Chapter 8
Handling Input and Output

Overview

- General Input and Output
- Input and Output in Java
- Streams
- Scanners
- Input and Output from the Command-Line
- Input and Output from Files

General Input and Output

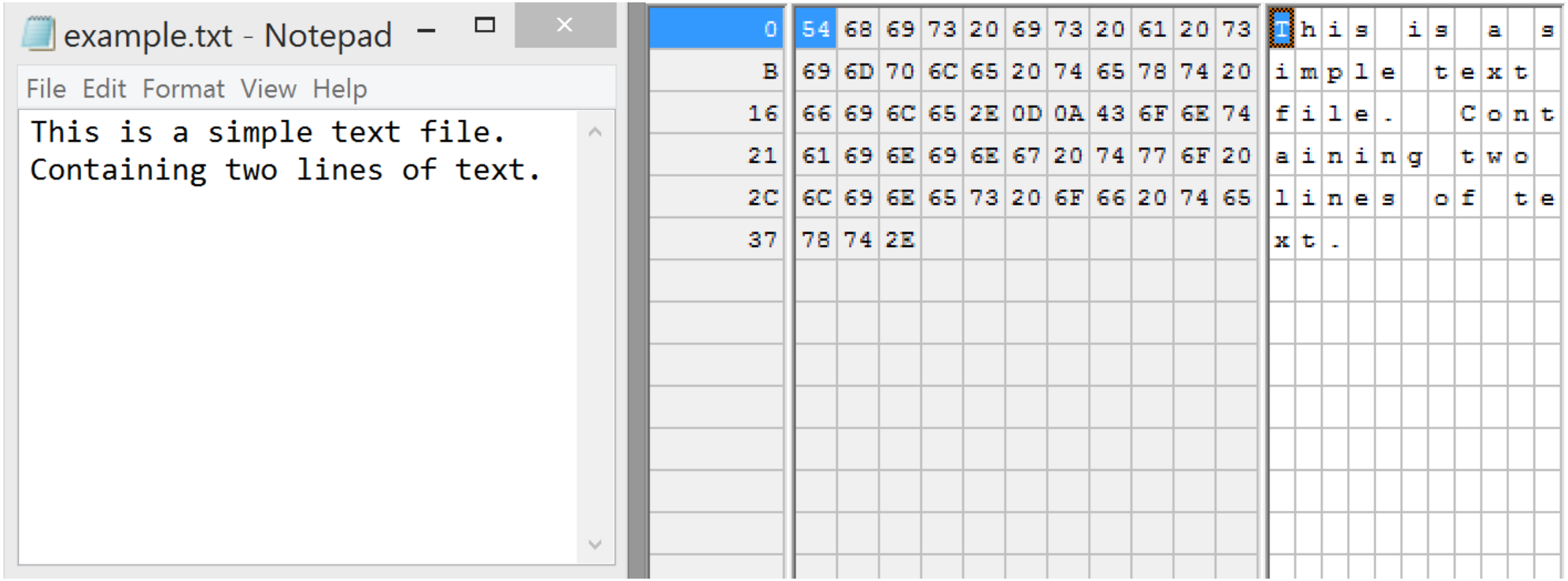
- Output to monitor, printer, file, and other programs
- Input from keyboard, mouse, file, other programs
- Focus: File I/O
- Important aspects
  - file modes
  - text versus binary files

General Input and Output

File Mode	Meaning	pointer
r	Open a file for reading only	beginning of file
r+	Open a file for reading and writing	beginning of file
w	Open a file for writing only	beginning of file
w+	Open a file for reading and writing	beginning of file
a	Open a file for writing only	end of file
a+	Open a file for reading and writing	end of file
x	Create a file and open for writing only; fail if file already exists	beginning of file
x+	Create a file and open for reading and writing; fail if file already exists	beginning of file
c	Open a file for writing only	beginning of file
c+	Open a file for reading and writing	beginning of file

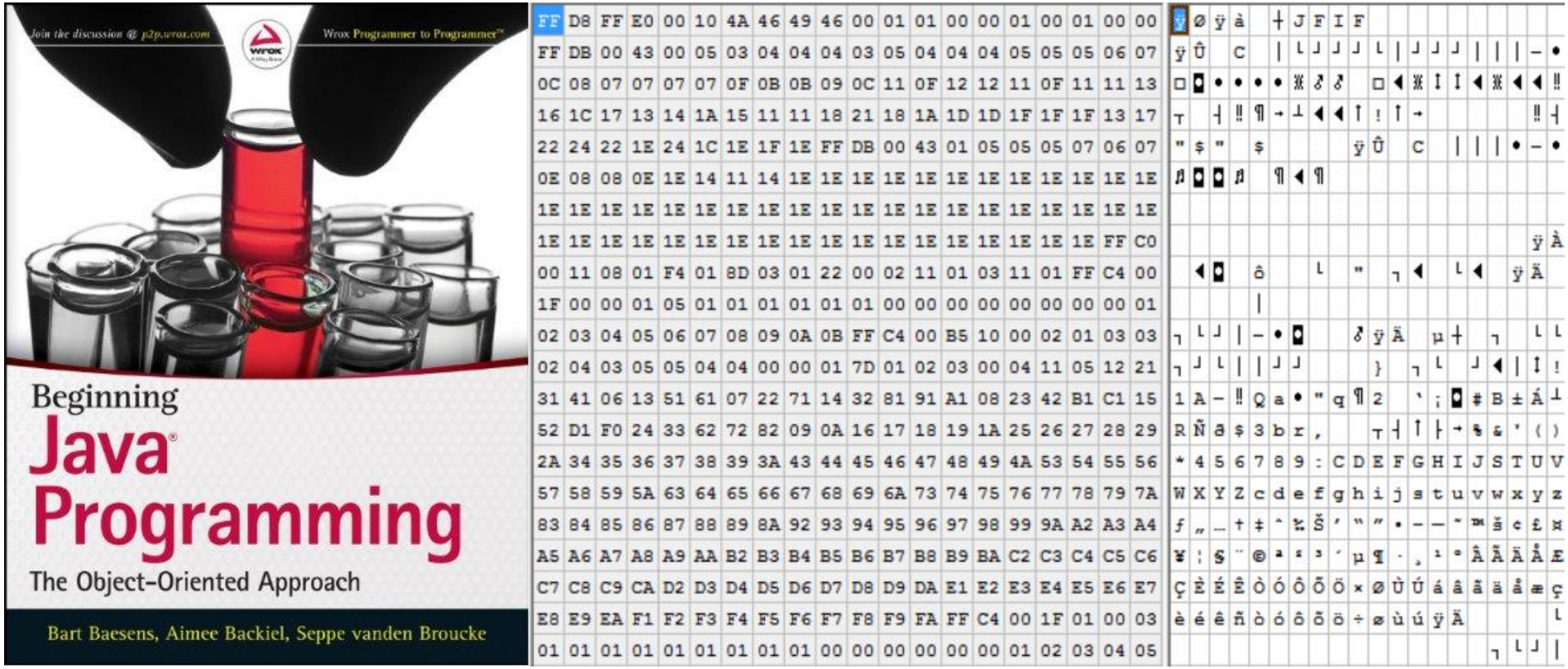
General Input and Output

- text file
  - bits represent characters (numbers, spaces, letters, etc.)



General Input and Output

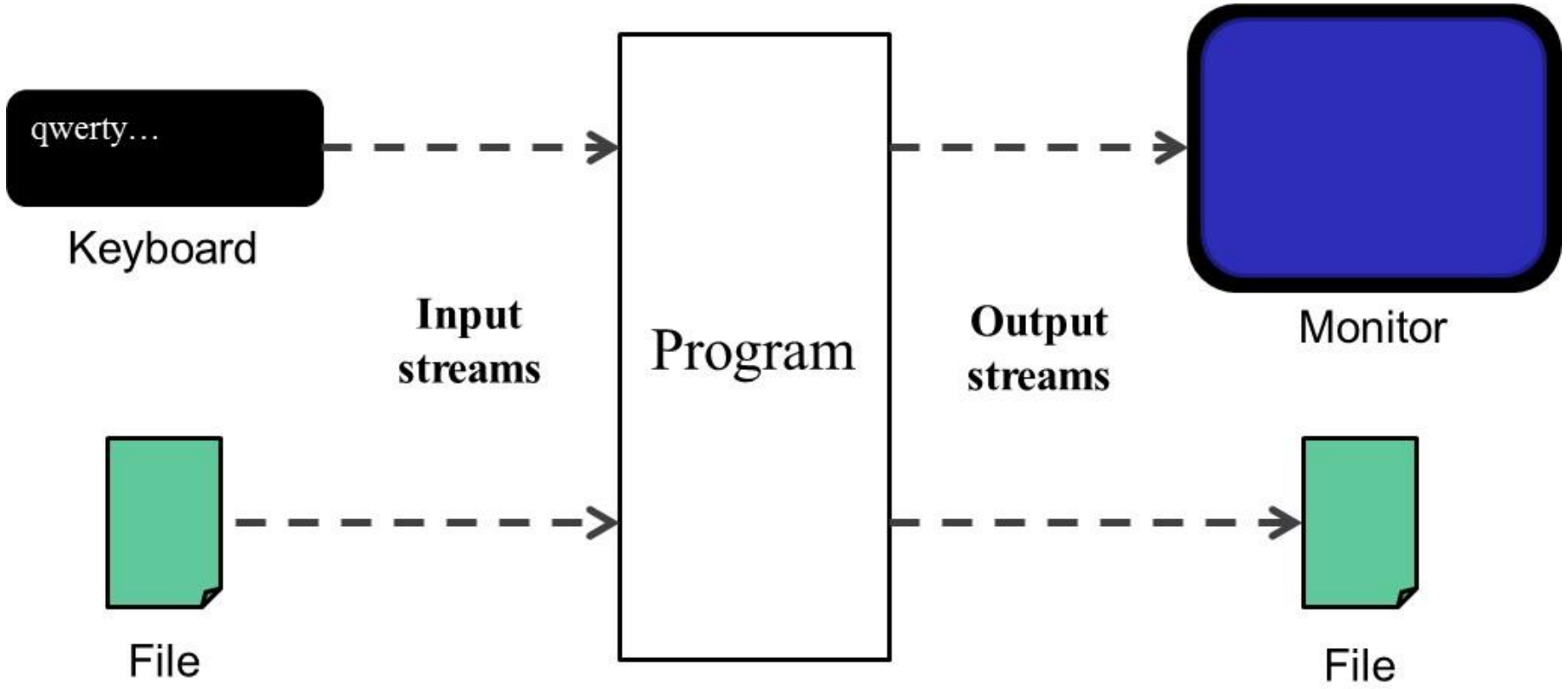
- binary file
  - bits represent custom data



Input and Output from Java

- Based on I/O streams
- A stream is an abstraction of a particular input source or output destination
  - Can represent files, program console, other programs, memory locations, or hardware devices
  - Can support various data formats such as raw bits and bytes, characters, primitive data types, or complete objects
  - Represent a sequence of data
  - Offer a unified model to deal with I/O

Input and Output from Java



Streams

- Byte Streams
- Character Streams
- Buffered Streams
- Data and Object Streams
- Other Streams

Byte Streams

- Sequence of data represented as bytes (eight bits)
- Lowest level of I/O
- Subclass `InputStream` and `OutputStream`

Byte Streams

- Key methods for `InputStream`

Method	Meaning
`int read()`	Reads the next byte of data from the input stream
`int read(byte[] b)`	Reads some number of bytes from the input stream and stores them into the array b
`int read(byte[] b, int off, int len)`	Reads up to `len` bytes of data from the input stream at offset off into an array b.
`long skip(long n)`	Skips over and discards n bytes of data from this input stream
`void close()`	Closes this input stream and releases any system resources associated with it
`int available()`	Returns an estimate of the number of bytes that can be read (or skipped over) from this input stream without blocking
`boolean markSupported()`	Tests if this input stream supports the mark and reset methods
`void mark(int readlimit)`	Marks the current position in this input stream
`void reset()`	Repositions this stream to the position at the time the mark method was last called on this input stream.

Byte streams

- Key methods for `OutputStream`

Method	Meaning
`void write(int b)`	Writes the specified byte (represented using an `int` variable) to this output stream
`void write(byte[] b)`	Writes the specified byte array `b` to this output stream
`void write(byte[] b, int off, int len)`	Writes `len` bytes from the `b` starting at offset `off` to this output stream
`void close()`	Closes this output stream and releases any system resources associated with this stream
`void flush()`	Flushes this output stream, i.e. forces any buffered output bytes to be written out

Byte Streams

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class FileCopier {

public static void main(String[] args) {
FileInputStream in = null;
FileOutputStream out = null;

try {
in = new FileInputStream("groceries.txt");
out = new FileOutputStream("groceries_copy.txt");
int c;
while ((c = in.read()) != -1) {
out.write(c);
System.out.print((char) c);
}
}
…
```

```
catch (IOException e) {
e.printStackTrace();
}

finally {
if (in != null) try { in.close(); }
 catch (IOException e)
{ e.printStackTrace(); }
if (out != null) try { out.close(); }
catch (IOException e)
{ e.printStackTrace(); }
}
}
}
```

Byte Streams

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class FileCopier {
public static void main(String[] args) {
try (
FileInputStream in = new FileInputStream("groceries.txt");
FileOutputStream out = new FileOutputStream("groceries_copy.txt");
) {
int c;
while ((c = in.read()) != -1) {
out.write(c);
System.out.print((char) c);
}
} catch
e.printStackTrace();
}}}
```

Byte Streams

- Key methods for `PrintStream`

Method	Meaning
`void print(String s)`	Writes the specified string to this print stream
`void println(String s)`	Writes the specified string to this print stream, but also terminates the line (i.e. starts a new line)
`void format(String format, Object... args)`	This method takes a "format string" as its first argument and the variables you want to format as the following arguments. The format string contains a number of percentage (%) fields representing where, which, and how variables should be formatted

Note: `System.out` is an example of a `PrintStream`

Byte Streams

```
public class FormattingOutput {

public static void main(String[] args) {

/* System.out is a PrintStream, which is a
subclass of OutputStream. */
System.out.write(50); // 50 corresponds to '2'
System.out.write((int)'\n'); // newline

// However, it is much easier to use print and println:
System.out.print("Text without newline");
System.out.print("\r\nYou can enter a newline\r\n“ + "manually, as
well as tabs using \t tab \t tab \t ...\r\n“ + “
Backslashes themselves are entered with \\...\r\n");
System.out.println("println is easier to show a “ + "string with a newline");

// The format method can be used to format arguments in a string
int number = 10;
double othernumber = 1.134;
System.out.println("Using + is okay in most cases: " + number + ", " + othernumber);
System.out.format("But format allows for more flexibility: %d, %3.2f %n", number, othernumber);
System.out.format("Another %3$s: %2$+020.10f, %1$d%n", number, othernumber, "example");
}}
```

Output:

```
2
Text without newline
You can enter a newline
manually, as well as tabs using tab tab ...
Backslashes themselves are entered with \...
println is easier to show a string with a newline
Using + is okay in most cases: 10, 1.134
But format allows for more flexibility: 10, 1.13
Another example: +00000001.1340000000, 10
```

Character Streams

- Translate Unicode characters (used within Java) to and from the characters locally specified
- Subclasses of `Reader` and `Writer`

Character Streams

```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class FileCopier {
public static void main(String[] args) {
try (
Reader in = new FileReader("groceries.txt");
Writer out = new FileWriter("groceries_copy.txt");
) {
int c;
while ((c = in.read()) != -1) {
out.write(c);
System.out.print((char) c);
}
} catch (IOException e) {
e.printStackTrace();
}}}
```

Buffered Streams

- Buffered streams are wrapped around other streams
- They provide a dedicated space in memory (a buffer) to store data in an efficient manner
  - will request time-expensive operations only if necessary (e.g. when buffer is full)
- Four buffer classes exist which can be wrapped around a byte or character input/output stream:
  - `BufferedInputStream`, `BufferedOutputStream`, `BufferedReader`, and `BufferedWriter`.

Buffered Streams

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class FileCopier {
public static void main(String[] args) {
try (
Reader in = new BufferedReader(
new FileReader("groceries.txt"));
Writer out = new BufferedWriter(
new FileWriter("groceries (copy).txt"));
) {
String line;
while ((line = in.readLine()) != null) {
out.write(line + System.lineSeparator());
System.out.println(line);
}
} catch (IOException e) {
e.printStackTrace();}}}
```

Data and Object Streams

- Data streams support binary input and output of primitive data type values (`boolean`, `char`, `byte`, `short`, `int`, `long`, `float`, and `double`) as well as String values
- Data streams implement either the `DataInput` interface or the `DataOutput` interface, i.e. `DataInputStream` and `DataOutputStream`
- Subclass `InputStream` and `OutputStream`

Data and Object Streams

- Object streams are similar to data streams, but allow the serialization of all objects that implement the `Serializable` marker interface
- Object streams implement either the `ObjectInput` or `ObjectOutput` interfaces (which themselves are subinterfaces of `DataInput` and `DataOutput`), i.e. `ObjectInputStream` and `ObjectOutputStream`.
- Subclass `InputStream` and `OutputStream`

Data and Object Streams

```
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;
import java.util.ArrayList;
import java.util.List;

public class ObjectOutputStreamTest {
public static void main(String[] args) {
int number1 = 5;
double number2 = 10.3;
String string = "a string";
List<String> list = new ArrayList<>();
list.add("a");
list.add("b");
try (
ObjectOutputStream out = new ObjectOutputStream(
new FileOutputStream("saved.txt"));
) {
out.writeInt(number1);
out.writeDouble(number2);
out.writeBytes(string);
out.writeObject(list);
} catch (IOException e) {
e.printStackTrace();}}}
```

Output:

@$™™™™™ša string.í sr FileCopier$1íéJ;1–
\----------
xr java.util.ArrayListxÒ&traed;Ça•----------
I ----------
sizexpw----------
t at bx.í w @$™™™™™ša stringsr FileCopier$1íéJ;1
\----------
xr java.util.ArrayListxò™.a---------- I
----
sizexpw----------
t at bx

Other Streams

- Examples:
  - `AudioInputStream`: reads in audio-based data
  - `ZipOutputStream`: implements an output stream for writing `ZIP` (compressed) files.
- Most subclass `InputStream` and `OutputStream`

Scanners

- Allows to break down input into various fragments (tokens) and map them according to their data type
- Implemented as `java.util.Scanner`

Groceriesswithprices.txt

```
apples, 5.33
bananas, 4.61
water, 1.00
orange juice, 2.50
milk, 3.20
bread, 1.11
```

Scanners

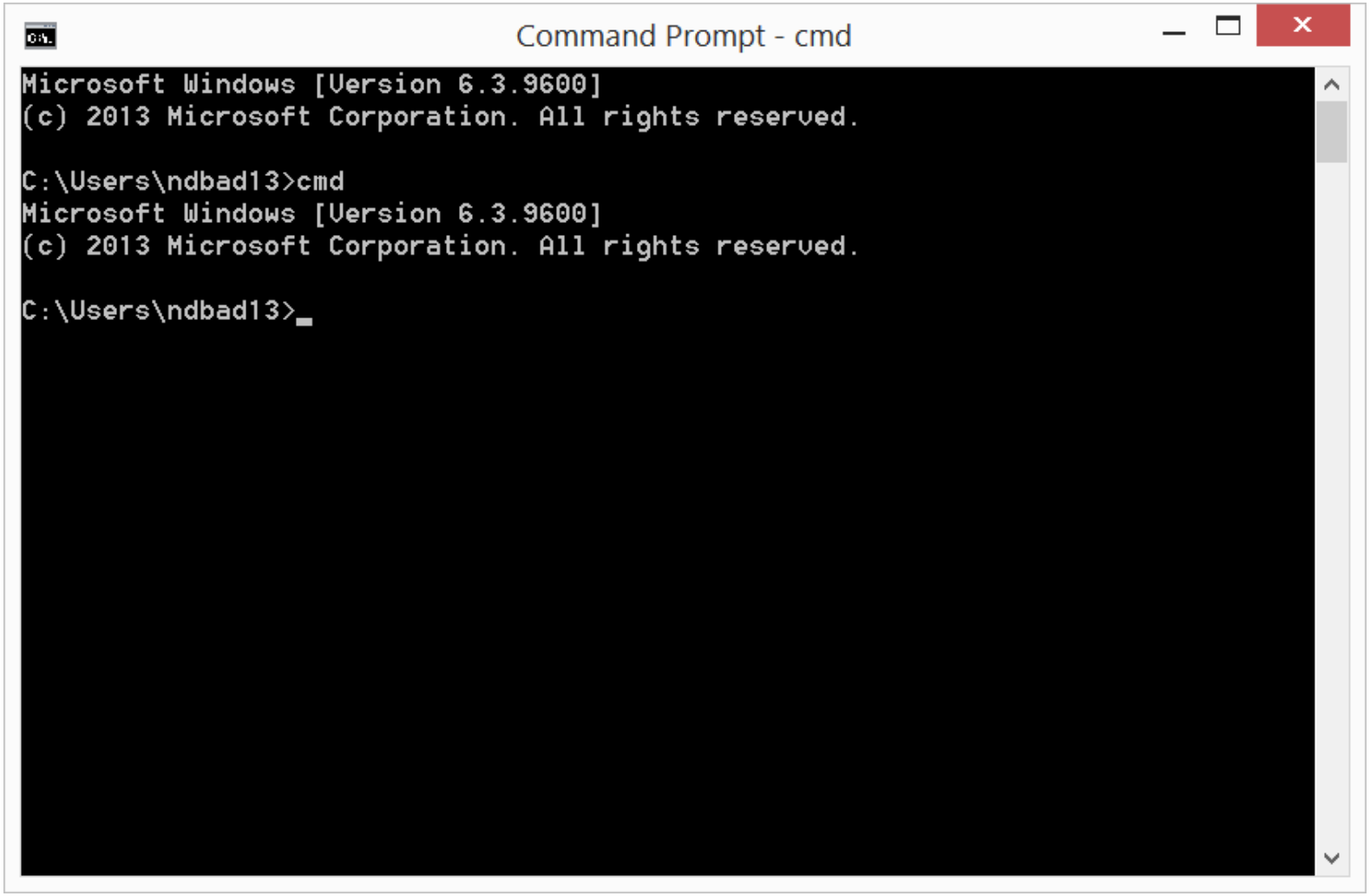
```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class ShowGroceries {
public static void main(String[] args) {
try (
BufferedReader in = new BufferedReader(new FileReader(
"grocerieswithprices.txt"));
) {
String line;
while ((line = in.readLine()) != null) {
String[] splittedLine = line.split(", ");
String item = splittedLine[0].trim();
double price = Double.parseDouble(splittedLine[1].trim());
System.out.format("Price of %s is: %.2f%n", item, price);
}
} catch (IOException e) {
e.printStackTrace();
}}}
```

Scanners

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.Locale;
import java.util.Scanner;
import java.util.regex.Pattern;

public class ShowGroceries {
public static void main(String[] args) {
try (
Scanner sc = new Scanner(
new FileReader("groceriesswithprices.txt"));
) {
sc.useDelimiter(Pattern.compile("(, )|(\r\n)"));
sc.useLocale(Locale.ENGLISH);

…
```

```
while (sc.hasNext()) {
String item = sc.next();
double price = sc.nextDouble();
System.out.format("Price of %s is:
%.2f%n", item, price);
}
} catch (IOException e) {
e.printStackTrace();
}
}
}
```

Input and Output from the Command-Line



Input and Output from the Command-Line

- Standard streams in Java
  - `System.in`: A byte `InputStream` to take user input
  - `System.err`: A `PrintStream` to output error messages
  - `System.out`: A `PrintStream` to output normal messages

Input and Output from the Command-Line

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class ReadName {
public static void main(String[] args) {
try(
BufferedReader reader = new BufferedReader(
new InputStreamReader(System.in));
) {
System.out.println("What is your name, user?");
String name = reader.readLine();
if (name.trim().equals(""))
throw new IllegalArgumentException();
System.out.println("Welcome, " + name);
} catch (IllegalArgumentException e) {
System.err.println("Error: name cannot be blank!");
} catch (IOException e) {
e.printStackTrace();
}}}
```

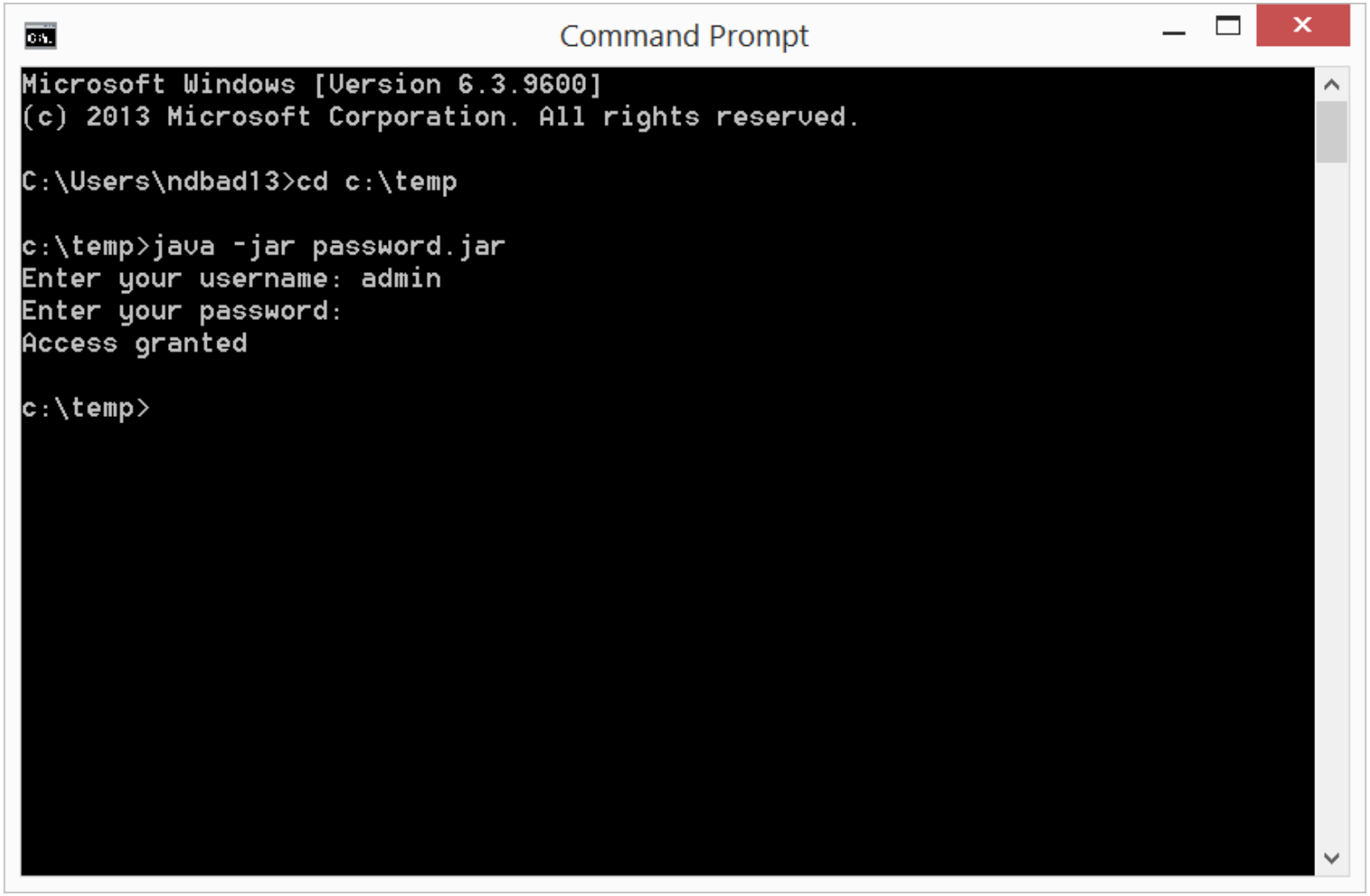
Input and Output from the Command-Line

```
import java.util.Scanner;

public class ReadName {
public static void main(String[] args) {
try(
Scanner sc = new Scanner(System.in);
) {
System.out.println("What is your name, user?");
String name = sc.nextLine();
if (name.trim().equals(""))
throw new IllegalArgumentException();
System.out.println("Welcome, " + name);
} catch (IllegalArgumentException e) {
System.err.println("Error: name cannot be blank!");
}}}
```

Input and Output from the Command-Line

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

public class GetPassword {
public static void main (String args[]) throws IOException {
Console c = System.console();
if (c == null) {
System.err.println("Console object is not available");
System.exit(1);
}
String username = c.readLine("Enter your username: ");
char[] password = c.readPassword("Enter your password: ");
if (username.equals("admin") && new String(password).equals("swordfish")) {
c.writer().println("Access granted");
} else {
c.writer().println("Oops, didn't recognize you there");
}}}
```

Input and Output from the Command-Line



Input and Output From Files

- Java NIO2 File Input and Output
- Legacy File Input and Output

Java NIO2 File Input and Output

- `Path` Interface
- `Files` Class

Path Interface

- A path specifies the access path to a directory or file
  - E.g. c:\projects\java book\bmi.class
- Part of `java.nio.file` package
- `Path` is an interface type whereas `Paths` is a normal class
- `Path` objects contain the filename and directory list used to build the path, and can be used to examine and work with files
- `Path myPath = Paths.get("C:\\projects\\outline.txt");`

Path Interface

Method	Meaning
`String myPath.toString()`	Returns the string representation of the `Path` object
`Path myPath.getFileName()`	Returns the filename or the last element in the `Path` object
`Path myPath.getName(int i)`	Returns the `Path` element corresponding to the specified index
`int myPath.getNameCount()`	Returns the number of elements in the path
`Path myPath.subpath(int i, int j)`	Returns the subsequence of the `Path` (not including a root element) as specified by beginning and ending indices
`Path myPath.getParent()`	Returns the `Path` of the parent directory of this path
`Path myPath.getRoot()`	Returns the root of the path
`Path myPath.normalize()`	Cleans up redundancies from a path and returns the cleaned-up result

`Files` Class

- Part of `java.nio.file` package
- Offers methods for reading, writing, and manipulating files and folders

Method	Meaning
`boolean Files.exists(Path pathToCheck, LinkOption... options)`	Tests whether a path exists
`boolean Files.notExists(Path pathToCheck, LinkOption... options)`	Tests whether a path does not exist.
`boolean Files.isReadable(Path pathToCheck)`	Tests whether a path is readable.
`boolean Files.isWritable(Path pathToCheck)`	Tests whether a path is writable.
`boolean Files.isExecutable(Path pathToCheck):`	Tests whether a path is executable.
`boolean Files.isDirectory(Path pathToCheck)`	Tests whether the path represents a Directory
`boolean Files.isSameFile(Path firstPath, Path secondPath)`	Tests whether two paths resolve to the same location

`Files` Class

```
try {
Files.delete(path);
} catch (NoSuchFileException x) {
// File does not exist
} catch (DirectoryNotEmptyException x) {
// The directory is not empty
} catch (IOException x) {
// File permission problem, no access
}
```

Note: `Files.deleteIfExists(Path pathToDelete)` does not throw exception!

`Files` Class

Method	Meaning
`Path copy(Path source, Path target, CopyOptions... options)`	Copies a source file to a target file
`Path move(Path source, Path target, CopyOptions... options)`	Moves a source file to a target file

Method	Meaning
`byte[] Files.readAllBytes(Path path)`	Reads all the bytes from a file to a byte array
`List<String> readAllLines(Path path, Charset cs)`	Reads all lines from a file to a list of strings

`Files` Class

```
import java.io.IOException;
import java.nio.charset.Charset;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.ArrayList;
import java.util.List;

public class ShowGroceries {
public static void main(String[] args) {
List<String> groceries = new ArrayList<>();
try {
groceries = Files.readAllLines(
Paths.get("groceries.txt"),
Charset.defaultCharset());
} catch (IOException e) {
e.printStackTrace();
}
for (String item : groceries) {
System.out.println("Don't forget to pickup: " + item);}}}
```

`Files` Class

Method	Meaning
`Path write(Path path, byte[] bytes, OpenOption... options):`	Writes bytes to a file
`Path write(Path path, Iterable<? extends CharSequence> lines, Charset cs, OpenOption... options)`	Writes lines of text to a file

Method	Meaning
`BufferedReader Files.newBufferedReader(Path path, Charset cs)`	Returns a buffered character stream to read a text file in an efficient manner using the given character set to decode
`BufferedWriter Files.newBufferedWriter(Path path, Charset cs, OpenOption... options)`	Returns a buffered character stream to write a text file in an efficient manner using the given character set to encode

`Files` Class

```
Path file = Paths.get("groceries.txt");
try (BufferedReader reader = Files.newBufferedReader(file)) {
String line = null;
while ((line = reader.readLine()) != null) {
System.out.println(line);
}
} catch (IOException x) {
System.err.println("Something went wrong");
}
```

`Files` Class

Method	Meaning
`Files.createFile(Path path)`	Create an empty file
`Files.createTempFile(Path folder, String prefix, String suffix)`	Create temporary file in the specified folder
`Files.createDirectory(Path path)`	Create an empty directory
`Files.createTempDirectory(Path folder, String prefix)`	Create temporary directory in the specified folder
`Files.newDirectoryStream(Path folder)`	List all the contents of a directory

Files Class

```
import java.io.IOException;
import java.nio.file.DirectoryIteratorException;
import java.nio.file.DirectoryStream;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;

public class ShowDirectory {
public static void main(String[] args) {
Path folder = Paths.get("C:\\");
try (DirectoryStream<Path> stream = Files.newDirectoryStream(folder)) {
for (Path entry: stream) {
System.out.println(entry.getFileName());
}
} catch (IOException | DirectoryIteratorException x) {
System.err.println("An error occurred");
}}}
```

Output:

$Recycle.Bin
BOOTNXT
Documents and Settings
eclipse
hiberfil.sys
MSOCache
pagefile.sys
PerfLogs
Program Files
Program Files (x86)
…

Note: `Files.newDirectoryStream(dir, "*.{txt,doc,pdf}"));`

Files Class

```
import java.io.IOException;
import java.nio.file.DirectoryIteratorException;
import java.nio.file.DirectoryStream;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.ArrayList;
import java.util.List;

public class RecursiveOperations {

public static void delete(Path source) throws IOException
{
if (Files.isDirectory(source)) {
for (Path file : getFiles(source))
delete(file);
}
Files.delete(source);
System.out.println("DELETED "+source.toString());
}
```

```
public static List<Path> getFiles(Path dir) {
// Gets all files, but puts directories first
List<Path> files = new ArrayList<>();
if (!Files.isDirectory(dir)) return files;
try (DirectoryStream<Path> stream =
Files.newDirectoryStream(dir)) {
for (Path entry : stream)
if (Files.isDirectory(entry))
files.add(entry);
} catch (IOException | DirectoryIteratorException x)
{
}
try (DirectoryStream<Path> stream =
Files.newDirectoryStream(dir)) {
for (Path entry : stream)
if (!Files.isDirectory(entry))
files.add(entry);
} catch (IOException | DirectoryIteratorException x)
{
}
return files;}}
```

Files Class

```
public static void copy(Path source, Path target) throws IOException {
if (Files.exists(target) && Files.isSameFile(source, target))
return;
if (Files.isDirectory(source)) {
Files.createDirectory(target);
System.out.println("CREATED "+target.toString());
for (Path file : getFiles(source))
copy(file, target.resolve(file.getFileName()));
} else {
Files.copy(source, target);
System.out.println(
"COPIED "+source.toString()+" -> "+target.toString());}}

public static void move(Path source, Path target) throws IOException {
if (Files.exists(target) && Files.isSameFile(source, target))
return;
copy(source, target);
delete(source);}
```

Files Class

```
public static Set<Path> search(Path start, String glob,boolean includeDirectories, boolean
includeFiles) {
PathMatcher matcher = FileSystems.getDefault().getPathMatcher
("glob:" + glob);
Set<Path> results = new HashSet<>();
search(start, matcher, includeDirectories, includeFiles, results);
return results;
}

private static void search(Path path, PathMatcher matcher,
boolean includeDirectories, boolean includeFiles, Set<Path> results) {
if (matcher.matches(path.getFileName())
&& ((includeDirectories && Files.isDirectory(path))
|| (includeFiles && !Files.isDirectory(path)))) {
results.add(path);
}
for (Path next : getFiles(path))
search(next, matcher, includeDirectories, includeFiles, results);
}
```

Files Class

```
public static void main(String args[]) throws IOException {
// Set up test directory
try {
delete(Paths.get("C:\\javatest\\"));
delete(Paths.get("C:\\javatest2\\"));
} catch(NoSuchFileException e) {}
Files.createDirectory(Paths.get("C:\\javatest\\"));
Files.createDirectory(Paths.get("C:\\javatest\\subdir\\"));
Files.createFile(Paths.get("C:\\javatest\\text1.txt"));
Files.createFile(Paths.get("C:\\javatest\\text2.txt"));
Files.createFile(Paths.get("C:\\javatest\\other.txt"));
Files.createFile(Paths.get("C:\\javatest\\subdir\\text3.txt"));
Files.createFile(Paths.get("C:\\javatest\\subdir\\other.txt"));
// Test our methods
copy(Paths.get("C:\\javatest\\subdir\\"),
Paths.get("C:\\javatest\\subdircopy\\"));
System.out.println(search(Paths.get("C:\\javatest"),
"text*.txt", true, true));
move(Paths.get("C:\\javatest\\subdircopy\\"),
Paths.get("C:\\javatest\\subdircopy2\\"));
System.out.println(search(Paths.get("C:\\javatest\\"),
"text*.txt", true, true));
copy(Paths.get("C:\\javatest\\"), Paths.get("C:\\javatest2\\"));}
```

Files Class

Output:

```
CREATED C:\javatest\subdircopy
COPIED C:\javatest\subdir\other.txt -> C:\javatest\subdircopy\other.txt
COPIED C:\javatest\subdir\text3.txt -> C:\javatest\subdircopy\text3.txt
[C:\javatest\subdir\text3.txt, C:\javatest\text2.txt,
C:\javatest\subdircopy\text3.txt, C:\javatest\text1.txt]
CREATED C:\javatest\subdircopy2
COPIED C:\javatest\subdircopy\other.txt -> C:\javatest\subdircopy2\other.txt
COPIED C:\javatest\subdircopy\text3.txt -> C:\javatest\subdircopy2\text3.txt
DELETED C:\javatest\subdircopy\other.txt
DELETED C:\javatest\subdircopy\text3.txt
DELETED C:\javatest\subdircopy
[C:\javatest\subdir\text3.txt, C:\javatest\subdircopy2\text3.txt,
C:\javatest\text2.txt, C:\javatest\text1.txt]
CREATED C:\javatest2
CREATED C:\javatest2\subdir
COPIED C:\javatest\subdir\other.txt -> C:\javatest2\subdir\other.txt
COPIED C:\javatest\subdir\text3.txt -> C:\javatest2\subdir\text3.txt
CREATED C:\javatest2\subdircopy2
COPIED C:\javatest\subdircopy2\other.txt -> C:\javatest2\subdircopy2\other.txt
COPIED C:\javatest\subdircopy2\text3.txt -> C:\javatest2\subdircopy2\text3.txt
COPIED C:\javatest\other.txt -> C:\javatest2\other.txt
COPIED C:\javatest\text1.txt -> C:\javatest2\text1.txt
COPIED C:\javatest\text2.txt -> C:\javatest2\text2.txt
```

Legacy File Input and Output

- `java.io.File` class
  - Some methods don't throw exceptions when an error occurs, or do not provide enough information to know the root cause behind a failure.
  - Well-defined support for links is lacking.
  - Accessing file metadata can be difficult and slow.
  - Fetching information over a network introduces scalability issues.
  - Some methods do not work consistently on various operating systems and platforms.

Legacy File Input and Output

```
import java.io.File;

public class ShowDirectory {
public static void main(String[] args) {
File folder = new File("C:\\");
for (File entry : folder.listFiles()) {
System.out.println(entry.getName());
}}}
```

Conclusions

- General Input and Output
- Input and Output in Java
- Streams
- Scanners
- Input and Output from the Command-Line
- Input and Output from Files