can now rewrite the same functionality using the newer NIO2 API:

```
Path path = Paths.get("baeldung/tutorial.txt");
Files.delete(path);
```

Now, the compiler requires us to handle an IOException. Moreover, a thrown exception has details about its failure that will tell you, for example, if

the file does not exist.

Metadata Support

The File class in the java.io package has poor metadata support, which leads to problems across different platforms with I/O operations requiring meta-information about files.

The metadata may also include permissions, file owner, and security attributes. Due to this, the File class doesn’t support symbolic links at all, and the rename() method doesn’t work consistently across different platforms.

Summary

Since Java 7, developers can now choose between two APIs to work with files.

To provide developers need, Oracle decided to deliver the NIO package, which brings the same functionality with massive improvements.

java.io.File is now considered as legacy, and not recommended for new projects. However, currently there’s no plan to deprecate and remove it. == Java Reader and Writer

Java has a broad range of implementations in `java.io` package :

- **InputStream /OutputStream :** for byte based streaming
- **Reader/ Writer :** For character based streaming

Java **Writer** and **Reader** are abstract class for writing and reading character to/from streams.

Here in the table you can see the Byte stream classes and Character stream alternatives

	Byte Based Input	Byte Based Output	Character Based Input	Character Based Output
Basic	InputStream	OutputStream	Reader InputStreamReader	Writer OutputStreamWriter
Arrays	ByteArrayInputStream	ByteArrayOutputStream	CharArrayReader	CharArrayWriter
Files	FileInputStream, RandomAccessFile	FileOutputStream, RandomAccessFile	FileReader	FileWriter
Pipes	PipedInputStream	PipedOutputStream	PipedReader	PipedWriter
Buffering	BufferedInputStream	BufferedOutputStream	BufferedReader	BufferedWriter
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			

`FileReader` The FileReader is intended to read text, in other words. One character may correspond to one or more bytes depending on the character encoding scheme. This class uses default system Character Encoding so if you want to use different one you can go with `InputStreamReader`.

`PushbackReader` is for situations that you need to read some data ahead to see what is coming next! This class allows you to push data back and read it again! Difference with `PushbackInputStream` is datatype (Character and byte). You can set a limit for the size of data that can push back to the stream.

`LineNumberReader` is `BufferedReader` with keeping track of line number. It starts with 0 for the first line.

`PrintWriter` is uses for printing data in the formatted, instead of byte value. For example, in `HttpServletResponse` to write html string, we can use `getWriter()` method and write html directly in the response.

```
import java.io.*;


try( Writer writer = new StringWriter()) {
    try(PrintWriter printWriter = new PrintWriter(writer)){
        printWriter.println((int)123);
        printWriter.write("Hello");
        printWriter.write(" World ");

    }


System.out.println(writer.toString());
}
```

Assignment

Please modify writeDummyText exercise to use FileReader and FileWriter to read and write operation.