COMP 110/L Lecture 23

Kyle Dewey

Outline

- Reading from files
- Writing to files
- `finally`

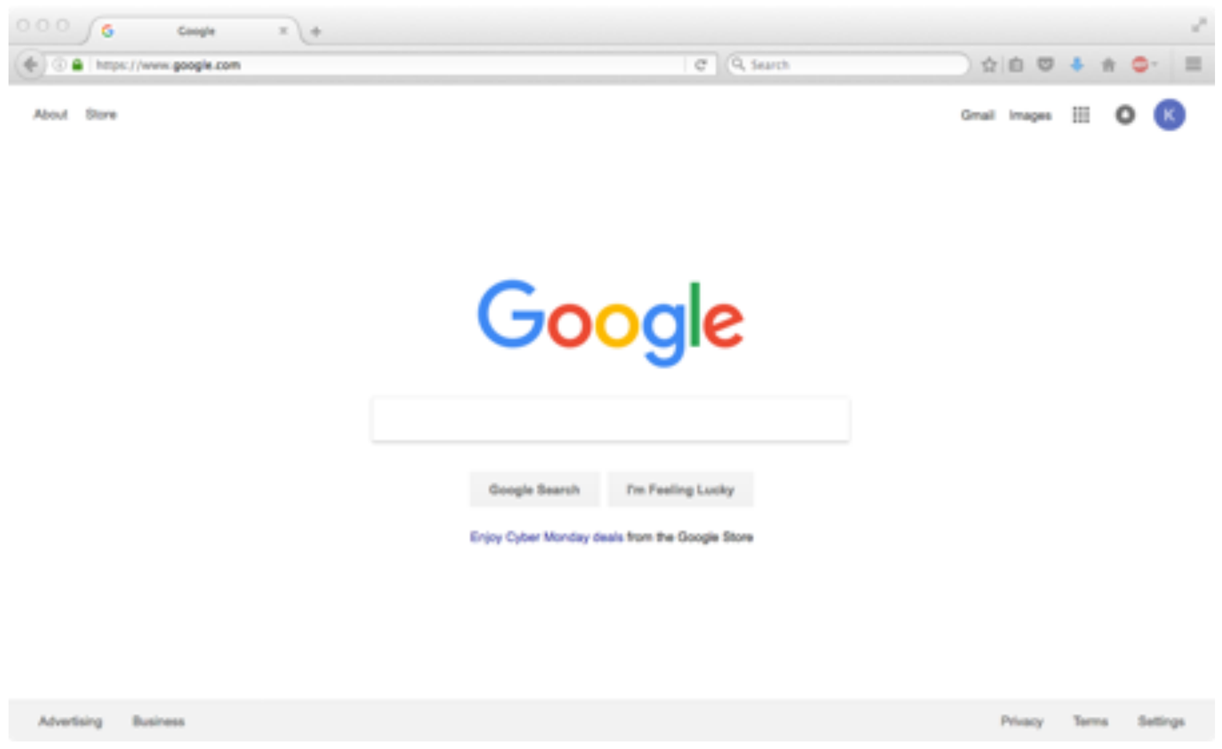
Reading From Files

Motivation

Files act like very large inputs; basis for most things.

Motivation

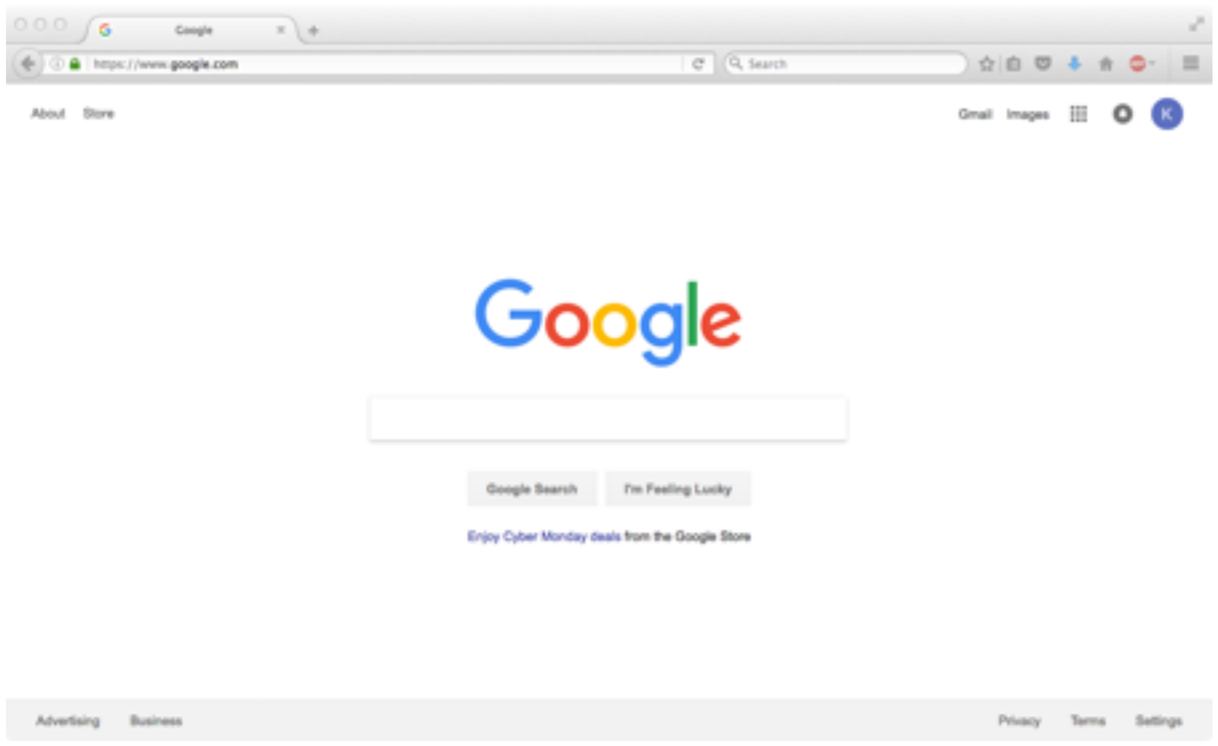
Files act like very large inputs; basis for most things.



-When you “access” a web page, you’re really downloading a HTML file, and subsequently reading the file

Motivation

Files act like very large inputs; basis for most things.



```
public class MyClass {
    ...
}
```

–When you write code, the Java compiler will read it from the file.

Reading from Files

Reading from Files

```
myFile.txt
```

```
myFile.txt
```

Contents

```
one
two
three
```

–On disk somewhere, I have the file myFile.txt

Reading from Files

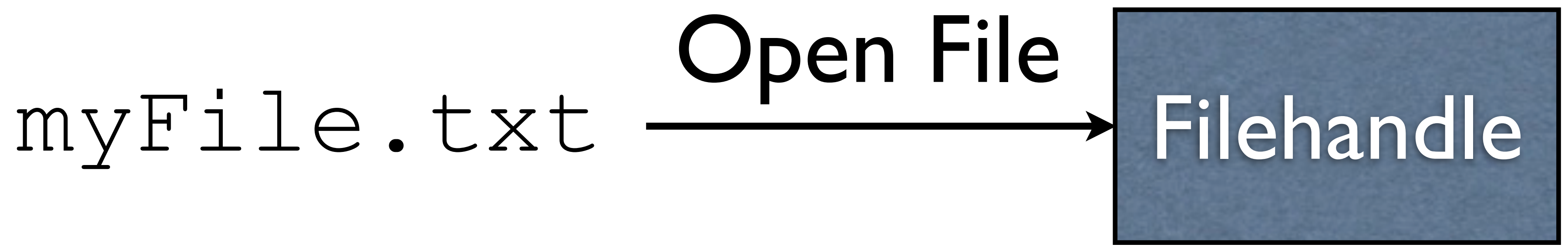
$$\texttt{myFile.txt} \xrightarrow{\text{Open File}}$$

`myFile.txt`
Contents

```
one
two
three
```

-On disk somewhere, I have the file myFile.txt

Reading from Files

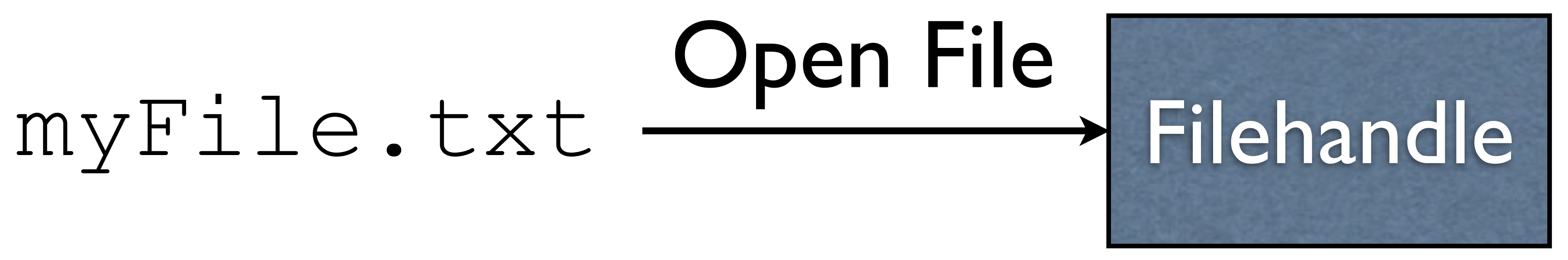


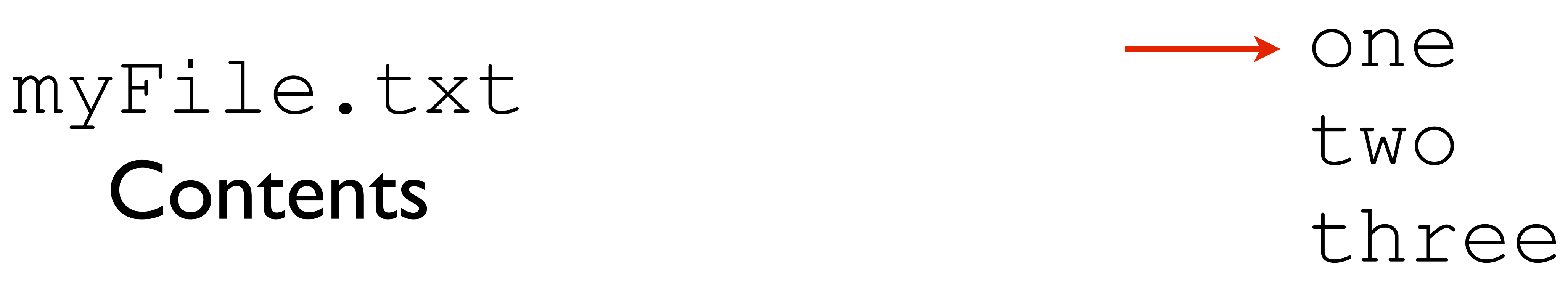
`myFile.txt`
Contents

```
one
two
three
```

-Opening a file creates a “filehandle”, that is, a handle on the open file.
-We call it a “handle” in much the same way as a pan has a handle - this is how to hold the pan (file) and manipulate the pan (file)

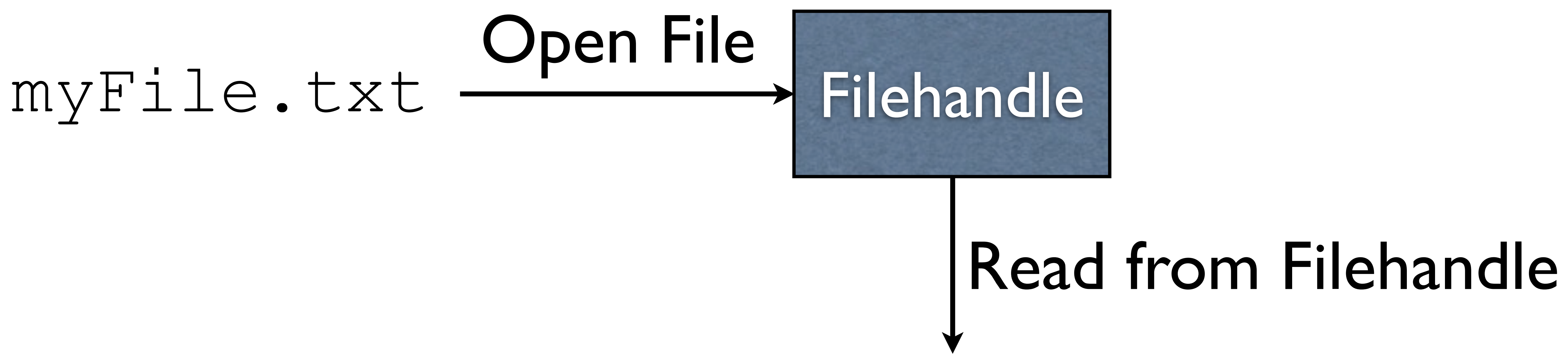
Reading from Files





-The filehandle keeps track of where we are in the file
-Initially, we are right at the start of the file

Reading from Files

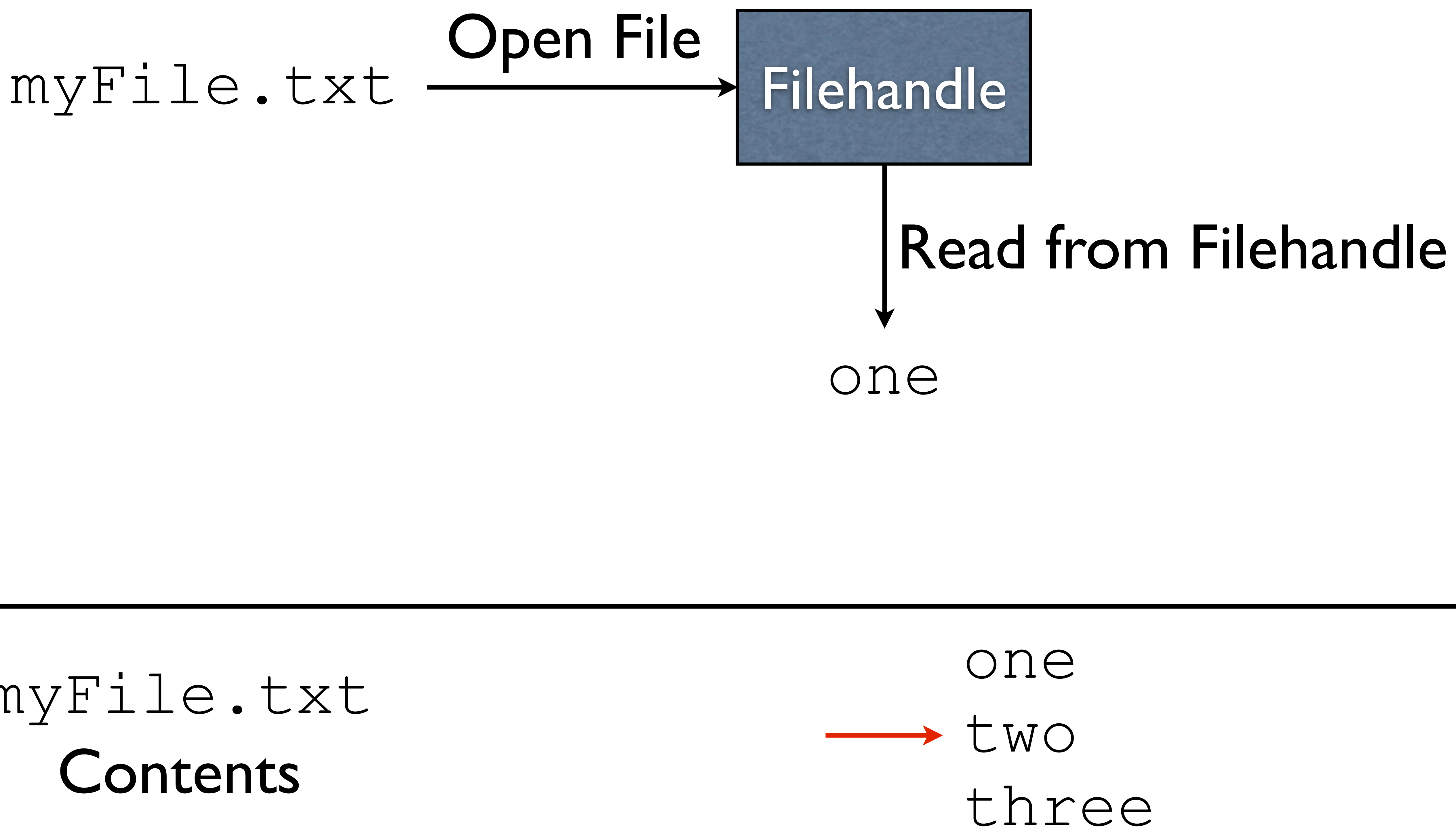


myFile.txt
Contents

```
one
two
three
```

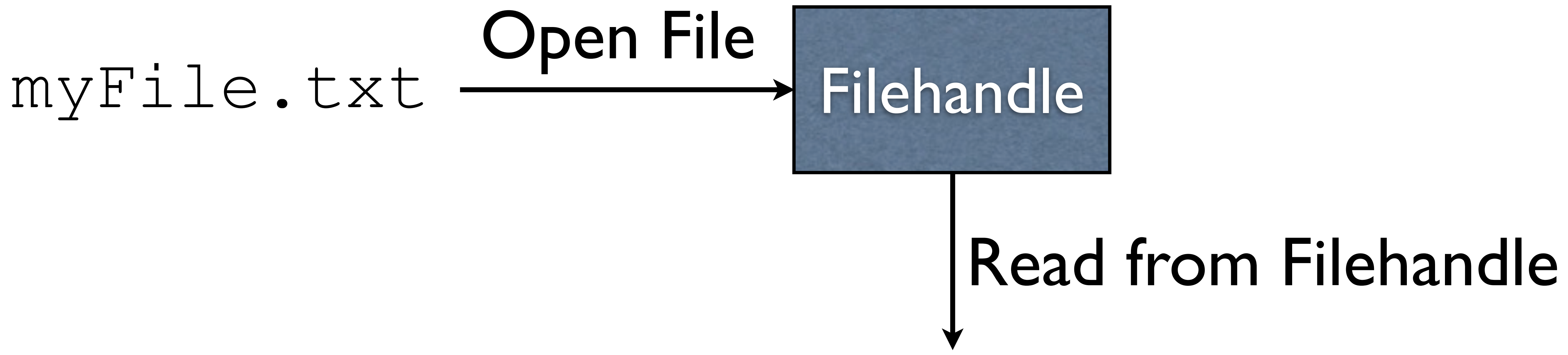
-We can then read from the filehandle

Reading from Files



-When we read from a filehandle, we get whatever is where the file pointer (the red arrow) is
-The file pointer is updated to point to the next position in the file

Reading from Files

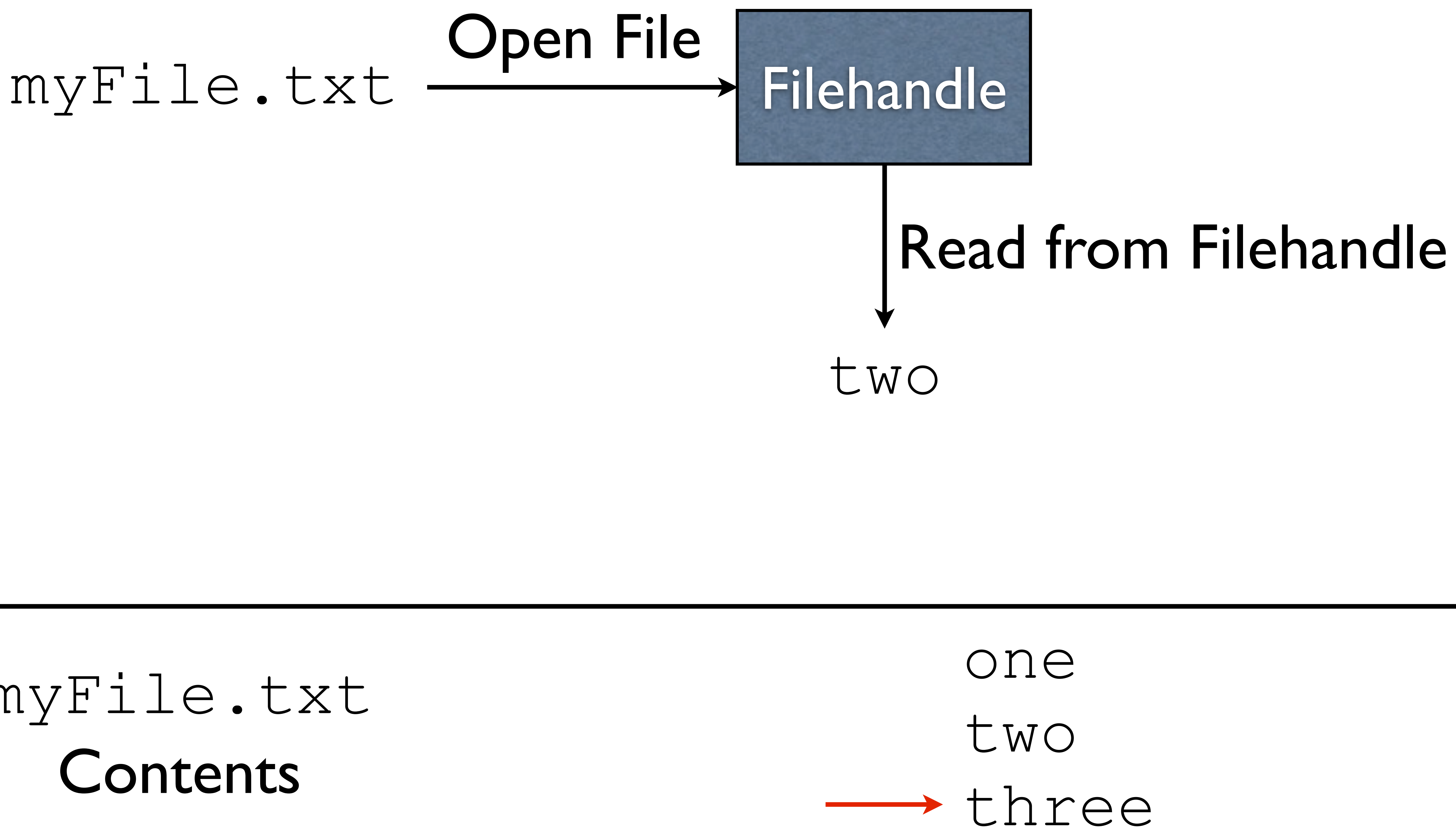


`myFile.txt`
Contents

```
  one
→ two
  three
```

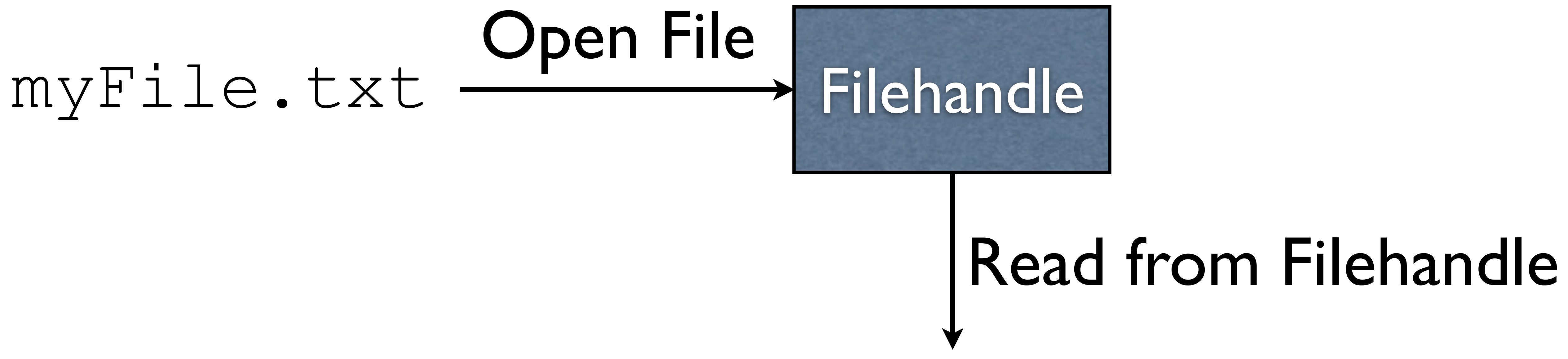
-We can then read again...

Reading from Files

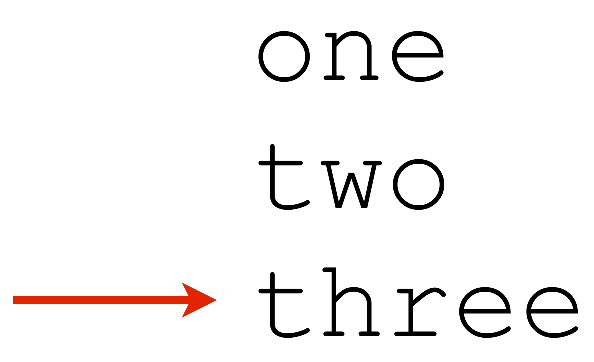


-...resulting in the next value read from the file
-The file pointer (red arrow) is updated as before

Reading from Files

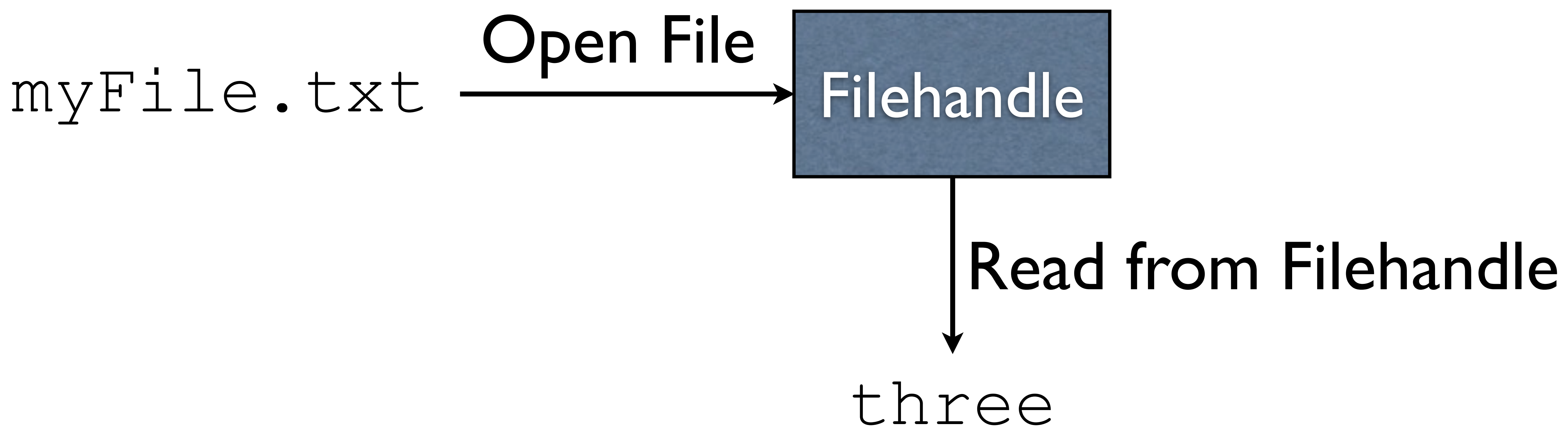


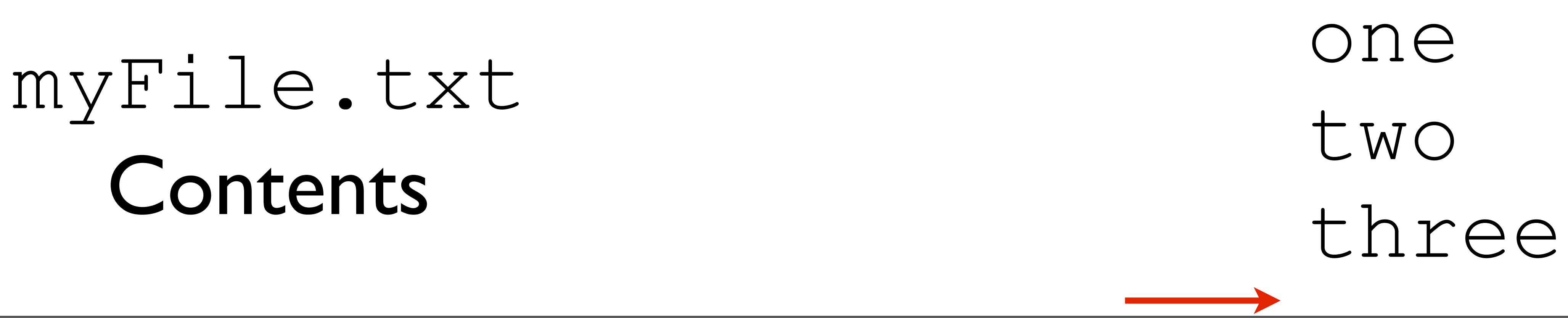
myFile.txt
Contents



-We can read again...

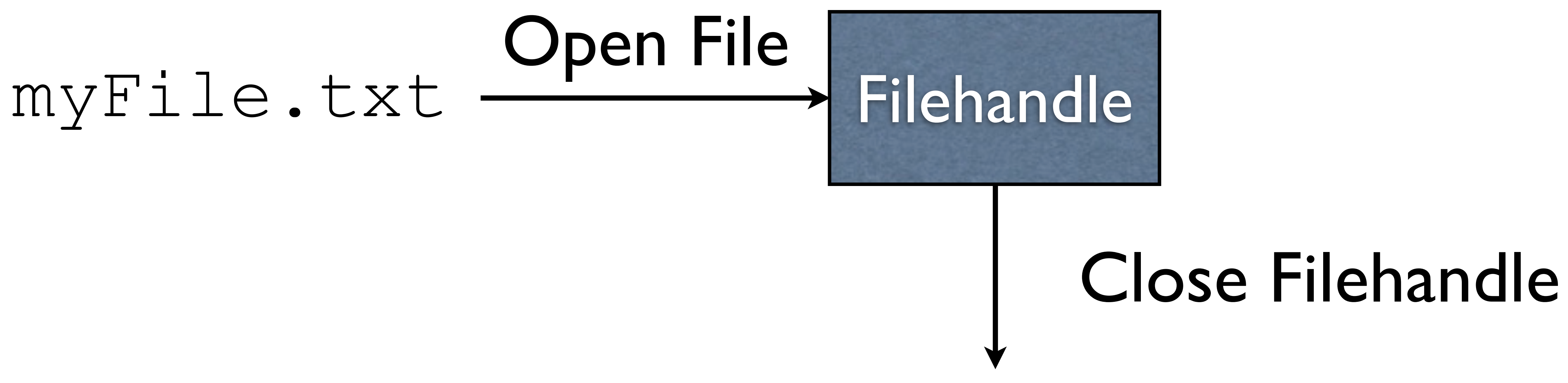
Reading from Files





-...and we get the next thing with a file pointer update, as before

Reading from Files

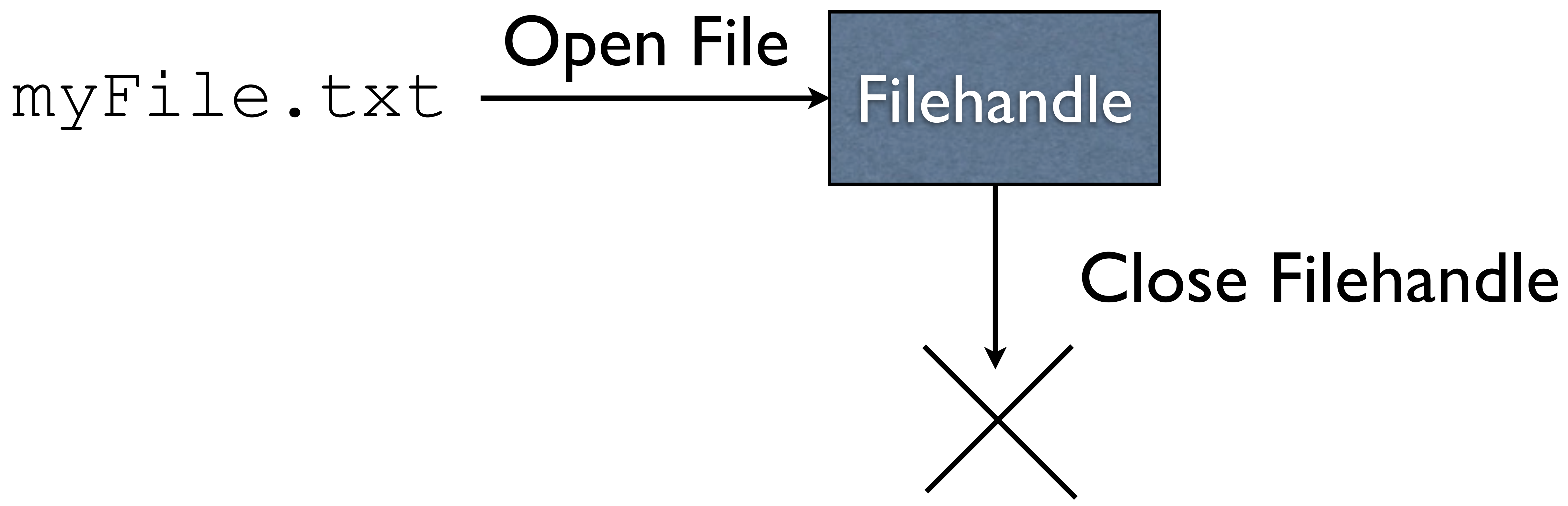


myFile.txt
Contents

```
one
two
three
```

-The last thing we do is close the filehandle when we are done with it

Reading from Files



`myFile.txt`
Contents

```
one
two
three
```

-Closing the filehandle doesn't visibly _do_ anything
-Internally, the file is no longer opened, and we no longer keep track of where we were in the file
-The underlying operating system puts a limit on how many files we can have open at once, so it's important to close a file when we're done with it.

Reading from Files with `Scanner`

Reading from Files with `Scanner`

Step 1: Create `File` object

Reading from Files with `Scanner`

Step 1: Create `File` object

```
File myFile = new File(“myFile.txt”);
```

Reading from Files with `Scanner`

Step 1: Create `File` object

```
File myFile = new File(“myFile.txt”);
```

Step 2: Create `Scanner` object with the `File` object

Reading from Files with `Scanner`

Step 1: Create `File` object

```
File myFile = new File("myFile.txt");
```

Step 2: Create `Scanner` object with the `File` object

```
Scanner input = new Scanner(myFile);
```

Reading from Files with `Scanner`

Step 1: Create `File` object

```
File myFile = new File(“myFile.txt”);
```

Step 2: Create `Scanner` object with the `File` object

```
Scanner input = new Scanner(myFile);
```

Step 3: Read from `Scanner` object

Reading from Files with `Scanner`

Step 1: Create `File` object

```
File myFile = new File(“myFile.txt”);
```

Step 2: Create `Scanner` object with the `File` object

```
Scanner input = new Scanner(myFile);
```

Step 3: Read from `Scanner` object

```
if (input.hasNextLine()) {
  String line = input.nextLine();
  ...
```

Reading from Files with `Scanner`

Step 4: Close `Scanner` object

Reading from Files with `Scanner`

Step 4: Close `Scanner` object

```
input.close();
```

Example:
`ReadFirstLine.java`

Example:

`ReadWholeFile.java`

`FileNotFoundException`

`Scanner`'s constructor will throw a `FileNotFoundException` if the file does not exist.

`FileNotFoundException`

`Scanner`'s constructor will throw a `FileNotFoundException` if the file does not exist.

Example:
`ReadWholeFileWithTry.java`

Writing to Files

Writing to Files

–Same step as with reading files

Writing to Files

Step 1: Create a `File` object

```
File myFile = new File(“myFile.txt”);
```

–Same step as with reading files

Writing to Files

Step 1: Create a `File` object

```
File myFile = new File(“myFile.txt”);
```

Step 2: Create a `FileWriter` object

```
FileWriter fw = new FileWriter(myFile);
```

–Same step as with reading files

Writing to Files

Step 1: Create a `File` object

```
File myFile = new File("myFile.txt");
```

Step 2: Create a `FileWriter` object

```
FileWriter fw = new FileWriter(myFile);
```

Step 3: Create a `BufferedWriter` object

```
BufferedWriter bw =
  new BufferedWriter(fw);
```

–Same step as with reading files

Writing to Files

Step 4: Write to `BufferedWriter` object as needed

```
bw.write(“Hello”);
bw.newLine();
bw.write(“World”);
bw.newLine();
```

–Same step as with reading files

Writing to Files

Step 4: Write to `BufferedWriter` object as needed

```
bw.write(“Hello”);
bw.newLine();
bw.write(“World”);
bw.newLine();
```

Step 5: Close the `BufferedWriter` object

```
bw.close();
```

–Same step as with reading files

Example:
`WriteStrings.java`

`BufferedWriter`

Observation: `PrintWriter` seems to do everything `BufferedWriter` does, so why is `BufferedWriter` needed?

`BufferedWriter`

Observation: `PrintWriter` seems to do everything `BufferedWriter` does, so why is `BufferedWriter` needed?

- Acts as a *buffer*
    - Layer between us saying `write` and the actual writing to the file
- Repeated short writes to files is **slow**
- Buffering idea: collect "writes" together in memory, then write to file all at once

-BufferedWriter transparently collects these writes in memory, and will write to the file when the space in memory is full.

`finally`

Motivation

Sometimes we want to perform an action, whether or not an exception is thrown.

Motivation

Sometimes we want to perform an action, whether or not an exception is thrown.

```
try {
  maybeThrowException();
  maybeDoThis();
} catch (SomeException e) {
  maybeDoThat();
} finally {
  alwaysDoThis();
}
maybeDoTheOtherThing();
```

-In the code above, the only thing guaranteed to always run is maybeThrowException (which might end early if it throws an exception), and alwaysDoThis.
-maybeDoThis will get skipped if maybeThrowException throws an exception
-maybeDoThat will get skipped if the body of the try does not throw a SomeException
-maybeDoTheOtherThing will get skipped if the body of the try throws an exception that isn't a SomeException, or if maybeDoThat throws an exception

Example:
`FinallyExample.java`

Common Use

- `finally` is often used to make sure a file was closed, even if an exception was thrown while manipulating the file
    - `WriteStrings.java` will **not** do this
    - See `WriteStringsFinally.java`