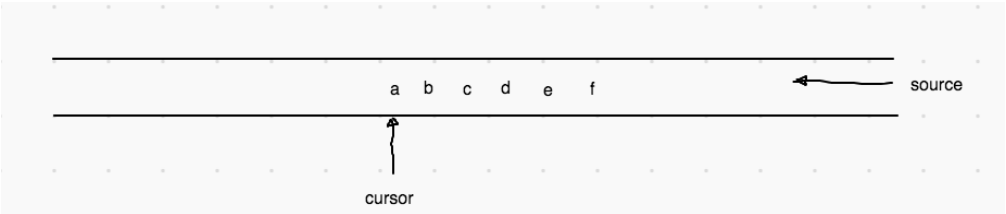
Streams

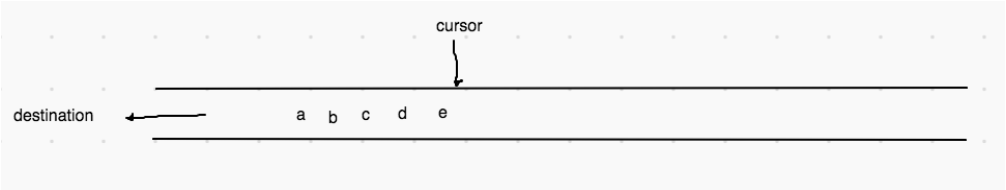
We can think of a stream as a ”river” of data: we are standing at a position alongside the river and we either see the data coming at us (input), or we can see the data moving away from us (output).

Input streams are streams that bring data into our program. These will have a *source* that contains the data our program is working with.



- Operation `read()`: retrieve data and advance.

Output Streams are streams that send data out of our program. These will have a *destination* that will receive the data.



- Operation `write()`: put data onto the output stream

Examples of sources and destinations: files, console, pipes and network data.

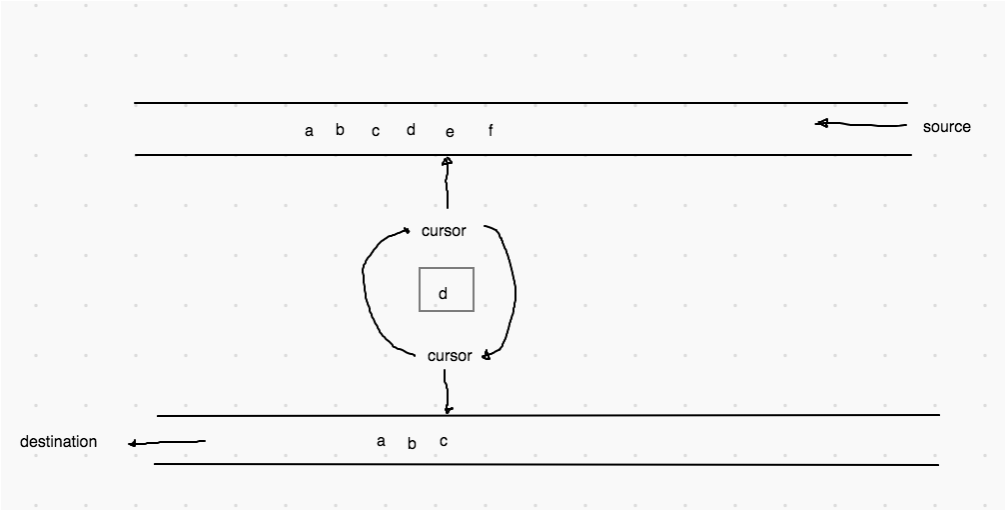
Text streams

- Java has `Reader` and `Writer` classes to process text-based streams.
- Basically the same as their binary alternatives, but `read()` and `write()` process character data.

Example: copy a text file.

Copy all the characters from one file `fileNameSrc` to a new file `fileNameDest`:

Idea: read and write a single char at a time.



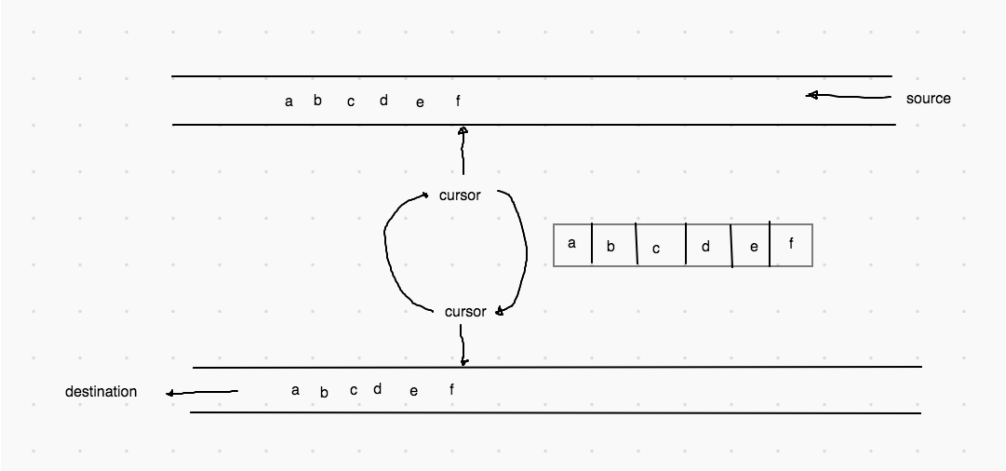

```
public static void copy(String fileNameSrc, String fileNameDest) throws IOException {
    FileReader input = new FileReader(fileNameSrc);
    FileWriter output = new FileWriter(fileNameDest);

    int c = input.read();
    while (c != -1) {
        output.write(c);
        c = input.read();
    }

    input.close();
    output.close();
}
```

Example: copy a text file using an array buffer.

Same as above but we can use the read and write overloads to write a more efficient copy:




```
public static void bufferedCopy(String fileNameSrc, String fileNameDest) throws IOExceptio
    FileReader input = new FileReader(fileNameSrc);
    FileWriter output = new FileWriter(fileNameDest);

    char[] buffer = new char[100];

    int length = input.read(buffer);
    while(length =/= -1) {
        output.write(buffer, 0, length);
        length = input.read(buffer);
    }

    input.close();
    output.close();
}
```

Scanners

Consider a text file (or any stream) with numbers and words:

```
-----------------------------------
-1 123 abc 3.1415\n               |
-----------------------------------
```

To process this data as numbers and strings, `read()`, `read(char[])` don't convert to primitives and strings so we use a `Scanner` instead.

```
FileReader reader = new FileReader("input.txt");
Scanner scanner = new Scanner(reader);

int x;
int y;
String s;
double d;

x = scanner.nextInt();    // -1
y = scanner.nextInt();    // 123
s = scanner.next();       // "abc"
d = scanner.nextDouble(); // 3.1415

scanner.close();
```

What happens if we try to scan incorrect data:

```
FileReader reader = new FileReader("input.txt");
Scanner scanner = new Scanner(reader);

x = scanner.nextInt();    // -1
x = scanner.nextInt();    // 123
x = scanner.nextInt();    // InputTypeMismatch exception!

scanner.close();
```

We can use the ”has-next” methods to protect our calls to next:

```
if(!scanner.hasNext())
    throw new Exception(..);
s = scanner.next();       // "-1"

if(!scanner.hasNextInt())
    throw new Exception(..);
x = scanner.nextInt();    // 123

if(!scanner.hasNextInt())
    throw new Exception(..);
y = scanner.nextInt();

if(!scanner.hasNextDouble())
    throw new Exception(..);
d = scanner.nextDouble();
```

Example: Count the number of lines in a file using a Scanner.

```
public static int lineCount(String fileNameSrc) throws IOException {
    FileReader reader = new FileReader(fileNameSrc);
    Scanner scanner = new Scanner(reader);
    int count = 0;
    while(scanner.hasNextLine()) {
        scanner.nextLine();
        count++;
    }
    return count;
}
```

Example: Count the number of tokens ina file using a Scanner.

Recall that tokens are by default whitespace delimited:

```
public static int tokenCount(String fileNameSrc) throws IOException {
    FileReader reader = new FileReader(fileNameSrc);
    Scanner scanner = new Scanner(reader);
    int count = 0;
    while(scanner.hasNext()) {
        scanner.next();
        count++;
    }
    return count;
}
```

Writing our own input and output streams

Java has designed their stream libraries so that programmers can write their own readers and writers. In Asg #3 you will do just this!

Exercise:

- Code a class called `SubstitutionReader`: it will read from a reader and replace every occurrence of one character with another.

- Code a class called `SubstitutionWriter`: it will write to a writer and replace every occurrence of one character with another.

Videos:

- `https://youtu.be/bcNLLYdTO6g`
- `https://youtu.be/jME2O5YBeuc`