Input and Output with Streams and Files

Jim Wilson

Mobile Solutions Developer & Architect

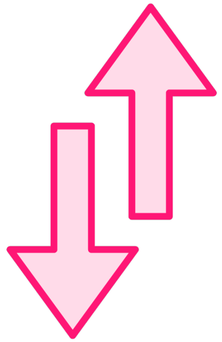
@hedgehogjim | jwhh.com

Overview

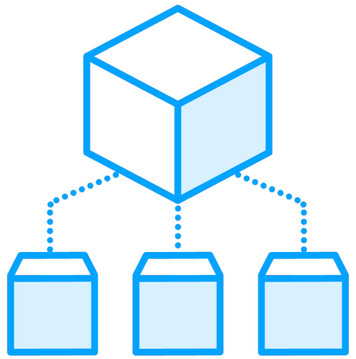
- Streams
- Stream errors and cleanup
- Chaining streams
- File and buffered streams
- Accessing files with java.nio.file package
- File systems
- Creating & working with zip file systems

Streams

Stream is an ordered sequence of data



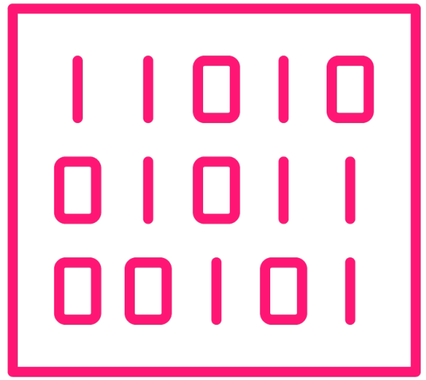
Provides a common I/O model



Abstracts details of underlying source/destination

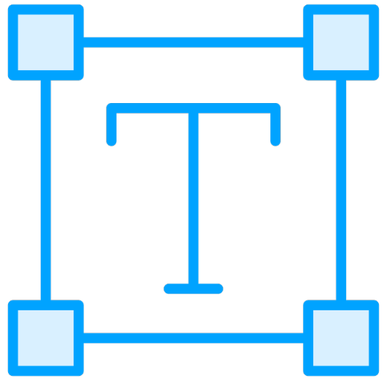
Stream types are unidirectional

2 categories of streams



Byte streams

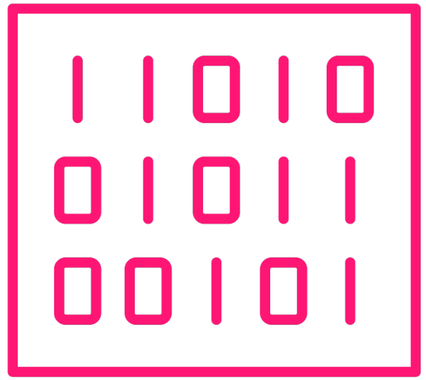
Interact as binary data



Text streams

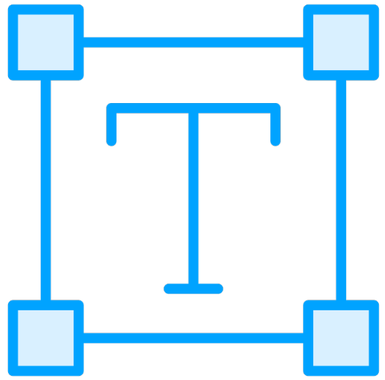
Interact as Unicode characters

Streams for Reading



InputStream

int read()



Reader

int read()

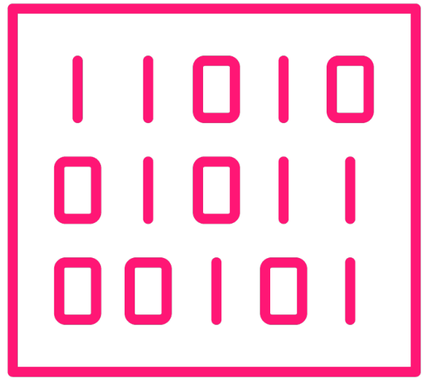
Reading One Byte at a Time

```
InputStream input = // create input stream
int intVal;
while ((intVal = input.read()) >= 0) { // returns -1 at end-of-stream
  byte byteVal = (byte) intVal;
  // do something with  byteVal
}
```

Reading One Character at a Time

```
Reader reader = // create reader
int intVal;
while ((intVal = reader.read()) >= 0) { // returns -1 at end-of-stream
  char charVal = (char) intVal;
  // do something with  charVal
}
```

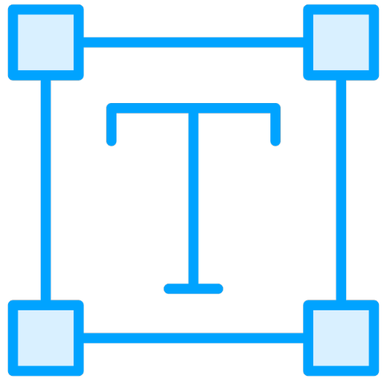
Streams for Reading



InputStream

int read()

int read(byte[] buff)



Reader

int read()

int read(char[] buff)

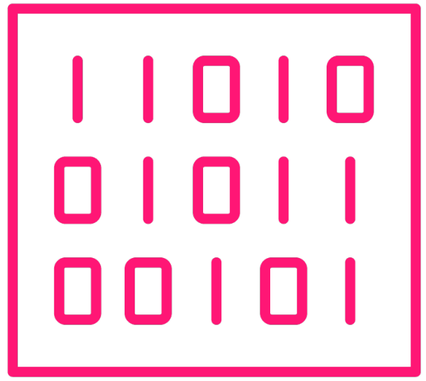
Reading Array of Bytes

```
InputStream input = // create input stream
int length;
byte[] byteBuff = new byte[10];
while ((length = input.read(byteBuff)) >= 0) { // returns -1 at end-of-stream
  for(int i=0; i < length; i++) {
    byte byteVal = byteBuff[i];
      // do something with byteVal
  }
}
```

Reading Array of Characters

```
Reader reader = // create reader
int length;
char[] charBuff = new char[10];
while ((length = reader.read(charBuff)) >= 0) { // returns -1 at end-of-stream
  for(int i=0; i < length; i++) {
    char charVal = charBuff[i];
    // do something with charVal
  }
}
```

Streams for Writing



OutputStream

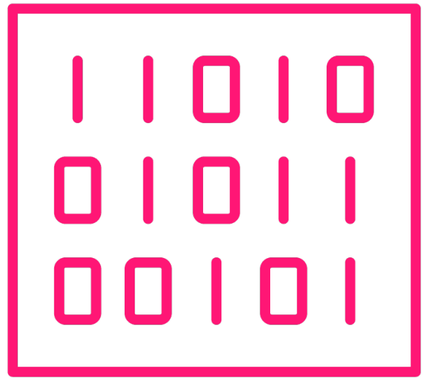
void write(int b)

void write(byte[] buff)

Writing Bytes

```
OutputStream output = // create output stream

byte byteVal = 100;
output.write(byteVal);

byte[] byteBuff = {0, 63, 127};
output.write(byteBuff);
```

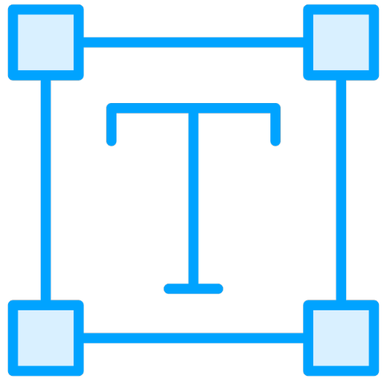
Streams for Writing



OutputStream

void write(int b)

void write(byte[] buff)



Writer

void write(int ch)

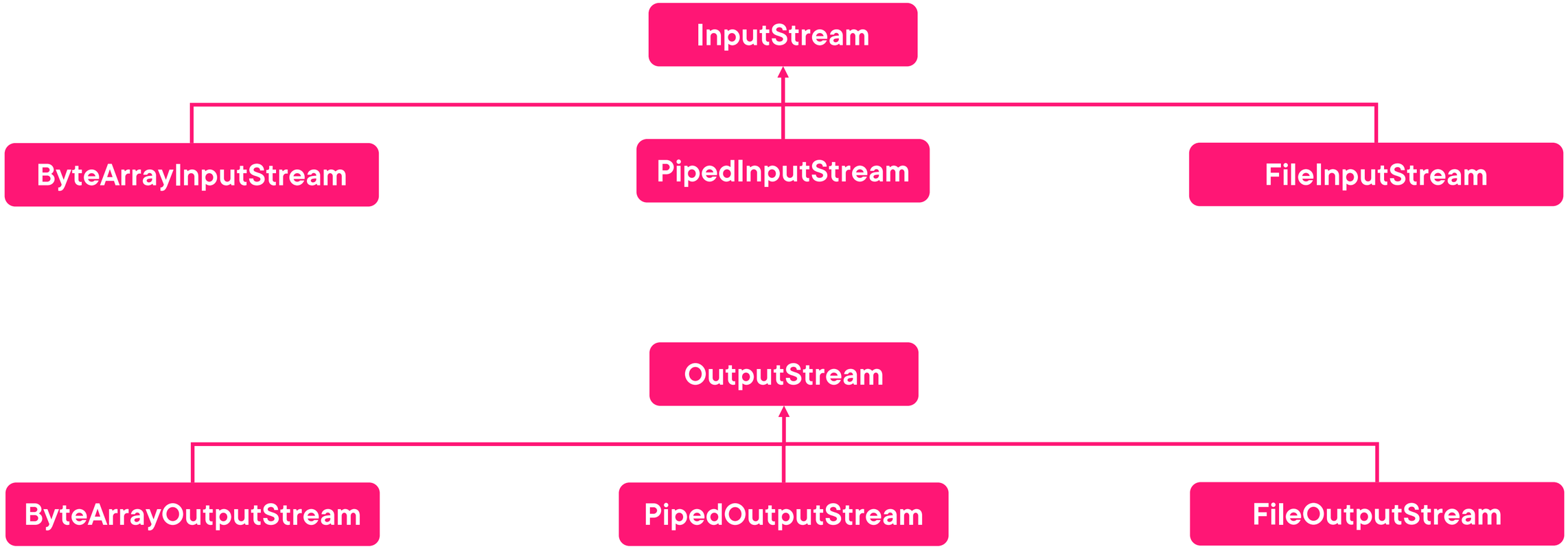
void write(char[] buff)

void write(String str)

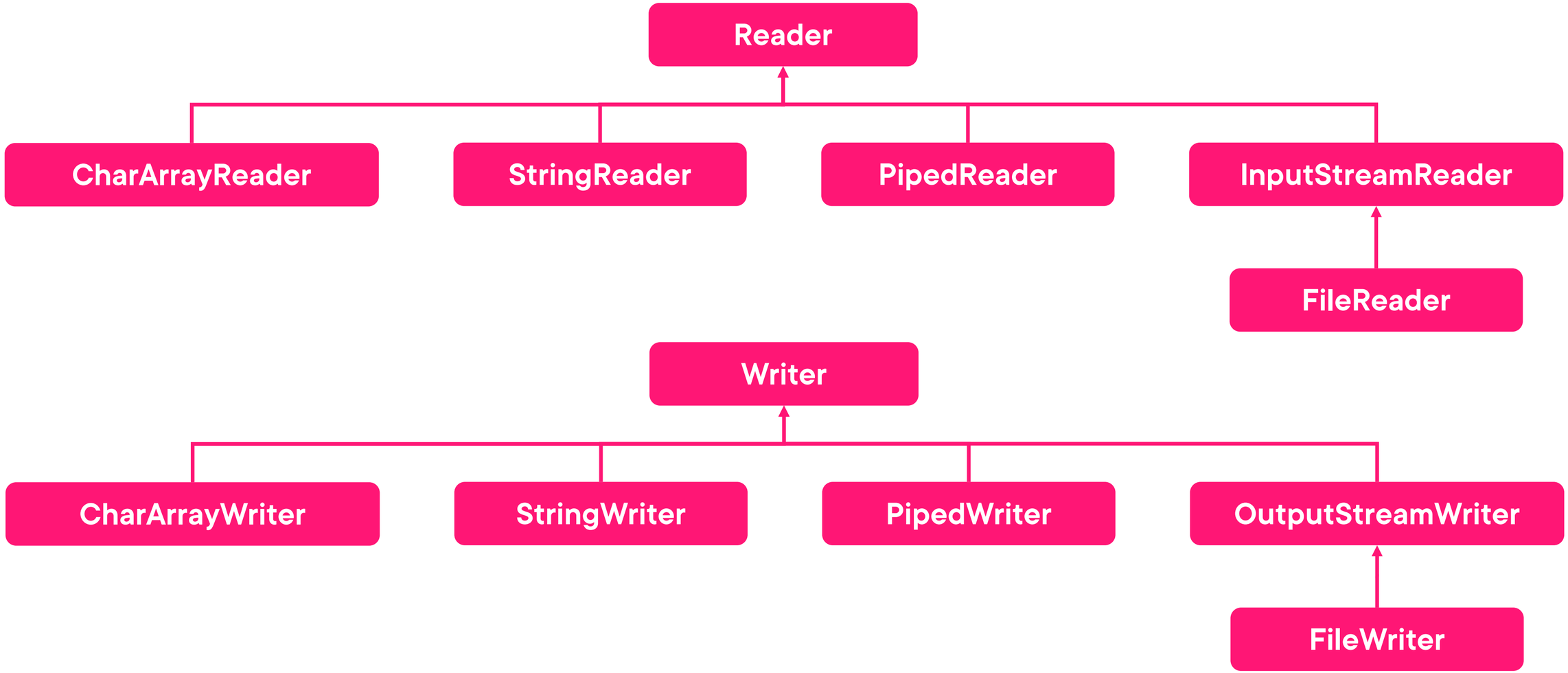
Writing Characters

```
Writer writer = // create writer

char charVal = ‘a’;
writer.write(charVal);

char[] charBuff = {‘a’, ‘b’, ‘c};
writer.write(charBuff);

String stringVal = “Hello World”;
writer.write(stringVal);
```

Common Input/OutputStream Derived Classes



Common Reader/Writer Derived Classes



Stream Realities

Error Handling

Stream methods throw exceptions to indicate errors

Cleanup

Cannot rely on standard Java resource recovery

Cleanup

Streams are backed by physical storage

- Often exist outside Java runtime
- Runtime may not reliably clean up

Providing reliable cleanup

- Streams implement Closeable interface
  - 1 method: close

Exception Handling and Closing

```
Reader reader;
try {
  reader = // open reader
  // do something with reader
} catch(IOException e) {
  // handle exception
} finally {
    reader.close();
}
```

```
try {
if(reader != null)
    reader.close();
} catch(IOException e2) {
  // handle exception
}
```

Automating Cleanup

AutoCloseable interface

- 1 method: close
- Base interface of Closeable interface
- Provides support for try-with-resources

```
interface AutoCloseable {
  void close() throws Exception;
}
```

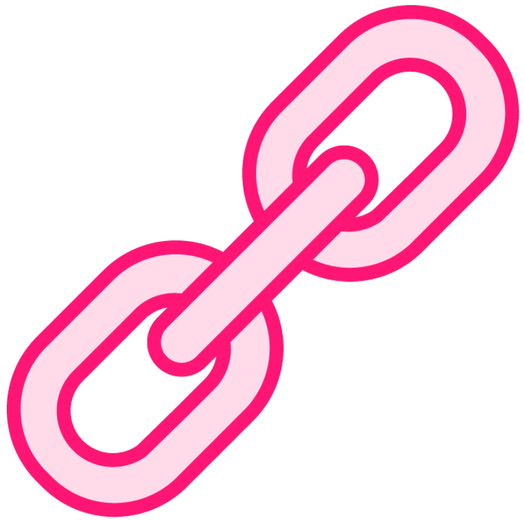
```
interface Closeable extends AutoCloseable {
  void close() throws IOException;
}
```

Automating Cleanup

Try-with-resources

- Automates cleanup of 1 or more resources
  - A “resource” is any type that implements AutoCloseable
- Syntax similar to traditional try statement
- Optionally includes catch block(s)
  - Handle try body
  - Handle close method call

Chaining Streams



Streams are often chained together

- One stream instance leverages another
- Creates higher-level functionality
- Simplifies reusability
- Chain using constructor

InputStreamReader leverages chaining

- Reader behavior over InputStream
- Character behavior over binary

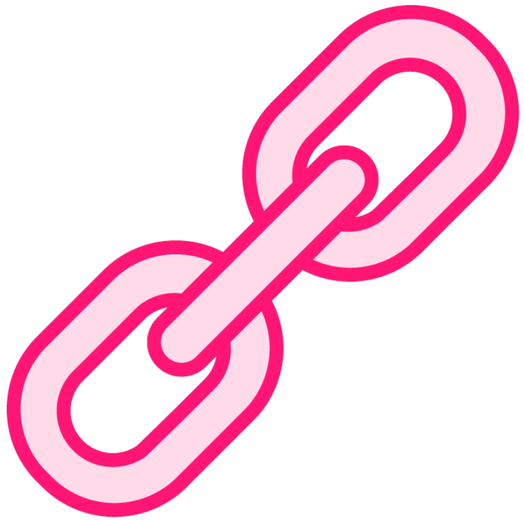
Chaining Streams

```
void doChain(InputStream in) throws IOException {
  int length;
  char[] charBuff = new char[128];
  try (InputStreamReader rdr = new InputStreamReader(in)) {
    while((length = rdr.read(charBuff)) >= 0) {
      // do something with charBuff
    }
  }
}
```

InputStreamReader then closes InputStream

Try-with-resources closes InputStreamReader

Chaining Streams



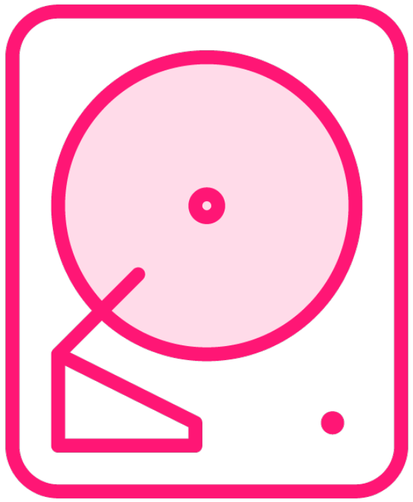
Can create your own “high-level” streams

- Most commonly chain similar streams
- Chain a reader over a reader, etc.

Classes available to simplify customization

- FilterReader, FilterWriter, FilterInputStream, FilterOutputStream
- Abstract classes
- Methods call to contained stream methods
- Override only customized methods

Accessing Files



Often use streams for file-based I/O

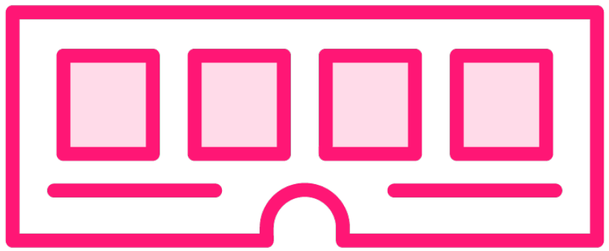
Class for each stream type in java.io package

- FileReader, FileWriter
- FileInputStream, FileOutputStream

The java.io classes are now deprecated

- Still widely used in code

Buffered Streams



Direct file access can be inefficient

Buffered streams can improve efficiency

- Buffers content in memory
- Performs reads/writes in large chunks
- Reduces underlying stream interaction

Buffering available for all 4 stream types

- BufferedReader, BufferedWriter
- BufferedInputStream, BufferedOutputStream

Buffered Streams

```
try (BufferedReader br =
                        new FileReader(“file1.txt”)
  int intVal;
  while((intVal = br.read()) >= 0) {
    char charVal = (char) intVal;
    // do something with charVal
  }
}
```

Buffered Streams and Line Breaks

Line breaks vary across platforms

- Unix: \n (new line)
- Windows: \r\n (carriage return & new line)

Buffered streams add line break support

- Use correct value for current platform

BufferedWriter

- Generate line breaks: newLine()

BufferedReader

- Line based read: readLine()

Writing with Line Breaks

```
void writeData(String[] data) throws IOException {



}
```

```
String[] data = {
  “Line 1”,
  “Line 2 2”,
  “Line 3 3 3”,
  “Line 4 4 4 4”,
  “Line 5 5 5 5 5”
}
```

Writing with Line Breaks

```
void writeData(String[] data) throws IOException {
  try (BufferedWriter bw =
       new BufferedWriter(new FileWriter("data.txt"))) {
    for(String d:data) {
      bw.write(d);
      bw.newline();
    }
  }
}
```

```
Line 1Line 2 2Line 3 3 3Line 4 4 4 4Line 5 5 5 5 5
```

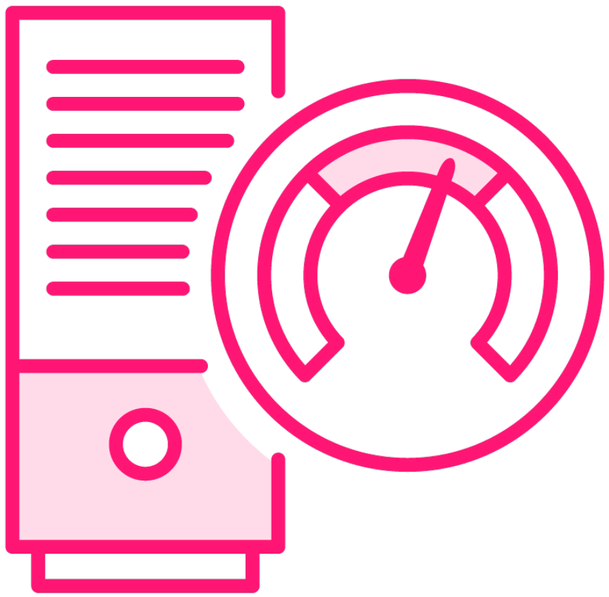
```
Line 1
Line 2 2
Line 3 3 3
Line 4 4 4 4
Line 5 5 5 5 5
```

Reading Lines

```
void readData() throws IOException {
  try (BufferedReader br =
       new BufferedReader(new FileReader(“data.txt”))) {
    String inValue;
    while((inValue = br.readLine()) != null) {
      System.out.println(inValue);
    }
  }
}
```

```
Line 1
Line 2 2
Line 3 3 3
Line 4 4 4 4
Line 5 5 5 5 5
```

Accessing Files with the java.nio.file Package



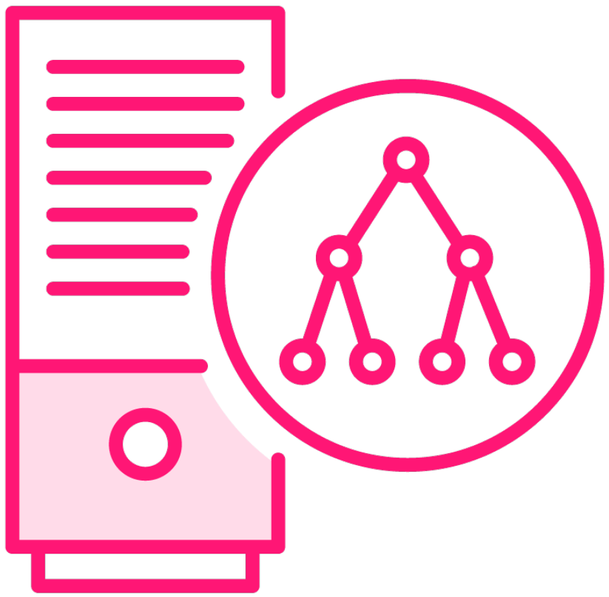
java.nio.file preferred package for files

- java.io FileXXX streams are deprecated

Provides a number of benefits over java.io

- Better exception reporting
- Greater scalability
- More file system feature support
- Simplifies common tasks

Path & Paths Type



Path

- Used to locate a file system item
- Can be a file or a directory

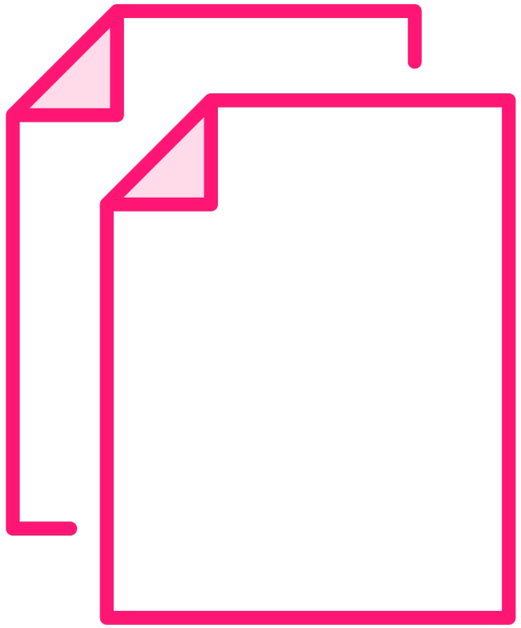
Paths

- Static Path factory methods
- From string-based hierarchical path
- From URI

```
Path p1 = Paths.get("\\documents\\data\\foo.txt");
```

```
Path p2 = Paths.get("\\documents", "data", "foo.txt");
```

Files Type



Files

- Static methods for interacting with files
- Create, copy, delete, etc.
- Open file streams
  - newBufferedReader, newBufferedWriter
  - newInputStream, newOutputStream
- Read/write file contents
  - readAllLines
  - write

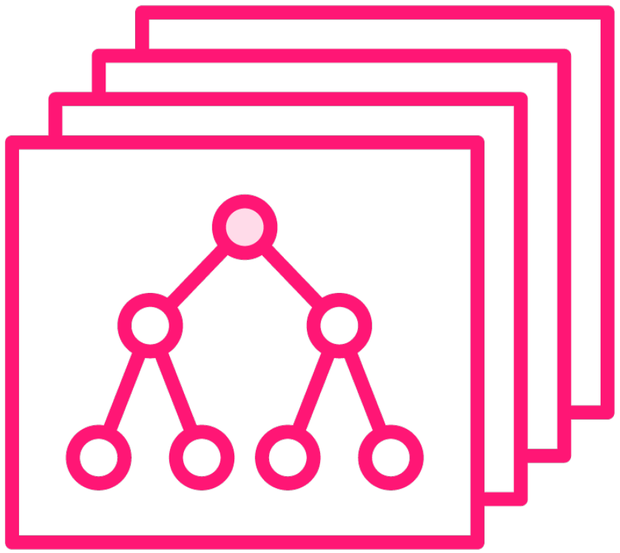
Reading Lines with BufferedReader

```
void readData() throws IOException {
  try (BufferedReader br =
       Files.newBufferedReader(Paths.get(“data.txt”))) {
    String inValue;
    while((inValue = br.readLine()) != null) {
      System.out.println(inValue);
    }
  }
}
```

Read All Lines

```
void readThemAll() throws IOException {
  List<String> lines =
    Files.readAllLines(Paths.get((“data.txt”));

  for(String line:lines)
    System.out.println(line);
}
```

File Systems



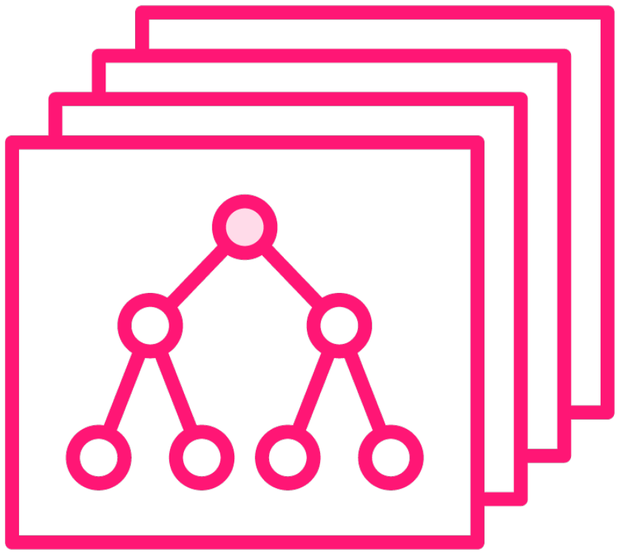
Files are contained within a file system

- Has a default file system
- Specialized file systems are supported
- Example: Zip file system

Path instances are tied to a file system

- Paths class works only for default

File System Types



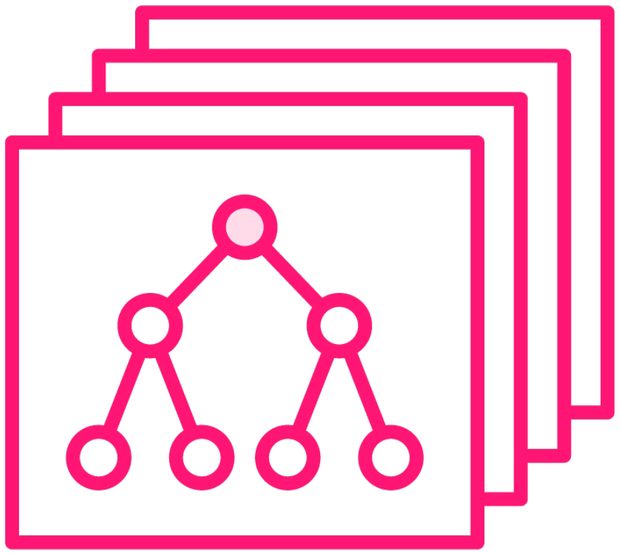
FileSystem

- Represents an individual file system
- Factory for Path instances

FileSystems

- Static FileSystem factory methods
- Open or create a file system
- Use newFileSystem method

Accessing File Systems



File systems identified by URIs

- Specifics of URI vary greatly
- Zip file system uses “jar:file” scheme
- jar:file:/jimwilson/data/bar.zip

File systems support custom properties

- Different for each file system type

Custom property examples

- Whether to create if doesn’t exist
- String encoding

Summary

java.io package

- Stream-based I/O
- Legacy file/filesystem types

java.nio.file package

- File/filesystem types

Summary

Streams

- Ordered sequence of data
- Unidirectional
- Binary or character based
- Can be chained together

Try-with-resources

- Automates resource cleanup
- Resources implement AutoCloseable

Summary

Path

- Locates a file system item
- Includes the file system

Paths

- Factory for Path instances for default file system

Files

- Methods for interacting with files

Summary

FileSystem

- Represents a file system
- Can have specialized such as zip
- File systems identified by URIs

FileSystems

- Methods to create/open file system