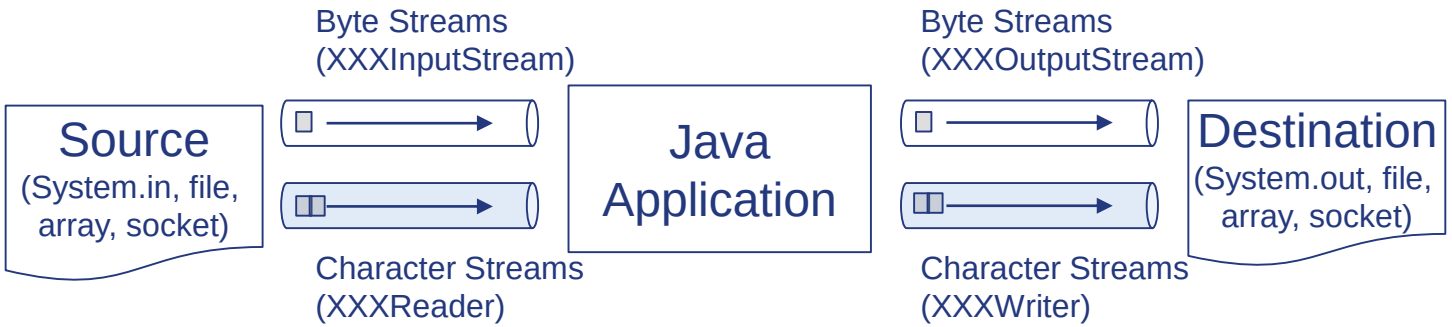
IO

- **IO Stream**
- **Byte Stream**
  - InputStream & OutputStream
- **Character Stream**
  - Reader & Writer
  - Standard IO
- **File Class**
- **NIO.2 (Paths, Files)**

자바의 정석: 입출력
Practical 모던 자바: 파일 IO
https://www.javatpoint.com/java-io
https://docs.oracle.com/javase/tutorial/essential/io/

Java IO Overview

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



Input	Output	Description
BufferedInputStream BufferedReader	BufferedOutputStream BufferedWriter	Buffered IO
FileInputStream FileReader	(they can create a file) FileOutputStream FileWriter	File IO
	PrintStream PrinterWriter	print, printf, println

IO Streams

- The java.io package contains many classes that your programs can use to read and write data.
  - **Byte Streams** handle I/O of raw binary data.
  - **Character Streams** handle I/O of character data, automatically handling translation to and from the local character set.
  - **Buffered Streams** optimize input and output by reducing the number of calls to the native API.
  - **Scanning and Formatting** allows a program to read and write formatted text.
  - **I/O from the Command Line** describes the Standard Streams and the Console object.
  - **Data Streams** handle binary I/O of primitive data type and String values.
  - **Object Streams** handle binary I/O of objects.
  -
- Most of the classes implement sequential access streams.
- The sequential access streams can be divided into two groups:
  - read and write bytes (1 byte)
  - read and write Unicode characters (2 bytes)
- Each sequential access stream has a speciality, such as reading from or writing to a file, filtering data as its read or written, or serializing an object.

Byte Stream

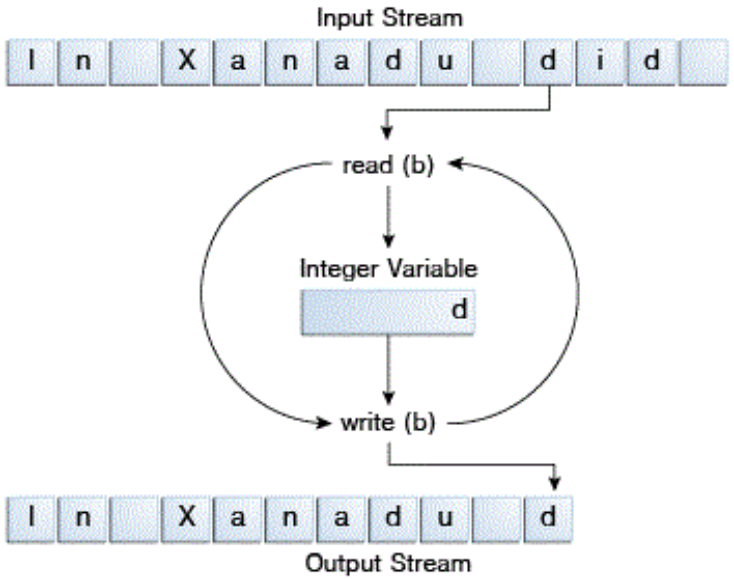
- Programs use byte streams to perform input and output of **1 bytes**.
- All byte stream classes are descended from **InputStream** and **OutputStream**.
- Closing a stream when it's no longer needed is very important

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

    FileInputStream in = null;
    FileOutputStream out = null;

    try {
      in = new FileInputStream("xanadu.txt");
      out = new FileOutputStream("outagain.txt");
      int c;

      while ((c = in.read()) != -1) {
        out.write(c);
      }
    } finally {
      if (in != null) {
        in.close();
      }
      if (out != null) {
        out.close();
      }
    }
  }
}
```



Character Stream

- The Java platform stores character values using **Unicode** conventions. (2 bytes)
- Character stream I/O automatically translates this internal format 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.
- The character stream uses the byte stream to perform the physical I/O, while the character stream handles translation between characters and bytes.

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

    BufferedReader inputStream = null;
    PrintWriter outputStream = null;

    try {
      inputStream = new BufferedReader(new FileReader("xanadu.txt"));
      outputStream = new PrintWriter(new FileWriter("characteroutput.txt"));

      String l;
      while ((l = inputStream.readLine()) != null) {
        outputStream.println(l);
      }
    } finally {
      if (inputStream != null) {
        inputStream.close();
      }
      if (outputStream != null) {
        outputStream.close();
      }
    }
  }
}
```

Input & Output Operations

- By providing common methods for IO, IO is possible in the same way even if source and destination are different.

InputStream	Reader	Description
abstract int read()	int read()	Reads a byte or a single character.
int read(byte[] b)	int read(char[] cbuf)	Reads bytes or characters into an array.
int read(byte[] b, int off, int len)	abstract int read(char[] cbuf, int off, int len)	Reads bytes or characters into a portion of an array.
int available()		Returns an estimate of the number of bytes that can be read from this input stream without blocking by the next invocation
void close()	abstract void close()	Closes this input stream and releases any system resources

OutStream	Writer	Description
abstract void write(int b)	void write(int c)	Writes a byte or a single character.
void write(byte[] b)	void write(char[] cbuf)	Writes bytes or characters into an array.
void write(byte[] b, int off, int len)	abstract void write(char[] cbuf, int off, int len)	Writes bytes or characters into a portion of an array.
void flush()	abstract void flush()	Flushes the stream
void close()	abstract void close()	Closes the stream, flushing it first.

BufferedInput/OutputStream Example

- Buffered input streams read data from a memory area known as a buffer; the native input API is called only when the buffer is empty.
- A program can convert an unbuffered stream into a buffered stream using the wrapping idiom we've used several times now, where the unbuffered stream object is passed to the constructor for a buffered stream class.
- It often makes sense to write out a buffer at critical points, without waiting for it to fill. This is known as flushing the buffer.
- **When closing the stream, the buffered stream flushes a buffer first**.

```
public class BufferedOutputStreamTest {
    public static void main(String[] args) throws IOException {
        FileOutputStream fos = null;
        BufferedOutputStream out = null;
        byte[] data = {1, 2, 3, 4, 5, 6, 7, 8, 9};

        FileInputStream fis = null;
        BufferedInputStream in = null;

        try {
            fos = new FileOutputStream ("test.txt");
            out = new BufferedOutputStream (fos, 4);  //wrapping
            for (int i=0; i< data.length; i++)
                out.write(data[i]);
            //out.close();                                    //flushing
            fis = new FileInputStream ("test.txt");
            in = new BufferedInputStream (fis);             //wrapping
            int b = 0;
            while((b = in.read())!=-1)
                System.out.print(b);
            //in.close();
        } finally {
            //if (out != null)  out.close();  // fos is automatically closed.
            //if (in != null)  in.close();  // fis is automatically closed.
        }
    }
}
```

12345678

PrintStream Example

- The Java platform supports three Standard Streams
  - Standard Input (System.in)
  - Standard Output (System.out)
  - Standard Error (System.err)
- These objects are defined automatically and do not need to be opened.
- You might expect the Standard Streams to be character streams, but, for historical reasons, they are byte streams.
  - **System.out and System.err are defined as PrintStream objects**
  - System.in is a byte stream with no character stream features. To use Standard Input as a character stream, wrap System.in in InputStreamReader.

```
public class StandardIOTest {
    public static void main(String[] args) {
        FileOutputStream fos = null;
        PrintStream ps = null;

        try {
            ps = new PrintStream (new FileOutputStream("standard.txt"));
            System.setOut(ps);
            System.out.println("Hello, World!");
            ps.close();
        } catch (FileNotFoundException e){
            System.err.println(e);
        } finally {
            if(ps!=null) ps.close();
        }
    }
}
```

standard.txt

```
Hello, World!
```

File Class

- An abstract representation of file and directory pathnames.
- This class presents an abstract, system-independent view of hierarchical pathnames:

```
C:\Users\me\Downloads\test.txt  ( MS Windows, seperator(₩) )
/home/me/Downloads/test.txt     ( Linux, seperator(/) )
```

- A pathname, whether abstract or in string form, may be either absolute or relative.

```
String currentPath = new File(".").getCanonicalPath();
        System.out.println("Current dir:" + currentPath);
String dataPath = new File(
      "data" + File.separator + "standard.txt").getCanonicalPath();
System.out.println("data dir: " + dataPath);
```

- Instances of this class may or may not denote an actual file-system object such as a file or a directory.

```
File f = new File ("C:\", "passwd"); // No Exception throws if the file does not exist
f.createNewFile();                   // creates a file
```

- Instances of the File class are immutable; that is, once created, the abstract pathname represented by a File object will never change.

File Example

- Displays all directories and file names in the user working directory.

```
public class FileListTest {
    public static void main(String[] args) {
        String currentDir = System.getProperty("user.dir");
        File f = new File(currentDir);

        if(!f.exists() || !f.isDirectory()) {
            System.out.println("Invalid directory!");
            System.exit(0);
        }

        File[] files = f.listFiles();

        for(int i=0; i < files.length; i++) {
            String fileName = files[i].getName();
            System.out.println(
                        files[i].isDirectory() ? "["+fileName+"]" : fileName);
        }
    }
}
```

Drawbacks in File Class

- **Many methods didn't throw exceptions** when they failed, so it was impossible to obtain a useful error message. For example, if a file deletion failed, the program would receive a "delete fail" but wouldn't know if it was because the file didn't exist, the user didn't have permissions, or there was some other problem.
- The ***rename*** method didn't work consistently across platforms.
- There was no real support for **symbolic links**.
- More support for **metadata** was desired, such as file permissions, file owner, and other security attributes.
- Accessing file metadata was inefficient.
- **Many of the *File* methods didn't scale**. Requesting a large directory listing over a server could result in a hang. Large directories could also cause memory resource problems, resulting in a denial of service.
- It was not possible to write reliable code that could recursively walk a file tree and respond appropriately **if there were circular symbolic links.**

NIO.2 (Java 7)

- The ***java.nio.file*** package and its related package, ***java.nio.file.attribute***, provide comprehensive support for file I/O and for accessing the default file system
- This package includes **support for symbolic links** where implementations provide these semantics.
- The ***File*** class defines the ***toPath ()*** method to construct a Path by converting the abstract path represented by the java.io.File object.
- The view of the files and file system provided by classes in this package are guaranteed to be consistent with other views provided by other instances in the same Java virtual machine.
- The SYNC and DSYNC options are used when opening a file to require that updates to the file are written synchronously to the underlying storage device.
- Passing a null argument to a constructor or method of any class or interface in this package will cause a ***NullPointerException*** to be thrown.
  - Invoking a method with a collection containing a null element will cause a ***NullPointerException***, unless otherwise specified.
  - Methods that attempt to access the file system will throw ***ClosedFileSystemException*** when invoked on objects associated with a FileSystem that has been closed

Package java.nio.file

- Defines interfaces and classes for the Java virtual machine **to access files, file attributes, and file systems**.

Interface	Description
Path	An object that may be used to locate a file in a file system.
PathMatcher	An interface that is implemented by objects that perform match operations on paths.
DirectoryStream<T>	An object to iterate over the entries in a directory.
WatchService	A watch service that watches registered objects for changes and events.
Watchable	An object that may be registered with a watch service so that it can be watched for changes and events.

Class	Description
Files	This class consists exclusively of static methods that operate on files, directories, or other types of files.
FileSystem	Provides an interface to a file system and is the factory for objects to access files and other objects in the file system.
FileSystems	Factory methods for file systems.
Paths	This class consists exclusively of static methods that return a Path by converting a path string or URI.

Mapping java.io.File Functionality to java.nio.file

- There is no one-to-one correspondence between the two APIs, but the following table gives you a general idea of what functionality in the java.io.File API maps to in the java.nio.file API

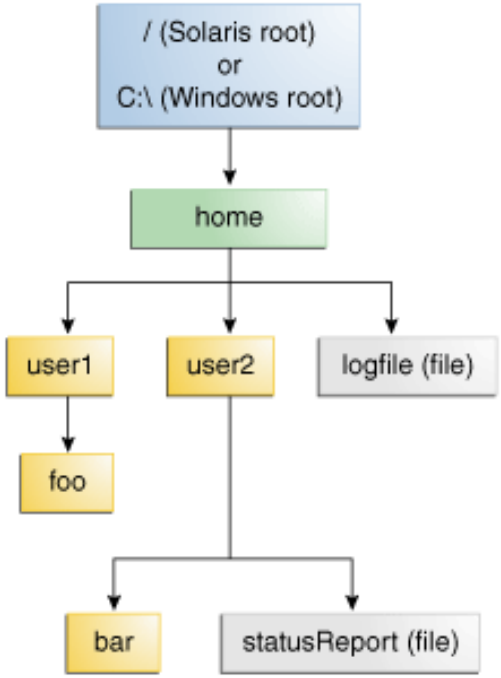
java.io.File	java.nio.file
java.io.File	java.nio.file.Path
File.getAbsolutePath	Path.toAbsolutePath
File.getCanonicalPath	Path.toRealPath or normalize
File.toURI	Path.toURI
File.list and listFiles	Path.newDirectoryStream
File.canRead, canWrite, canExecute	Files.isReadable, Files.isWritable, and Files.isExecutable.
File.isDirectory(), File.isFile(), and File.length()	Files.isDirectory(Path, LinkOption...), Files.isRegularFile(Path, LinkOption...), and Files.size(Path)
new File(parent, "newfile")	parent.resolve("newfile")
File.renameTo	Files.delete
File.createNewFile	Files.createFile
File.delete	Files.delete
File.exists	Files.exists and Files.notExists

Path Example

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

public class PathTest {
    public static void main(String[] args) {
        // None of these methods requires that the file corresponding to the Path exists.
        // Microsoft Windows syntax
        Path path = Paths.get("C:₩₩home₩₩joe₩₩foo");
        // Linux syntax
        //Path path = Paths.get("/home/joe/foo");

        System.out.format("toString: %s%n", path.toString());
        System.out.format("getFileName: %s%n", path.getFileName());
        System.out.format("getName(0): %s%n", path.getName(0));
        System.out.format("getNameCount: %d%n", path.getNameCount());
        System.out.format("subpath(0,2): %s%n", path.subpath(0,2));
        System.out.format("getParent: %s%n", path.getParent());
        System.out.format("getRoot: %s%n", path.getRoot());
        System.out.format("File exists: %s%n", Files.exists(path));
    }
}
```




```
toString: C:\home\joe\foo
getFileName: foo
getName(0): home
getNameCount: 3
subpath(0,2): home\joe
getParent: C:\home\joe
getRoot: C:\
File exists: false
```

Files Example (1/2)

- The Files class is the other primary entrypoint of the java.nio.file package.

```
public class FilesTest {
    public static void main(String[] args) {
        Path path = Paths.get("", "src", "main", "java",
                                    "FilesTest.java");
        if (Files.exists(path) && !Files.notExists(path))
            System.out.println("File is existed!");
        if (Files.isDirectory(path))
            System.out.println("It is a directory!");
        if (Files.isRegularFile(path))
            System.out.println("It is a file!");
        if (Files.isReadable(path))
            System.out.println("It is readable!");
        if (Files.isWritable(path))
            System.out.println("It is writable!");
    }
}
```

```
File is existed!
It is a file!
It is readable!
It is writable!
```

```
public class CopyFileTest {
    public static void main(String[] args) {
        Path src = Paths.get("", "src", "main", "java",
                                    "CopyFileTest.java");
        Path dst = Paths.get("",  "CopyFileTest.java");
        CopyOption[] options =
                {StandardCopyOption.REPLACE_EXISTING};
        if (!Files.isDirectory(src) && Files.exists(src)){
            try {
                Files.copy (src, dst, options);
            } catch (FileAlreadyExistsException e) {
                System.err.println(e);
            } catch (IOException e) {
                System.err.println(e);
            }
        }

    }
}
```

Files Example (2/2)

- Reading a File by Using Buffered Stream I/O

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.IOException;
import java.nio.charset.Charset;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;

public class BufferedFileNIOTest {
    public static void main(String[] args) {
        Path src = Paths.get("", "src", "main", "java", "BufferedFileNIOTest.java");
        Path dst = Paths.get("",  "BufferedFileNIOTest.java");
        Charset charset = Charset.forName("UTF-8");

        try(BufferedReader br = Files.newBufferedReader (src, charset);
            BufferedWriter bw = Files.newBufferedWriter (dst, charset);
        ){
            String line;
            while((line=br.readLine())!=null) {
                bw.write(line, 0, line.length());
                bw.newLine();
            }

        } catch (IOException e){
            System.err.println(e);
        }
    }
}
```

Q&A