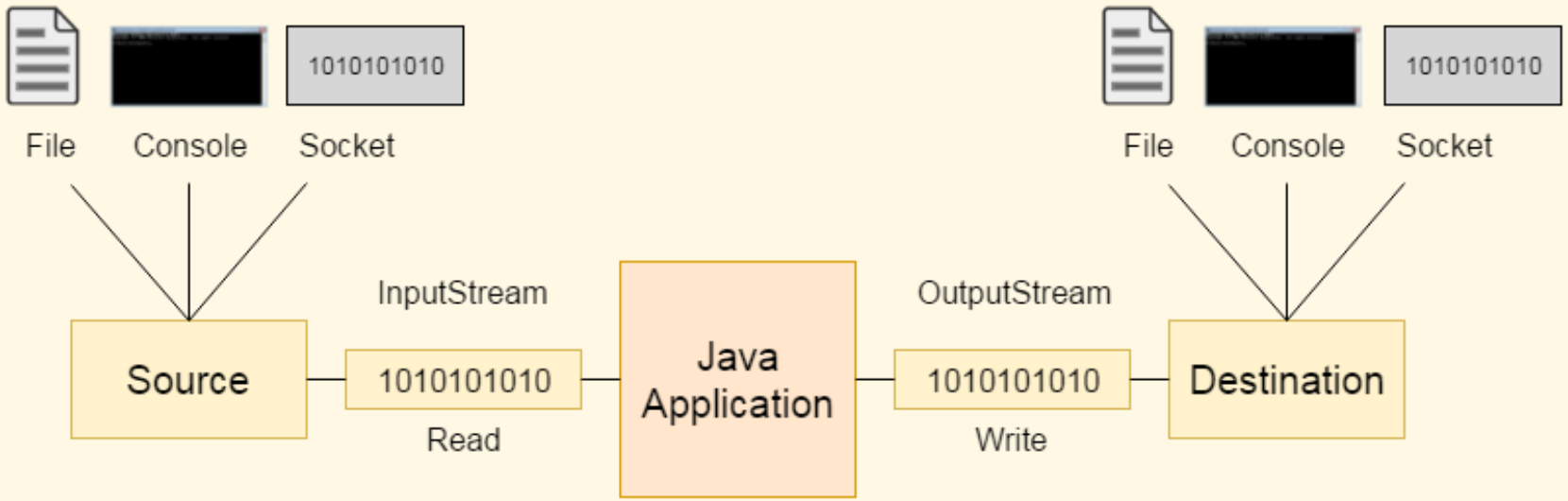
Input and Output (I/O)

- System I/O
- File I/O
- NIO

Introduction

- Java I/O (Input and Output) is used to process the input and produce the output
- Java uses the concept of a stream to make I/O operation fast
  - A stream is a sequence of data (composed of bytes)
- The `java.io` package contains all the classes required for input and output operations



System I/O

General

- For each program, there are 3 streams that are created automatically:
  - A standard output (`System.out` object): is used to print the program output, and can be considered as a virtual write-only file
  - A standard error output (`System.err` object): like `System.out` but is usually used to print error or warning messages when the program encounters
  - A standard input (`System.in` object): can be considered as a virtual read-only file
- By default, all the 3 above streams are attached to the console, but they can be redirected to other devices such as a file on disk, or printer,…

Standard Output

- Print formatted values:
  - `System.out.print(value)`
    `System.out.println(value)`
    where `value` can be of any type
  - `System.out.printf(String format, Object ...args)`
- Write binary data:
  - `System.out.write(int byte)`
    `System.out.write(byte[] buf, int off, int len)`
- `System.err` has the same methods as `System.out`

Standard Input

- `System.in` has only a few methods to support the low-level reading:
  - `int System.out.read(int byte)`: Reads one byte
  - `int System.out.read(byte[] buf, int off, int len)`: Reads up to `len` bytes
- Since it's inconvenient to use `System.in` directly to read meaningful data, one should use a supporting class:
  - Using `java.util.Scanner`:

```
Scanner in = new Scanner(System.in);
String line = in.nextLine();
double b = in.nextDouble();
String c = in.next();
String d = in.next("[a-zA-Z1-9]*");
```

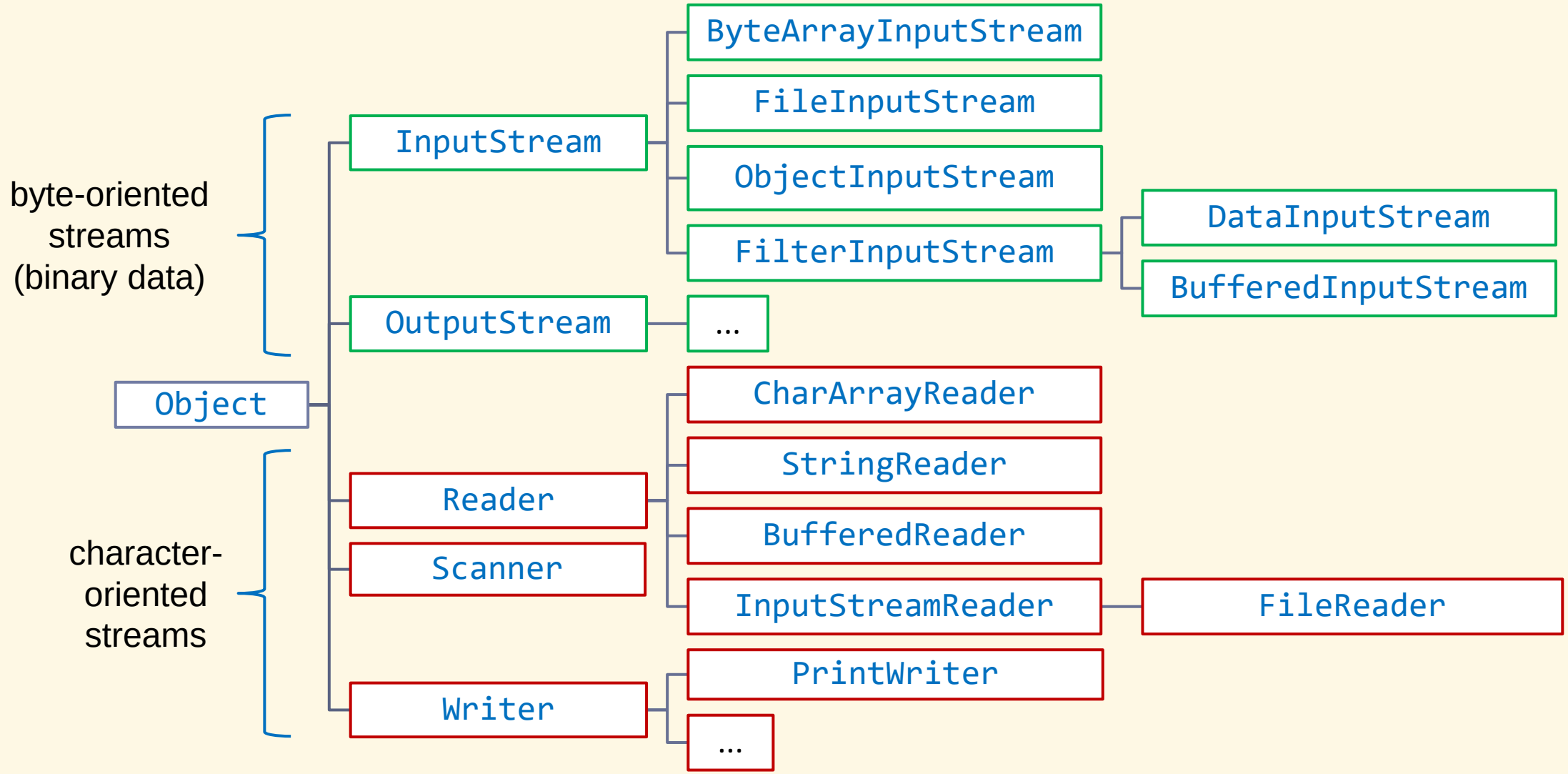
Standard Input/Output Redirection

Name	Description	Name	Description
`&0`	stdin	`nul`	Discard all data
`&1`	stdout	`prn`, `lpt1-9`	Printer
`&2`	stderr	`con`	Console
`aux`	AUX port	`com1-9`	COM ports

- Examples:
  - Redirect both stdout and stderr output to `result.txt` file
    - `C:\>dir *.dat >result.txt 2>&1`
  - Redirect stdout to printer and stderr output to `error.log` file
    - `C:\>stuff >prn 2>error.log`
  - Redirect stdin to `input.txt` file and stdout to `output.txt` file
    - `C:\>process <input.txt >output.txt`
  - Create a pipe (output of one command is input of the other)
    - `C:\>type source.c | more`

File I/O

The Hierarchy



Read Characters One by One

```
import java.io.*;

public class PrintTextFileContent {
    public static void main(String argv[]) throws IOException {
        FileReader reader = new FileReader("abc.txt");

        int v;
        while ((v = reader.read()) != -1)
            System.out.print((char)v);

        reader.close();  // don't forget this
    }
}
```

Read File in Blocks

```
import java.io.*;
import java.util.Arrays;

public class PrintTextFileContent {
    public static void main(String argv[]) throws IOException {
        FileReader reader = new FileReader("abc.txt");

        char[] buffer = new char[512];
        for (int n = 0; n >= 0; n = reader.read(buffer))
            System.out.print(Arrays.copyOfRange(buffer, 0, n));

        reader.close();
    }
}
```

Read File in Lines

```
import java.io.*;

public class PrintTextFileContent {
    public static void main(String argv[]) throws IOException {
        FileReader fileReader = new FileReader("abc.txt");
        BufferedReader reader = new BufferedReader(fileReader);

        String line;
        while ((line = reader.readLine()) != null)
            System.out.println(line);


        reader.close(); // no need to do fileReader.close()
    }
}
```

Read Formatted Values

```
import java.io.*;
import java.util.Scanner;

public class Hello {
    public static void main(String argv[]) throws IOException {
        FileReader reader = new FileReader("abc.txt");
        Scanner scanner = new Scanner(new BufferedReader(fileReader));

        String line = scanner.nextLine();
        int i = scanner.nextInt();
        double d = scanner.nextDouble();

        // ...

        scanner.close(); // no need to do reader.close()
    }
}
```

Write Characters and Strings to File

- Using `FileWriter`:
  - ```
    FileWriter writer = new FileWriter("abc.txt", false);
    writer.write("Hello");
    writer.write('*');
    writer.close();
    ```
- Using `BufferedWriter` (for better performance):
  - ```
    FileWriter fileWriter = new FileWriter("abc.txt", false);
    BufferedWriter writer = new BufferedWriter(fileWriter);
    // ... the rest are the same as above
    ```

Write Values to File with Formatting

- Using `PrintWriter` (together with `BufferedWriter` for performance):
  - ```
    FileWriter fileWriter = new FileWriter("abc.txt", false);
    PrintWriter writer = new PrintWriter(new
    BufferedWriter(fileWriter));

    writer.println(30.234);
    writer.print("Hello");
    writer.printf("%nToday is %02d/%02d/%04d", day, month, year);

    writer.close();
    ```
    ```

Working with Binary Data (1)

- Use `FileInputStream` and `FileOutputStream` for binary data, and optionally together with `BufferedInputStream` and `BufferedOutputStream` for performance
  - ```
    BufferedOutputStream stream = new BufferedOutputStream(
        new FileOutputStream("data.bin", false));

    double value = 123.456;
    byte[] bytes = new byte[8];
    ByteBuffer.wrap(bytes).putDouble(value);   // ByteBuffer is from java.nio
    stream.write(bytes);

    stream.close();
    ```
  - ```
    BufferedInputStream stream = new BufferedInputStream(
        new FileInputStream("data.bin"));

    byte[] bytes = new byte[8];
    stream.read(bytes);
    double value = ByteBuffer.wrap(bytes).getDouble();

    stream.close();
    ```

Working with Binary Data (2)

- `DataInputStream` and `DataOutputStream` can be used for more functionality
  - ```
    DataOutputStream stream = new DataOutputStream(
      new BufferedOutputStream(new FileOutputStream("data.bin")));
    stream.writeInt(20);
    stream.writeDouble(123.456);
    stream.write(rawData);
    stream.close();
    ```
  - ```
    DataInputStream stream = new DataInputStream(
      new BufferedInputStream(new FileInputStream("data.bin")));
    int i = stream.readInt();
    double d = stream.readDouble();
    stream.read(rawData, 0, 32);
    stream.close();
    ```

Working with Binary Data (3)

- `ObjectInputStream` and `ObjectOutputStream` are useful for serialization of objects (which must implement `Serializable` interface)
  - ```
    class Dog implements Serializable {
        String name;
        int birthYear;
        // ...
    }
    ```
  - ```
    Dog dog = new Dog1("Tyson", 2005);
    FileOutputStream file = new FileOutputStream("data.bin");
    ObjectOutputStream stream = new ObjectOutputStream(file);
    stream.writeInt(100);
    stream.writeObject(dog);
    stream.close();
    ```
  - ```
    FileInputStream file = new FileInputStream("data.bin");
    ObjectInputStream stream = new ObjectInputStream(file);
    int i = stream.readInt();
    Dog dog = (Dog)stream.readObject();
    stream.close();
    ```
    ```

Bridging Byte Streams to Character Streams

- Output stream to writer:
  - ```
    OutputStream stream = ...;
    Writer writer = new OutputStreamWriter(stream);
    ```
- Input stream to reader:
  - ```
    InputStream stream = ...;
    Reader reader = new InputStreamReader(stream);
    ```

`try` with Resources

Starting Examples

- Will the readers be closed if something goes wrong?
  - ```
    FileReader fileReader = new FileReader("abc.txt");
    BufferedReader reader = new BufferedReader(fileReader);
    // ...
    reader.close();
    ```
- How to close them in the `catch` block?
  - ```
    try {
        FileReader fileReader = new FileReader("abc.txt");
        BufferedReader reader = new BufferedReader(fileReader);
        // ...
        reader.close();
    } catch(IOException e) {
        // no access to the readers
    }
    ```

Solution?

```
BufferedReader reader;
try {
    FileReader fileReader = new FileReader("abc.txt");
    reader = new BufferedReader(fileReader);
    // ...
    reader.close();
} catch(IOException e) {
    reader.close();
}
```

- Other problems:
  - Redundant code
  - What if the 2nd `reader.close()` internally throws an exception before closing `fileReader`?

Prior to Java 7

- `finally` block was widely used in this case:
  - ```
    BufferedReader reader;
    try {
        FileReader fileReader = new FileReader("abc.txt");
        reader = new BufferedReader(fileReader);
        // ...
    } catch(IOException e) {
        // ...
    } finally {
        reader.close();
    }
    ```
- What problems does it solve?
  - Only the redundant code
    ```

Correct Solution

- Use the `try` with resources block introduced in Java 7:

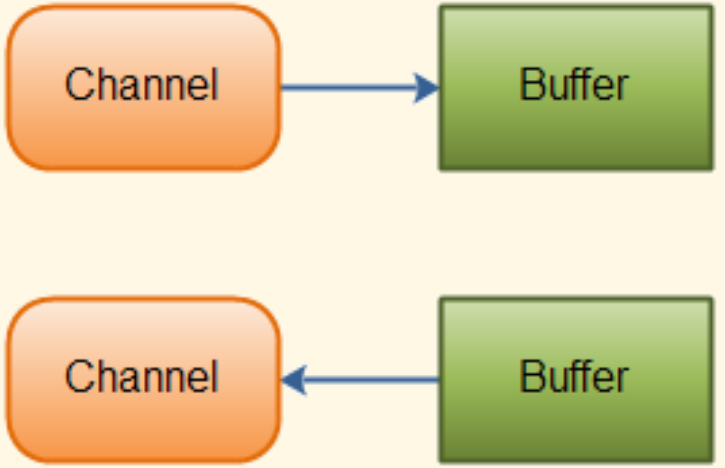
```
try (FileReader fileReader = new FileReader("abc.txt");
     BufferedReader reader = new BufferedReader(fileReader)) {
     // ...
} catch(IOException e) {
     // ...
}
```

  - The resources are guaranteed to be released after the execution of the `try` block, by invoking their `close()` methods
  - The `close()` methods of resources are called in the *opposite* order of their creation
  - The resources need to implement the `AutoCloseable` interface

Java NIO

Introduction

- NIO = New IO or Non-blocking IO, is an alternative IO API to the standard Java IO and Java Networking API's
  - `java.io` is blocking: the execution of the program is blocked until the requested IO operation finishes
  - `java.nio` is mostly non-blocking
- NIO uses *channels* and *buffers* instead of byte streams and character streams
  - Data is read from a channel to a buffer, or written from a buffer to a channel

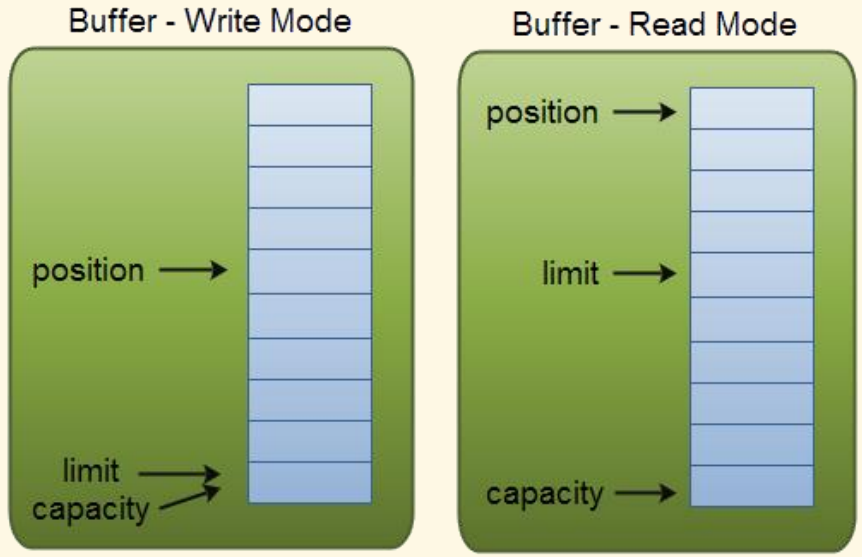


Channels

- Channels are similar to streams with a few differences:
  - A channel can be both read and written to, while streams are one-way
  - Read and write operations can be asynchronous
  - Channels always read to, or write from, a buffer
- Important channels:
  - `FileChannel`: works with files synchronously
  - `AsynchronousFileChannel`: works with files asynchronously
  - `DatagramChannel`: works with network connections via UDP
  - `SocketChannel`: works with network connections via TCP
  - `ServerSocketChannel`: allows to listen for incoming TCP connections

Buffers

- A buffer is essentially a block of memory into which you can write data, which you can then later read again
  - Has 2 modes (read, write) and 3 properties (capacity, position and limit)
- Basic usage with 4 steps:
  - Write data into it
  - Call `flip()` method to make it ready for read
  - Read data out of it
  - Call `clear()` or `compact()` methods to make it ready for writing again
- Buffer types: `ByteBuffer`, `MappedByteBuffer`, `CharBuffer`, `DoubleBuffer`, `FloatBuffer`, `IntBuffer`, `LongBuffer`, `ShortBuffer`



Example: Synchronous Reading

```
try (RandomAccessFile file = new RandomAccessFile("data.txt", "rw")) {
    FileChannel channel = file.getChannel();
    ByteBuffer buf = ByteBuffer.allocate(48);

    int bytesRead = channel.read(buf);
    while (bytesRead != -1) {
        System.out.println("Read " + bytesRead);
        buf.flip();

        while (buf.hasRemaining())
            System.out.print((char) buf.get());

        buf.clear();
        bytesRead = channel.read(buf);
    }
} catch (IOException e) {
    // ...
}
```

Example: Asynchronous Writing

```
AsynchronousFileChannel channel = AsynchronousFileChannel.open(Paths.get("test.txt"), StandardOpenOption.WRITE);
ByteBuffer buffer = ByteBuffer.allocate(1024);
buffer.put("test data".getBytes());
buffer.flip();

CompletableFuture<Void> future = new CompletableFuture();
channel.write(buffer, 0, buffer, new CompletionHandler<Integer, ByteBuffer>() {
    @Override
    public void completed(Integer result, ByteBuffer attachment) {
        System.out.println("bytes written: " + result);
        future.complete(null);              // notifies a completion with success
    }

    @Override
    public void failed(Throwable exc, ByteBuffer attachment) {
        System.out.println("Write failed");
        future.completeExceptionally(exc); // notifies a completion with error
    }
});

future.get();   // waits if the writing operation has not finished
channel.close();
```

`Files` Class

- Provides several methods for manipulating files in the file system
  - `Files.exists(Path path, LinkOption options)`
  - `Files.copy(Path source, Path destination, [CopyOption] option)`
  - `Files.move(Path source, Path destination, [CopyOption] option)`
  - `Files.delete(Path path)`
  - `Files.createDirectory(Path path)`
  - `Files.workFileTree(Path path, FileVisitor fv)`: traverses a directory tree recursively

Problems

Write programs to:

1. Convert all characters in a file to upper case
2. Count number of words and number of lines in a file (words separated by space, tab, newline characters)
3. Append a file to another
4. Print the 10^{th} line of a file
5. Insert a line into a file after the 10^{th} line
6. Write information of a student class to a file, then read back and print it
7. Write a function that returns the size of a file whose path is given from parameter