# CS1083 Week 6: Files

David Bremner

2018-02-06

# Outline

Streams

Data Input

Text Files

Error handling and files

Streams

Data Input

Text Files

Error handling and files

# Streams

- Abstraction of Files, input/output devices
  - Printer
  - Keyboard input
  - Terminal output
  - Files
- Sequence of bytes
- Different than graphics output, event driven input.

# Streams

- Abstraction of Files, input/output devices
  - Printer
  - Keyboard input
  - Terminal output
  - Files
- Sequence of bytes
- Different than graphics output, event driven input.

# Streams

- Abstraction of Files, input/output devices
  - Printer
  - Keyboard input
  - Terminal output
  - Files
- Sequence of bytes
- Different than graphics output, event driven input.

# Streams

- Abstraction of Files, input/output devices
  - Printer
  - Keyboard input
  - Terminal output
  - Files
- Sequence of bytes
  - read from front
  - add to end
- Different than graphics output, event driven input.

# Streams

- Abstraction of Files, input/output devices
  - Printer
  - Keyboard input
  - Terminal output
  - Files
- Sequence of bytes
  - read from front
  - add to end
- Different than graphics output, event driven input.

# Streams

- Abstraction of Files, input/output devices
  - Printer
  - Keyboard input
  - Terminal output
  - Files
- Sequence of bytes
  - read from front
  - add to end
- Different than graphics output, event driven input.

# Streams

- Abstraction of Files, input/output devices
  - Printer
  - Keyboard input
  - Terminal output
  - Files
- Sequence of bytes
  - read from front
  - add to end
- Different than graphics output, event driven input.

# Streams

- Abstraction of Files, input/output devices
  - Printer
  - Keyboard input
  - Terminal output
  - Files
- Sequence of bytes
  - read from front
  - add to end
- Different than graphics output, event driven input.

# Streams

- Abstraction of Files, input/output devices
  - Printer
  - Keyboard input
  - Terminal output
  - Files
- Sequence of bytes
  - read from front
  - add to end
- Different than graphics output, event driven input.

# Stream Example 1

```
public static void main(String [] args)
    throws IOException{
    byte outbyte=42;









} // Ignored exceptions?
```

ByteIO

# Stream Example 1

```java
public static void main(String [] args)
    throws IOException{
    byte outbyte=42;
    FileOutputStream out=
        new FileOutputStream("example.dat");
    out.write(outbyte);
    out.close();



} // Ignored exceptions?
```

ByteIO

# Stream Example 1

```java
public static void main(String [] args)
    throws IOException{
    byte outbyte=42;
    FileOutputStream out=
        new FileOutputStream("example.dat");
    out.write(outbyte);
    out.close();
    FileInputStream in=
        new FileInputStream("example.dat");
    System.out.println(in.read());
} // Ignored exceptions?
```

ByteIO

# Stream Example 2

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

}
```

FileCopy

# Stream Example 2

```
public static void main(String [] args)
    throws IOException{
    FileInputStream in=
        new FileInputStream(args[0]);
    FileOutputStream out=
        new FileOutputStream(args[1]);








}
```

FileCopy

# Stream Example 2

```
public static void main(String [] args)
    throws IOException{
    FileInputStream in=
        new FileInputStream(args[0]);
    FileOutputStream out=
        new FileOutputStream(args[1]);
    int b=in.read(); // int??
    while (b>=0){ // negative bytes??



    }
}
```

FileCopy

# Stream Example 2

```
public static void main(String [] args)
    throws IOException{
    FileInputStream in=
        new FileInputStream(args[0]);
    FileOutputStream out=
        new FileOutputStream(args[1]);
    int b=in.read(); // int??
    while (b>=0){ // negative bytes??
        System.out.println("Copying "+b);
        out.write(b);
        b=in.read();
    }
}
```

FileCopy

# Sample run

```
unix% java FileCopy mystery.dat foo.dat
Copying 0
Copying 74
Copying 0
Copying 65
Copying 0
Copying 86
Copying 0
Copying 65
unix%
```

# Buffering Input/Output

## Underlying problem

1. Disks are block devices: each physical read operation reads in many bytes.
2. Disks are very slow, relative to memory.

## Solution: Buffering

- Add a stage to the I/O pipeline that batches up reads or writes to the disk
- Reading and writing the same file makes this trickier: flush operation.

# Buffering Input/Output

## Underlying problem

1. Disks are block devices: each physical read operation reads in many bytes.
2. Disks are very slow, relative to memory.

## Solution: Buffering

- Add a stage to the I/O pipeline that batches up reads or writes to the disk
- Reading and writing the same file makes this trickier: flush operation.

# Buffered Input

Buffered Input Stream

read() // 1 byte

Input Stream

read(byteArray) // e.g. 512 bytes

Disk (OS)

# Performance improvement

## Without buffering:

```
[convex] java FileCopy2 bigFile.dat copy.dat
5120000 bytes copied in 7947ms
```

# Performance improvement

## Without buffering:

```
[convex] java FileCopy2 bigFile.dat copy.dat
5120000 bytes copied in 7947ms
```

## A small change

```
BufferedInputStream in= new BufferedInputStream(
                            new FileInputStream(args[0]));
BufferedOutputStream out= new BufferedOutputStream(
                            new FileOutputStream(args[1]));
```

# Performance improvement

## A small change

```
BufferedInputStream in= new BufferedInputStream(
                            new FileInputStream(args[0]));
BufferedOutputStream out= new BufferedOutputStream(
                            new FileOutputStream(args[1]));
```

## Big change

```
[convex] FileCopy3 bigFile.dat copy.dat
5120000 bytes copied in 212ms
```

Streams

Data Input

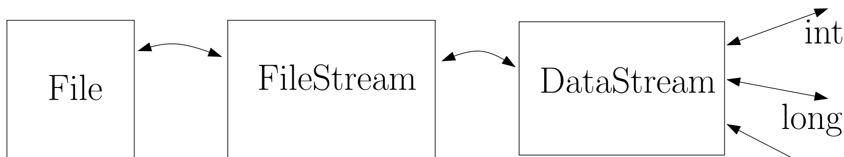
Text Files

Error handling and files

# I/O with Java primitive types

## Primitive Types

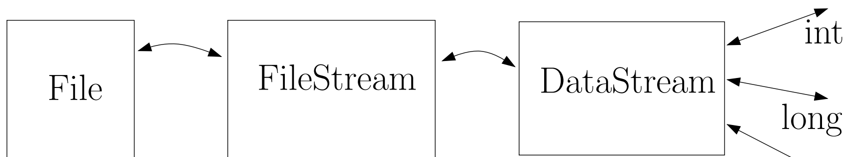
| type | size (bits) | size (bytes) |
|---|---|---|
| byte | 8 | 1 |
| char | 16 | 2 |
| int | 32 | 4 |
| long | 64 | 8 |



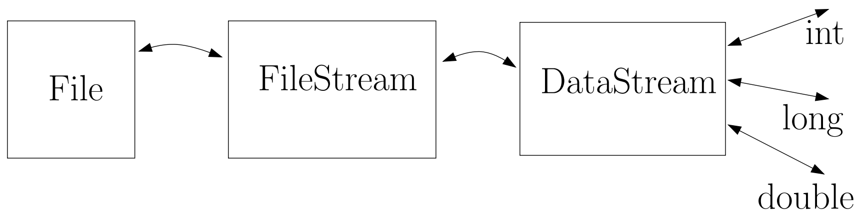
# I/O with Java primitive types

## Primitive Types

| type | size (bits) | size (bytes) |
|---|---|---|
| byte | 8 | 1 |
| char | 16 | 2 |
| int | 32 | 4 |
| long | 64 | 8 |



# I/O with Java primitive types




## Exceptions are somewhat unavoidable

- End of file indicated by exceptions for size $>1$ byte

# It's all in the interpretation

## Write some data

```
public static void writeIt() throws IOException{
  byte [] data={ 0, 74, 0, 65, 0, 86, 0, 65 };
  FileOutputStream out=
                  new FileOutputStream("mystery.dat");
  for (int i=0; i<data.length; i++)
  out.write(data[i]);
  out.close();
}
```

DataStreamTest

# Reading longs

## Read it back as one long

```
input=new DataInputStream(
                    new FileInputStream("mystery.dat"));
System.out.println(input.readLong());
```

Output: 20829427455098945

# Reading longs

## Read it back as one long

```
input=new DataInputStream(
                    new FileInputStream("mystery.dat"));
System.out.println(input.readLong());
```

## Output: 20829427455098945

# Reading shorts

## Read it back as four shorts

```
input=new DataInputStream(new FileInputStream("mystery.dat"));
for (int i=0; i<4; i++)
    System.out.print(input.readShort()+" ");
System.out.println("");
```

## Output: 74 65 86 65

- How are bytes turned into shorts?
- Where is this documented?
- What does "Java binary files are platform independent" mean?

# Reading shorts

## Read it back as four shorts

```
input=new DataInputStream(new FileInputStream("mystery.dat"));
for (int i=0; i<4; i++)
    System.out.print(input.readShort()+" ");
System.out.println("");
```

## Output: 74 65 86 65

- How are bytes turned into shorts?
- Where is this documented?
- What does "Java binary files are platform independent" mean?

# Reading shorts

## Read it back as four shorts

```
input=new DataInputStream(new FileInputStream("mystery.dat"));
for (int i=0; i<4; i++)
    System.out.print(input.readShort()+" ");
System.out.println("");
```

## Output: 74 65 86 65

- How are bytes turned into shorts?
- Where is this documented?
- What does “Java binary files are platform independent” mean?

# Reading shorts

## Read it back as four shorts

```
input=new DataInputStream(new FileInputStream("mystery.dat"));
for (int i=0; i<4; i++)
    System.out.print(input.readShort()+" ");
System.out.println("");
```

## Output: 74 65 86 65

- How are bytes turned into shorts?
- Where is this documented?
- What does “Java binary files are platform independent” mean?

# Reading shorts

## Read it back as four shorts

```
input=new DataInputStream(new FileInputStream("mystery.dat"));
for (int i=0; i<4; i++)
    System.out.print(input.readShort()+" ");
System.out.println("");
```

## Output: 74 65 86 65

- How are bytes turned into shorts?
- Where is this documented?
- What does "Java binary files are platform independent" mean?

# Reading doubles

## Read it as a double

```
input=new DataInputStream(new FileInputStream("mystery.dat"))
System.out.println(input.readDouble());
```

## Output:

2.892706362068199E-307

- Floating point is more complicated: IEEE standard.

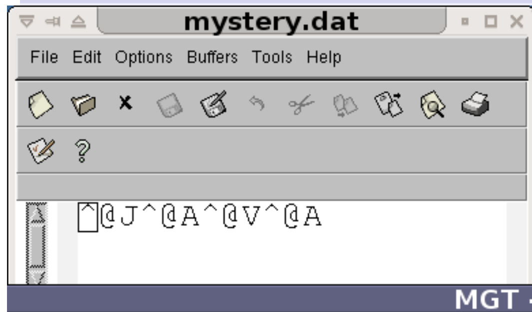
$$\underbrace{\text{mantissa}}_{53 \text{ bits}} | \underbrace{\text{exponent}}_{11 \text{ bits}} = 64 \text{ bits} = 8 \text{ bytes}$$

# Reading Chars

## read as 4 chars

```
for (int i=0; i<4; i++)
    System.out.print(input.readChar());
```

## Output: JAVA




- There seems to be some disagreement about what a character is!

Streams

Data Input

Text Files

Error handling and files

# Text files

- Sequence of characters
- Internal versus external representation
- Internal: char 16-bit unicode
- External: ASCII, UTF-8 (variable width).
- Logically organized into lines.

# Text files

- Sequence of characters
- Internal versus external representation
- Internal: char 16-bit unicode
- External: ASCII, UTF-8 (variable width).
- Logically organized into lines.

# Text files

- Sequence of characters
- Internal versus external representation
- Internal: char 16-bit unicode
- External: ASCII, UTF-8 (variable width).
- Logically organized into lines.

# Text files

- Sequence of characters
- Internal versus external representation
- Internal: char 16-bit unicode
- External: ASCII, UTF-8 (variable width).
- Logically organized into lines.

# Text files

- Sequence of characters
- Internal versus external representation
- Internal: char 16-bit unicode
- External: ASCII, UTF-8 (variable width).
- Logically organized into lines.

# Character Oriented I/O: rot13

```
public static void main(String [] args) throws IOException{
  InputStreamReader in=
      new InputStreamReader( new FileInputStream (args[0]));
  FileWriter out=new FileWriter(args[1]);
  int inChar=          ;
  while (            ){
    char outChar=Character.toUpperCase((char)inChar);
    if ('A' <= outChar && outChar <= 'Z')
      outChar = (char)(((int)outChar - 'A' + 13) % 26 +'A');
    if (Character.isLowerCase((char)inChar))
       outChar=Character.toLowerCase(outChar);

    inChar=in.read();
  }
               ;
}
```

Rot13

# Character Oriented I/O: rot13

```
public static void main(String [] args) throws IOException{
  InputStreamReader in=
      new InputStreamReader( new FileInputStream (args[0]));
  FileWriter out=new FileWriter(args[1]);
  int inChar= in.read() ;
  while (             ){
    char outChar=Character.toUpperCase((char)inChar);
    if ('A' <= outChar && outChar <= 'Z')
      outChar = (char)(((int)outChar - 'A' + 13) % 26 +'A');
    if (Character.isLowerCase((char)inChar))
       outChar=Character.toLowerCase(outChar);

    inChar=in.read();
  }
                 ;
}
```

Rot13

# Character Oriented I/O: rot13

```
public static void main(String [] args) throws IOException{
  InputStreamReader in=
      new InputStreamReader( new FileInputStream (args[0]));
  FileWriter out=new FileWriter(args[1]);
  int inChar= in.read() ;
  while ( inChar>=0 ){
    char outChar=Character.toUpperCase((char)inChar);
    if ('A' <= outChar && outChar <= 'Z')
      outChar = (char)(((int)outChar - 'A' + 13) % 26 +'A');
    if (Character.isLowerCase((char)inChar))
       outChar=Character.toLowerCase(outChar);

    inChar=in.read();
  }
              ;
}
```

Rot13

# Character Oriented I/O: rot13

```
public static void main(String [] args) throws IOException{
  InputStreamReader in=
      new InputStreamReader( new FileInputStream (args[0]));
  FileWriter out=new FileWriter(args[1]);
  int inChar= in.read() ;
  while ( inChar>=0 ){
    char outChar=Character.toUpperCase((char)inChar);
    if ('A' <= outChar && outChar <= 'Z')
      outChar = (char)(((int)outChar - 'A' + 13) % 26 +'A');
    if (Character.isLowerCase((char)inChar))
        outChar=Character.toLowerCase(outChar);
    out.write(outChar);
    inChar=in.read();
  }
                 ;
}
```

Rot13

# Character Oriented I/O: rot13

```java
public static void main(String [] args) throws IOException{
  InputStreamReader in=
      new InputStreamReader( new FileInputStream (args[0]));
  FileWriter out=new FileWriter(args[1]);
  int inChar= in.read() ;
  while ( inChar>=0 ){
    char outChar=Character.toUpperCase((char)inChar);
    if ('A' <= outChar && outChar <= 'Z')
      outChar = (char)(((int)outChar - 'A' + 13) % 26 +'A');
    if (Character.isLowerCase((char)inChar))
        outChar=Character.toLowerCase(outChar);
    out.write(outChar);
    inChar=in.read();
  }
  out.close() ;
}
```

Rot13

# Line Oriented Input

- Text streams are logically organized into lines, separated by the "new line character".
- Buffering is needed in Java to support line input.
- `readLine()` returns the next line, without the newline character.
- At the end of the file, `readLine` returns null

# Line Oriented Input

- Text streams are logically organized into lines, separated by the "new line character".
- Buffering is needed in Java to support line input.
- `readLine()` returns the next line, without the newline character.
- At the end of the file, `readLine` returns null

# Line Oriented Input

- Text streams are logically organized into lines, separated by the “new line character”.
- Buffering is needed in Java to support line input.
- `readLine()` returns the next line, without the newline character.
- At the end of the file, `readLine` returns null

# Line Oriented Input

- Text streams are logically organized into lines, separated by the "new line character".
- Buffering is needed in Java to support line input.
- `readLine()` returns the next line, without the newline character.
- At the end of the file, `readLine` returns null

# Numbering lines

NumberLines

```
public static void main(String [] args)throws IOException{
    BufferedReader in=
     new BufferedReader( new FileReader(args[0]) );
  int counter=1;
  String                    ;
  while (          ){
    System.out.print(counter);
    counter++;
    System.out.println("␣"+inLine);
                        ;
  }
}
```

# Numbering lines

NumberLines

```
public static void main(String [] args)throws IOException{
    BufferedReader in=
     new BufferedReader( new FileReader(args[0]) );
  int counter=1;
  String inLine=in.readLine() ;
  while (              ){
    System.out.print(counter);
    counter++;
    System.out.println("␣"+inLine);
                          ;
  }
}
```

# Numbering lines

NumberLines

```
public static void main(String [] args)throws IOException{
    BufferedReader in=
     new BufferedReader( new FileReader(args[0]) );
  int counter=1;
  String inLine=in.readLine() ;
  while ( inLine!=null ){
    System.out.print(counter);
    counter++;
    System.out.println(" "+inLine);
                        ;
  }
}
```

# Numbering lines

NumberLines

```
public static void main(String [] args)throws IOException{
    BufferedReader in=
     new BufferedReader( new FileReader(args[0]) );
  int counter=1;
  String inLine=in.readLine() ;
  while ( inLine!=null ){
    System.out.print(counter);
    counter++;
    System.out.println(" "+inLine);
     inLine=in.readLine() ;
  }
}
```

# Formatted Output: PrintWriter

- Readers/writers only deal with chars
- BufferedReaders read lines
- BufferedWriters output strings, but awkwardly.
- PrintWriters support methods print and println:

**Using PrintWriter**

NumberLines2

```
                    ⋮
PrintWriter out=new PrintWriter(new FileWriter(args[1]));
                    ⋮
      out.print(counter++);
      out.println(" "+inLine);
                    ⋮
```

# Formatted Output: PrintWriter

- Readers/writers only deal with chars
- BufferedReaders read lines
- BufferedWriters output strings, but awkwardly.
- PrintWriters support methods print and println:

## Using PrintWriter

NumberLines2

```
⋮
PrintWriter out=new PrintWriter(new FileWriter(args[1]));
⋮
    out.print(counter++);
    out.println(" "+inLine);
⋮
```

# Formatted Output: PrintWriter

- Readers/writers only deal with chars
- BufferedReaders read lines
- BufferedWriters output strings, but awkwardly.
- PrintWriters support methods print and println:

## Using PrintWriter

NumberLines2

```
⋮
PrintWriter out=new PrintWriter(new FileWriter(args[1]));
⋮
    out.print(counter++);
    out.println(" "+inLine);
⋮
```

# Formatted Output: PrintWriter

- Readers/writers only deal with chars
- BufferedReaders read lines
- BufferedWriters output strings, but awkwardly.
- PrintWriters support methods print and println:

## Using PrintWriter

NumberLines2

```
⋮
PrintWriter out=new PrintWriter(new FileWriter(args[1]));
⋮
    out.print(counter++);
    out.println(" "+inLine);
⋮
```

# Formatted Output: PrintWriter

- Readers/writers only deal with chars
- BufferedReaders read lines
- BufferedWriters output strings, but awkwardly.
- PrintWriters support methods print and println:

## Using PrintWriter

NumberLines2

```
                    ⋮
PrintWriter out=new PrintWriter(new FileWriter(args[1]));
                    ⋮
    out.print(counter++);
    out.println(" "+inLine);
                    ⋮
```

# When to use text, when to use binary

## Advantages of binary

- smaller, faster
- easier to read in data
- no ambiguity about data format.
- random access

## Advantages of text

- Human readable
- Less problems working with non-Java software.

For many applications, need for *random access* is key.

# When to use text, when to use binary

## Advantages of binary

- smaller, faster
- easier to read in data
- no ambiguity about data format.
- random access

## Advantages of text

- Human readable
  - Easier to debug
  - Can be reverse engineered
  - Can make test data with common tools
- Less problems working with non-Java software.

For many applications, need for *random access* is key.

# When to use text, when to use binary

## Advantages of binary

- smaller, faster
- easier to read in data
- no ambiguity about data format.
- random access

## Advantages of text

- Human readable
  - Easier to debug
  - Can be reverse engineered
  - Can make test data with common tools
- Less problems working with non-Java software.

For many applications, need for *random access* is key.

# When to use text, when to use binary

## Advantages of binary

- smaller, faster
- easier to read in data
- no ambiguity about data format.
- random access

## Advantages of text

- Human readable
  - Easier to debug
  - Can be reverse engineered
  - Can make test data with common tools
- Less problems working with non-Java software.

For many applications, need for *random access* is key.

# When to use text, when to use binary

## Advantages of binary

- smaller, faster
- easier to read in data
- no ambiguity about data format.
- random access

## Advantages of text

- Human readable
  - Easier to debug
  - Can be reverse engineered
  - Can make test data with common tools
- Less problems working with non-Java software.

For many applications, need for *random access* is key.

# When to use text, when to use binary

## Advantages of binary

- smaller, faster
- easier to read in data
- no ambiguity about data format.
- random access

## Advantages of text

- Human readable
  - Easier to debug
  - Can be reverse engineered
  - Can make test data with common tools
- Less problems working with non-Java software.

For many applications, need for *random access* is key.

# When to use text, when to use binary

## Advantages of binary

- smaller, faster
- easier to read in data
- no ambiguity about data format.
- random access

## Advantages of text

- Human readable
  - Easier to debug
  - Can be reverse engineered
  - Can make test data with common tools
- Less problems working with non-Java software.

For many applications, need for *random access* is key.

# When to use text, when to use binary

## Advantages of binary

- smaller, faster
- easier to read in data
- no ambiguity about data format.
- random access

## Advantages of text

- Human readable
  - Easier to debug
  - Can be reverse engineered
  - Can make test data with common tools
- Less problems working with non-Java software.

For many applications, need for *random access* is key.

# When to use text, when to use binary

## Advantages of binary

- smaller, faster
- easier to read in data
- no ambiguity about data format.
- random access

## Advantages of text

- Human readable
  - Easier to debug
  - Can be reverse engineered
  - Can make test data with common tools
- Less problems working with non-Java software.

For many applications, need for *random access* is key.

Streams

Data Input

Text Files

Error handling and files

# Error Handling and prevention I: who throws what

## FileInputStream(String)

- `FileNotFoundException` (checked)
- `SecurityException` (unchecked)

# Error Handling and prevention I: who throws what

## FileInputStream(String)

- `FileNotFoundException` (checked)
- `SecurityException` (unchecked)

## FileOutputStream(String)

- `FileNotFoundException` (checked)
  - What the heck? Check documentation. . .
- `SecurityException`

# Error Handling and prevention I: who throws what

## FileOutputStream(String)

- `FileNotFoundException` (checked)
  - What the heck? Check documentation...
- `SecurityException`

## read(byte)/write(byte)

IOException (checked, but unhandlable).

# A robust file copy I

```
BufferedInputStream in;
try{
    in= new BufferedInputStream(
                new FileInputStream(args[0]));
} catch (FileNotFoundException e){
    System.out.println("bad input: "+args[0]);
    System.exit(1);
}
```

FileCopy4

# A robust file copy I

```
BufferedOutputStream out;
try{
    out =new BufferedOutputStream(
                new FileOutputStream(args[1]));
} catch (FileNotFoundException e){
    System.out.println("bad output: "+args[1]);
    System.exit(1);
}
```

# Robust file copy II

FileCopy4

```
long bytes=0;
try {
    int b=in.read();
    while (b>=0){




    }
} catch (IOException e){
    System.out.println(e.getMessage());
    System.exit(1);
}
```

# Robust file copy II

FileCopy4

```
long bytes=0;
try {
    int b=in.read();
    while (b>=0){
        bytes++;
        out.write(b);
        b=in.read();
    }
} catch (IOException e){
    System.out.println(e.getMessage());
    System.exit(1);
}
```

# Class File: An ounce of prevention

## MetaData

- File permissions
- Directory listing
- Size
- File vs. Directory

# Class File: An ounce of prevention

## MetaData

- File permissions
- Directory listing
- Size
- File vs. Directory

# Class File: An ounce of prevention

## MetaData

- File permissions
- Directory listing
- Size
- File vs. Directory

# Class File: An ounce of prevention

## MetaData

- File permissions
- Directory listing
- Size
- File vs. Directory

# Check Before Opening

```
File inFile=new File(args[0]);
```

FileCopy5

# Check Before Opening

```
File inFile=new File(args[0]);
if (!inFile.canRead()){
    System.out.println("Could not open input "
                                    +args[0]);
    System.exit(1);
}
```

FileCopy5

# Check Before Opening

```
File inFile=new File(args[0]);
if (!inFile.canRead()){
    System.out.println("Could not open input "
                       +args[0]);
    System.exit(1);
}
File outFile=new File(args[1]);
if (!outFile.canWrite()){
    System.out.println("Could not open output "
                       +args[1]);
    System.exit(1);
}
```

FileCopy5

# Simplified exception handling

```
try {
    BufferedInputStream in=
        new BufferedInputStream(
            new FileInputStream(args[0]));
    BufferedOutputStream out =
         new BufferedOutputStream(
             new FileOutputStream(args[1]));
```

FileCopy5

# Simplified exception handling

```
try {
    BufferedOutputStream out =
        new BufferedOutputStream(
            new FileOutputStream(args[1]));
```

FileCopy5

# Simplified exception handling

```
try {
    int b=in.read();
    while (b>=0){
        bytes++;
        out.write(b);
        b=in.read();
    }
    out.close();
```

FileCopy5

# Simplified exception handling

```
try {
    int b=in.read();
    while (b>=0){
        bytes++;
        out.write(b);
        b=in.read();
    }
    out.close();
} catch (IOException e){
    System.out.println(e.getMessage());
    System.exit(1);
}
```

FileCopy5

# Running the example

```
unix% java FileCopy5 mystery.dat /usr/mystery.dat
Could not open output file /usr/mystery.dat
unix% java FileCopy5 mystery.dat /usr
Could not open output file /usr
unix% java FileCopy5 foo bar
Could not open input file foo
unix%
```