EECS 1720
Building Interactive Systems

Lecture 6 :: Introduction to Exceptions (2)

Recall: (last lecture)

- Different categories of Exceptions
  - Many classes use Exceptions to indicate unforeseen issues that occur at **runtime**
  - allows for possibility of recovery in code
- Exceptions are objects that get instantiated & "thrown"
- May be "caught" using try{} / catch {} blocks
  - If code in try{} triggers an exception, code suspended and program re-routed to any catch() blocks immediately following
  - catch offers some alternative code that can be run instead of the statements that triggered the exception
- If not handled, the program typically will crash with some error messages (stack trace + info from exception that occurred)

Categories of Exceptions (3 basic kinds)

1. Checked Exception

   An exception that **must be captured** and handled gracefully within the application via the try/catch clause (can anticipate/recover from)

   E.g. user supplied filename for reading of nonexistent file:

   `java.io.FileNotFoundException`

2. Error (Unchecked):

   External to the application (cannot anticipate or recover from)

   E.g. opens file but system malfunction prevents the read operation

   `java.io.IOError`

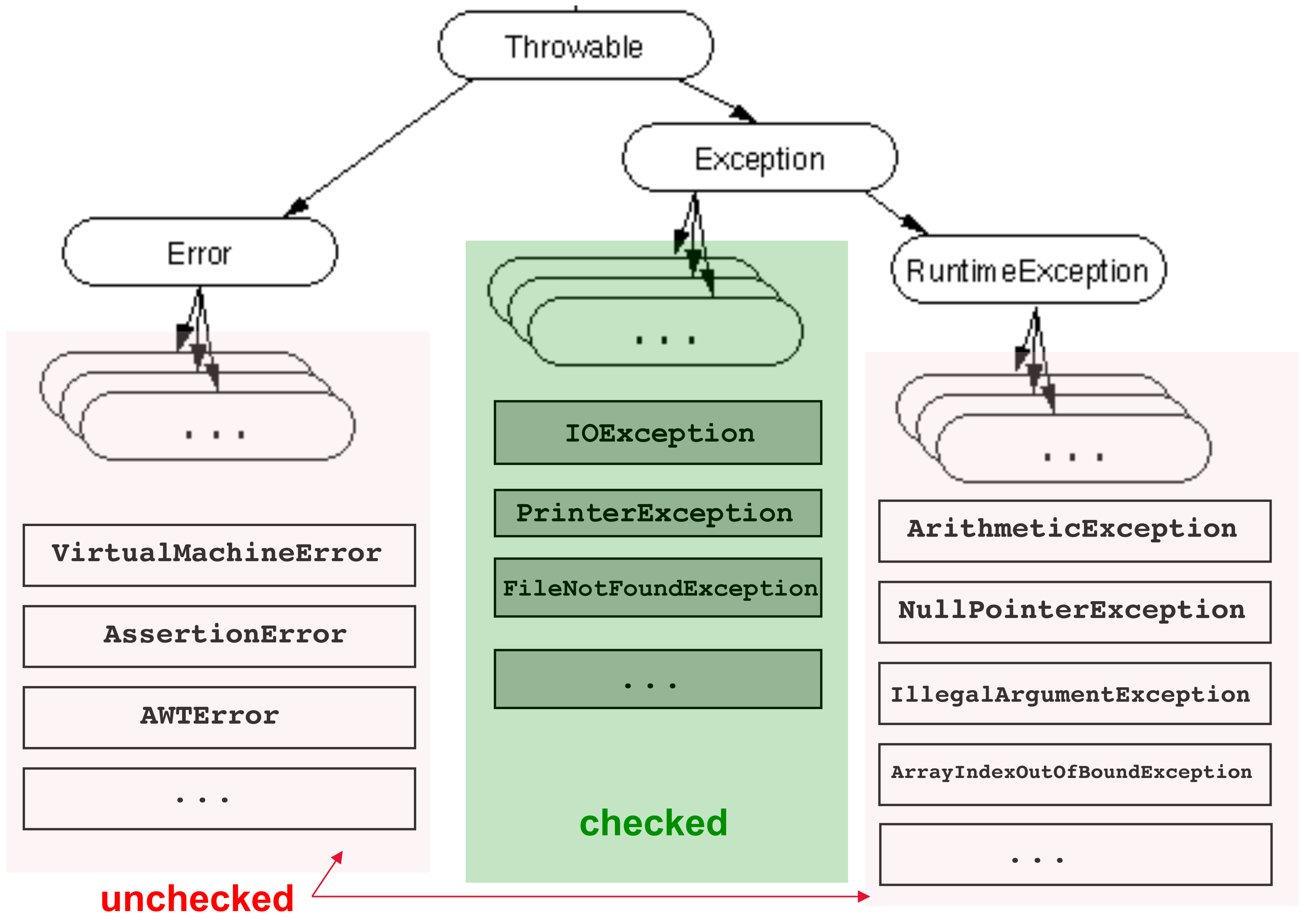
3. RuntimeException (Unchecked):

   Internal to the application (cannot anticipate or recover from)

   MOST COMMONLY

   i.e. program bugs, logic errors, or improper use of an API

   E.g. null as an argument for the constructor of a file object, causes `NullPointerException`


Throwable
Error
Exception
RuntimeException
. . .
. . .
. . .
VirtualMachineError
AssertionError
AWTError
. . .
IOException
PrinterException
FileNotFoundException
. . .
checked
ArithmeticException
NullPointerException
IllegalArgumentException
ArrayIndexOutOfBoundException
. . .
unchecked

- **Checked Exception:**
  - All Exceptions except *Error's* & *RuntimeException's*
  - Java ENFORCES these to be handled (at compile time)
    - **try { } catch { }** block around code that may cause exception

      OR
    - Use **throws** in method header (acknowledges that a particular exception may occur anywhere in the method)

- **Unchecked Exception:**
  - JAVA compiler does not ENFORCE handling
  - *RuntimeException* or *Error* family of objects

Try-catch is one way to handle expressions

- Must have both if there is a possibility of a checked exception being thrown
- Example: File I/O (input/output)

  → i.e. reading from/ writing to a file

  → reading (can use Scanner)

Previously.. in processing

- Simple text file read (in one method call):

colours.txt

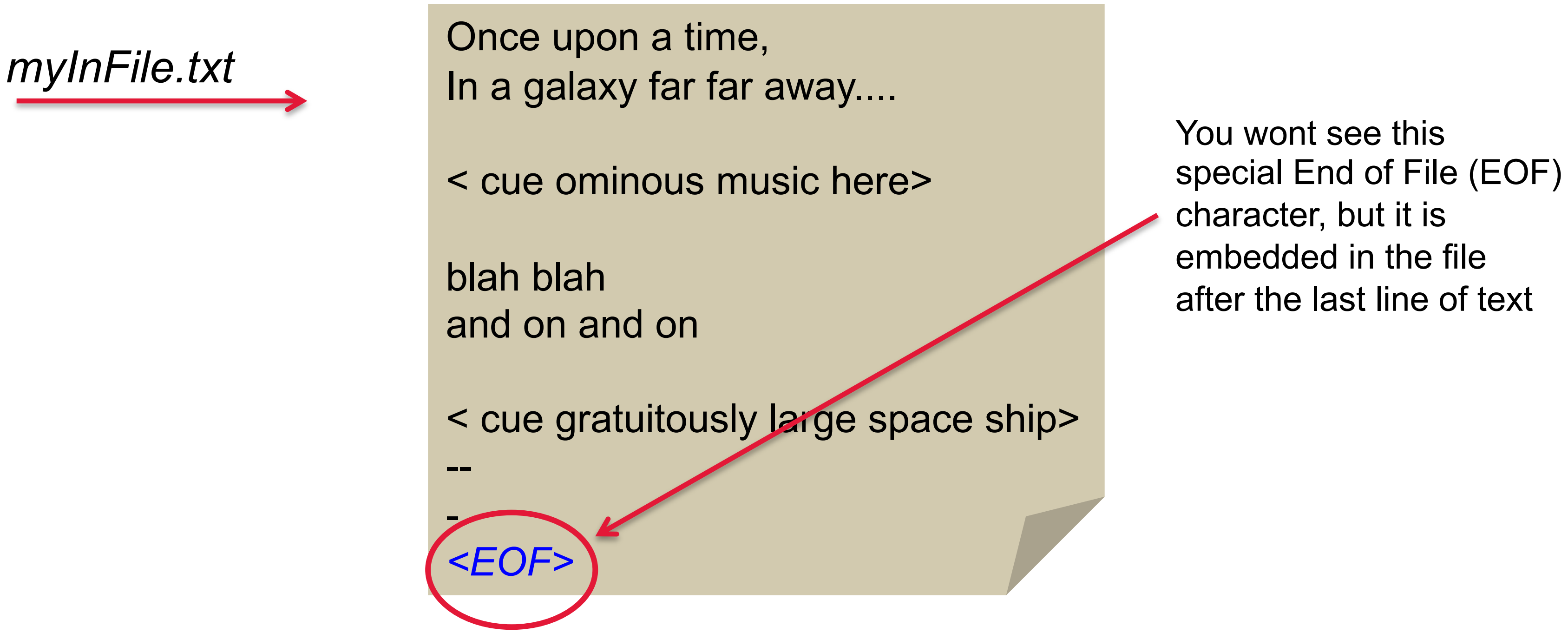
```
black 0 0 0
white 255 255 255

red 255 0 0
blue 0 0 255

green 0 255 0


grey 128 128 128

darkgrey 50 50 50
lightgrey 200 200 200


<EOF>
```

Can use a method directly:

```
Strings[] lines = loadStrings(filename);
```

filename (e.g. colours.txt) has to exist within the sketch folder

File reading

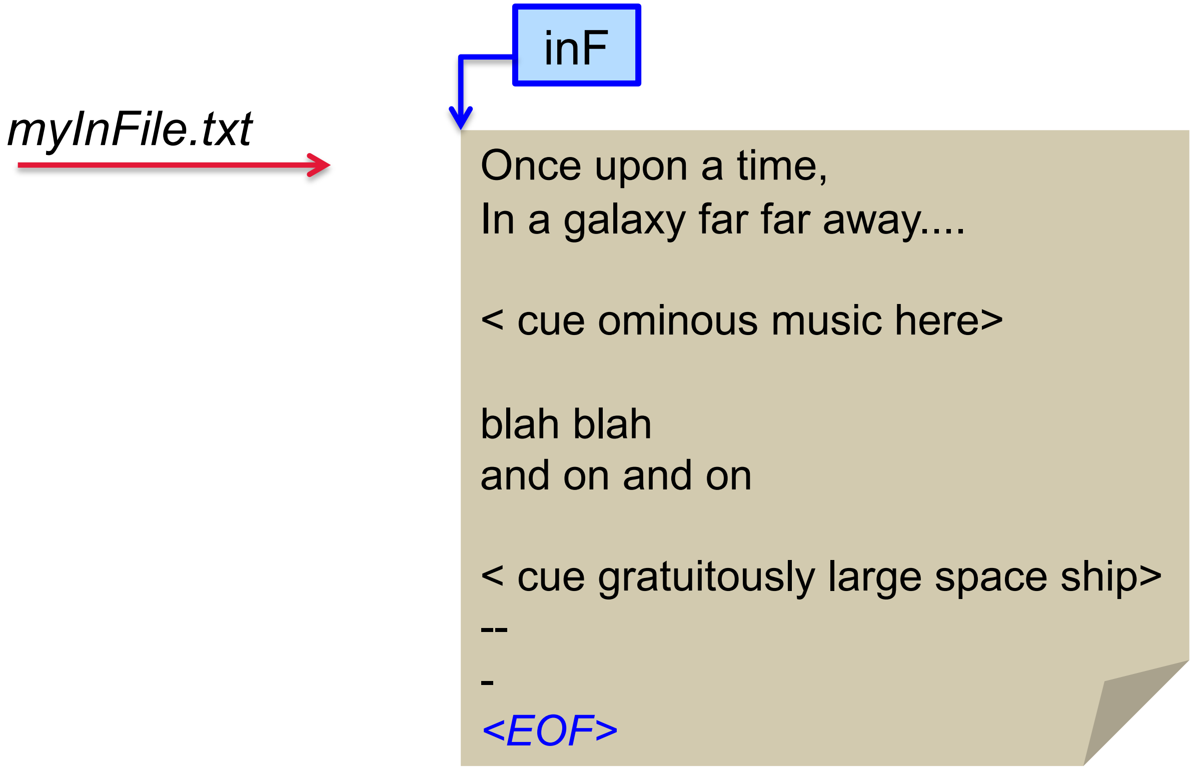

myInFile.txt
You wont see this special End of File (EOF) character, but it is embedded in the file after the last line of text
<EOF>

```
Once upon a time,
In a galaxy far far away....

< cue ominous music here>

blah blah
and on and on

< cue gratuitously large space ship>
--
-
<EOF>
```

Note: A Scanner object "scans" through the source it is connected to: i.e. when something has been read (using a next() or nextInt(), etc.) then the scanner moves forward to the next bit of input from that source

A **Scanner** is used together with a **File** object (which establishes a connection to a file)

File reading



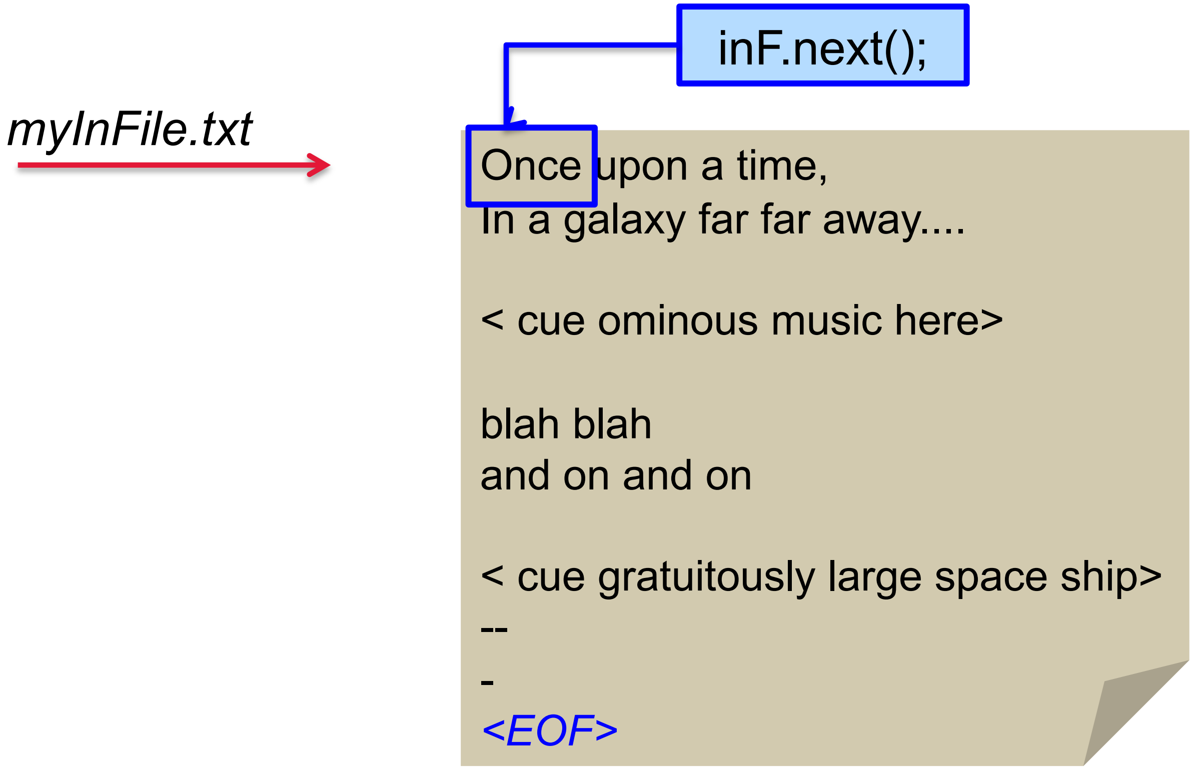
```
Once upon a time,
In a galaxy far far away....

< cue ominous music here>

blah blah
and on and on

< cue gratuitously large space ship>
--
-
<EOF>
```

You wont see this special End of File (EOF) character, but it is embedded in the file

```
// assume file is in current directory
File inFile = new File(“./myInFile.txt”);
Scanner inF = new Scanner(inFile);
```

File reading

will read first String
(up to next whitespace)



Once upon a time,
In a galaxy far far away....

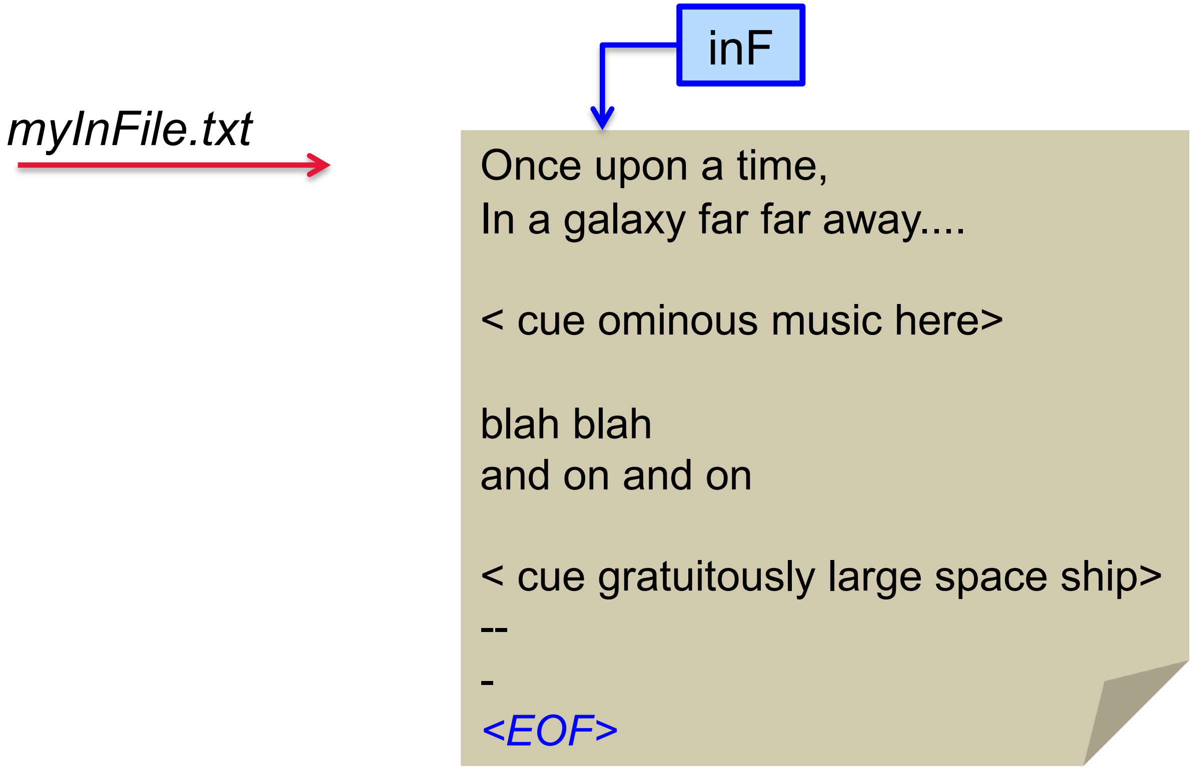
< cue ominous music here>

blah blah
and on and on

< cue gratuitously large space ship>
--
-
<EOF>

```
// assume file is in current directory
File inFile = new File("./myInFile.txt");
Scanner inF = new Scanner(inFile);
inF.next();
```

File reading

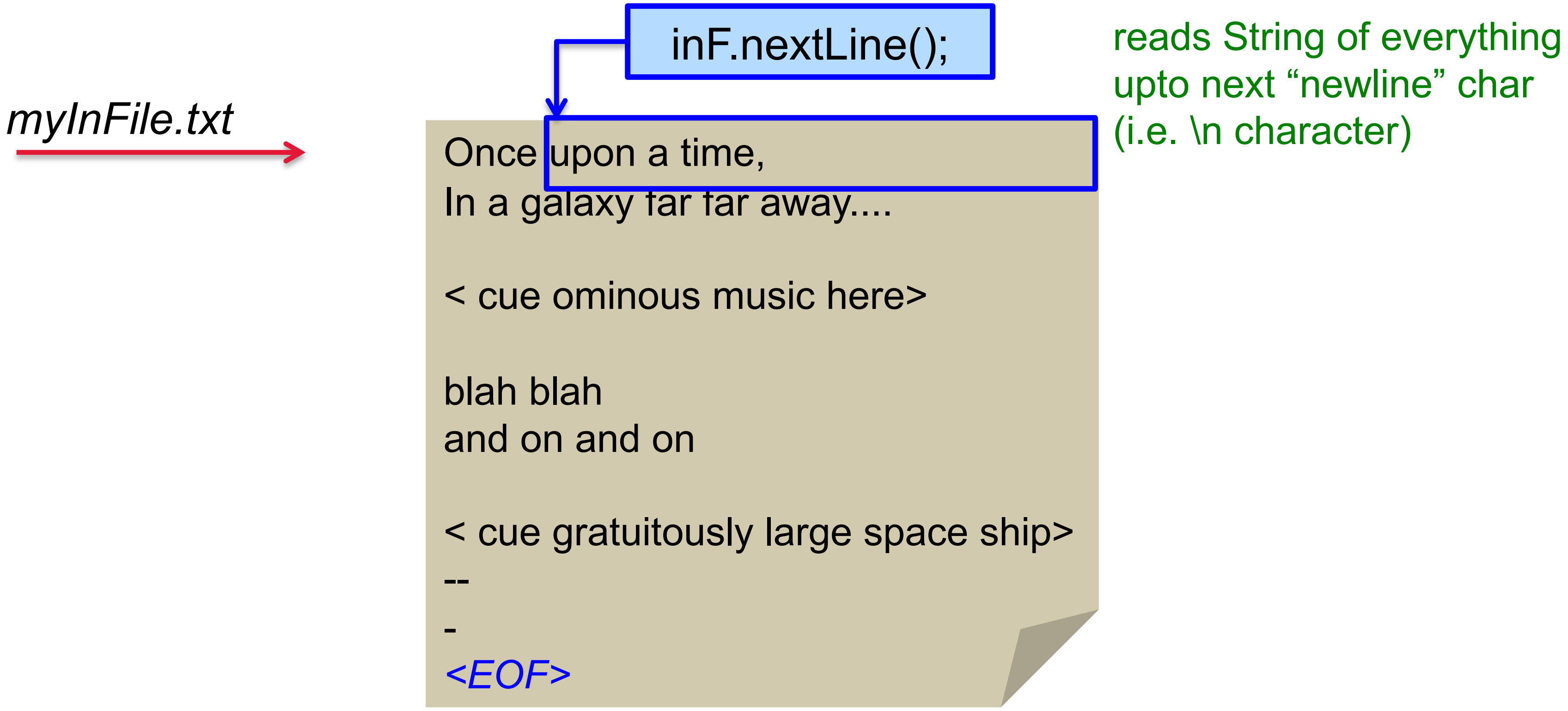


```
Once upon a time,
In a galaxy far far away....

< cue ominous music here>

blah blah
and on and on

< cue gratuitously large space ship>
--
-
<EOF>
```

afterward, Scanner (inF) will be positioned here

```
// assume file is in current directory
File inFile = new File(“./myInFile.txt”);
Scanner inF = new Scanner(inFile);
inF.next();
// after
```

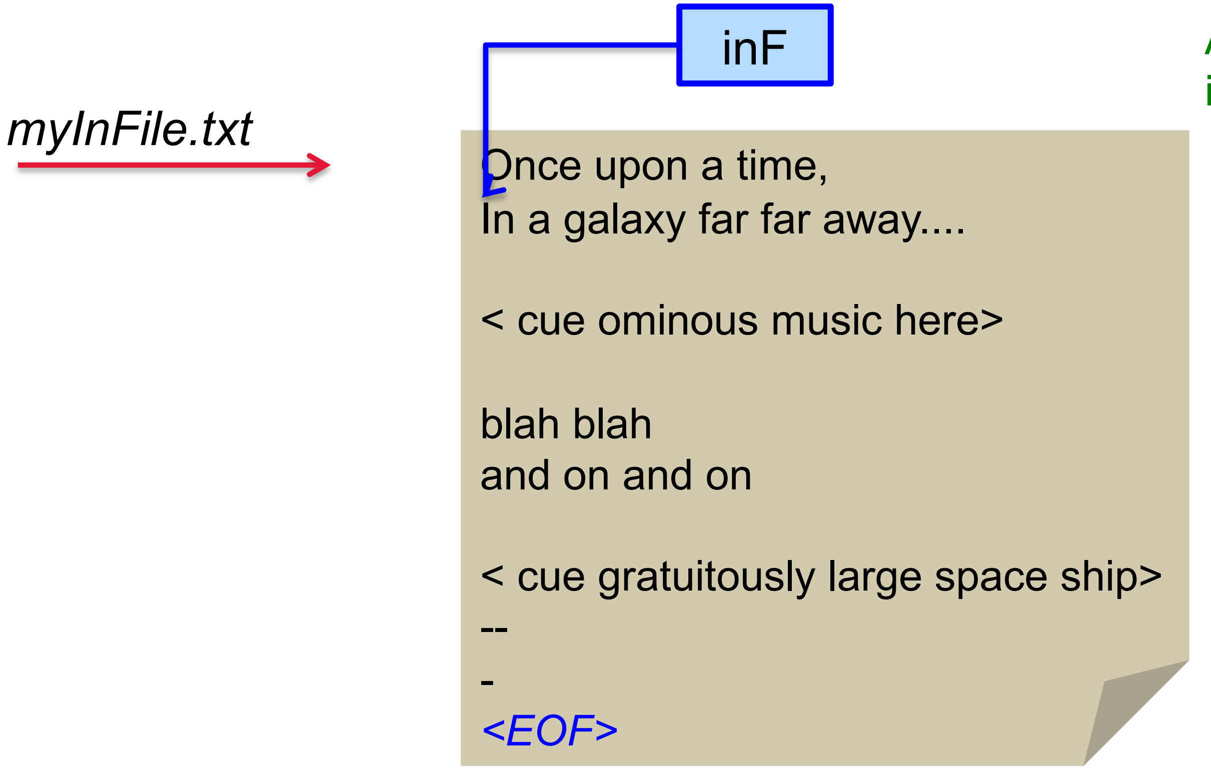
File reading


myInFile.txt
inF.nextLine();
reads String of everything upto next "newline" char (i.e. \n character)

```
Once upon a time,
In a galaxy far far away....

< cue ominous music here>

blah blah
and on and on

< cue gratuitously large space ship>
--
-
<EOF>
```

```
// assume file is in current directory
File inFile = new File("./myInFile.txt");
Scanner inF = new Scanner(inFile);
inF.next();
inF.nextLine();
```

File reading



Once upon a time,
In a galaxy far far away....

< cue ominous music here>

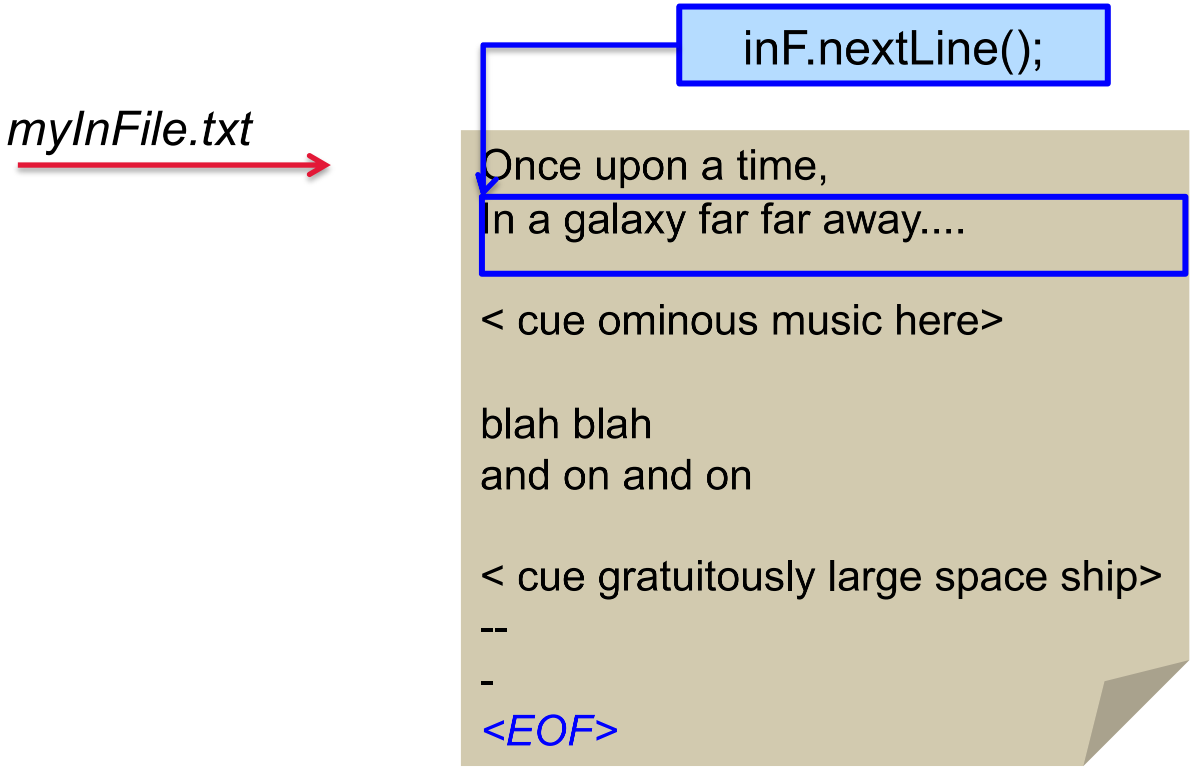
blah blah
and on and on

< cue gratuitously large space ship>
--
-
<EOF>

After reading whole line, inF now positioned here

```
// assume file is in current directory
File inFile = new File(“./myInFile.txt”);
Scanner inF = new Scanner(inFile);
inF.next();
inF.nextLine();
```

File reading



Once upon a time,
In a galaxy far far away....

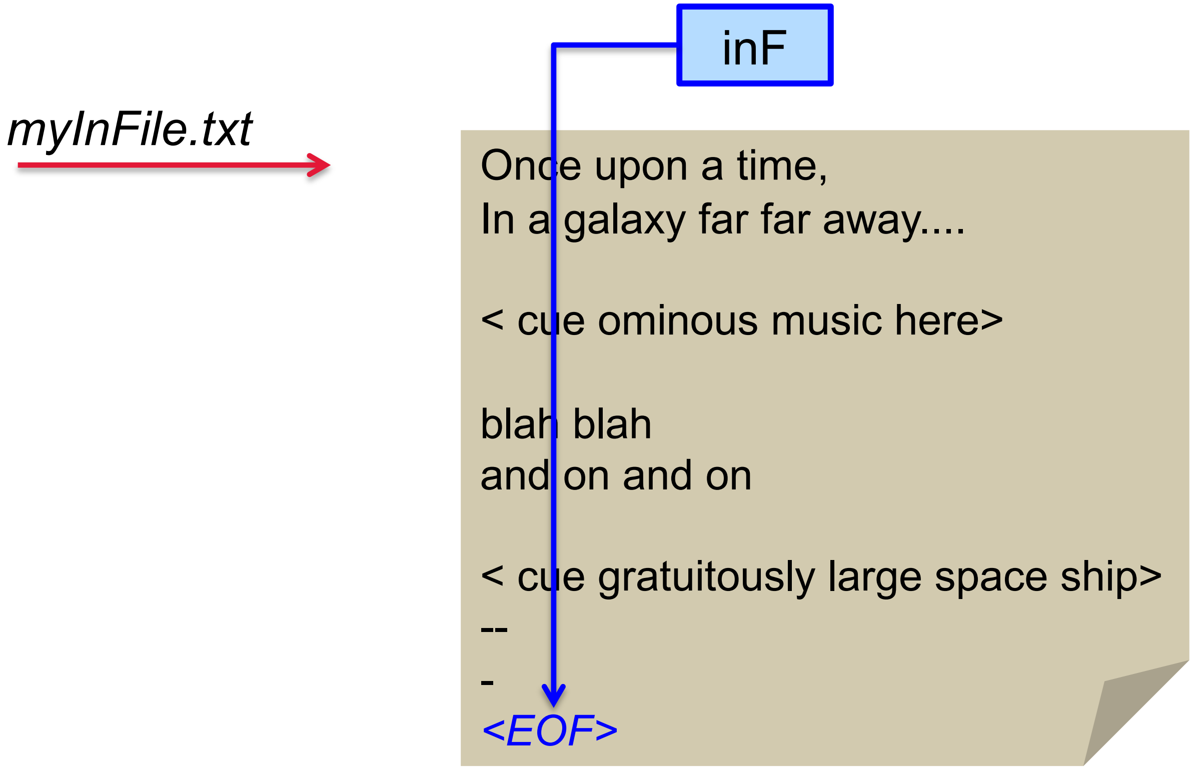
< cue ominous music here>

blah blah
and on and on

< cue gratuitously large space ship>
--
-
<EOF>

```
// assume file is in current directory
File inFile = new File(“./myInFile.txt”);
Scanner inF = new Scanner(inFile);
inF.next();
inF.nextLine();
inF.nextLine();
```

File reading



Once upon a time,
In a galaxy far far away....

< cue ominous music here>

blah blah
and on and on

< cue gratuitously large space ship>
--
-
<EOF>

While loop allows us to continue reading until we get to the <EOF>

```
…
while (inF.hasNextLine()) {
        // read next something
}
```

If inF positioned at <EOF>, then hasNextLine() returns false

File output

- Similar to console output; instead of

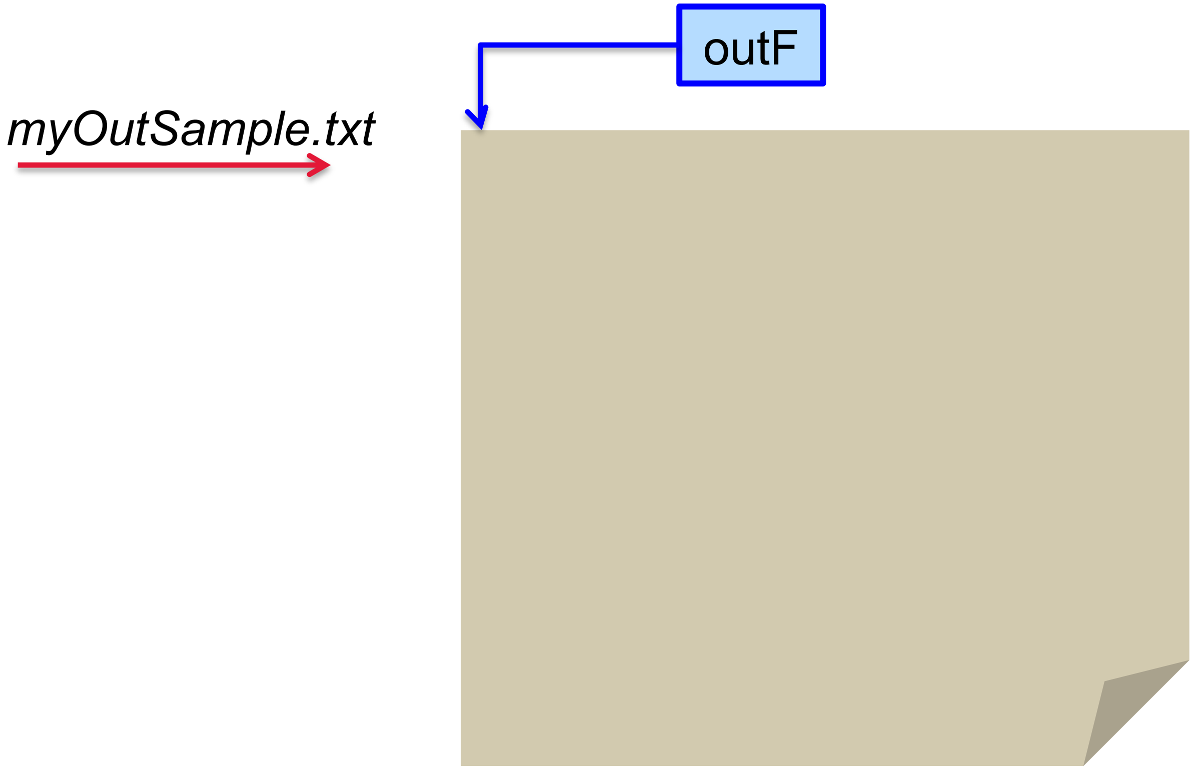
```
PrintStream out = System.out;
```

- Use

```
PrintStream outF = new PrintStream(filename);
```

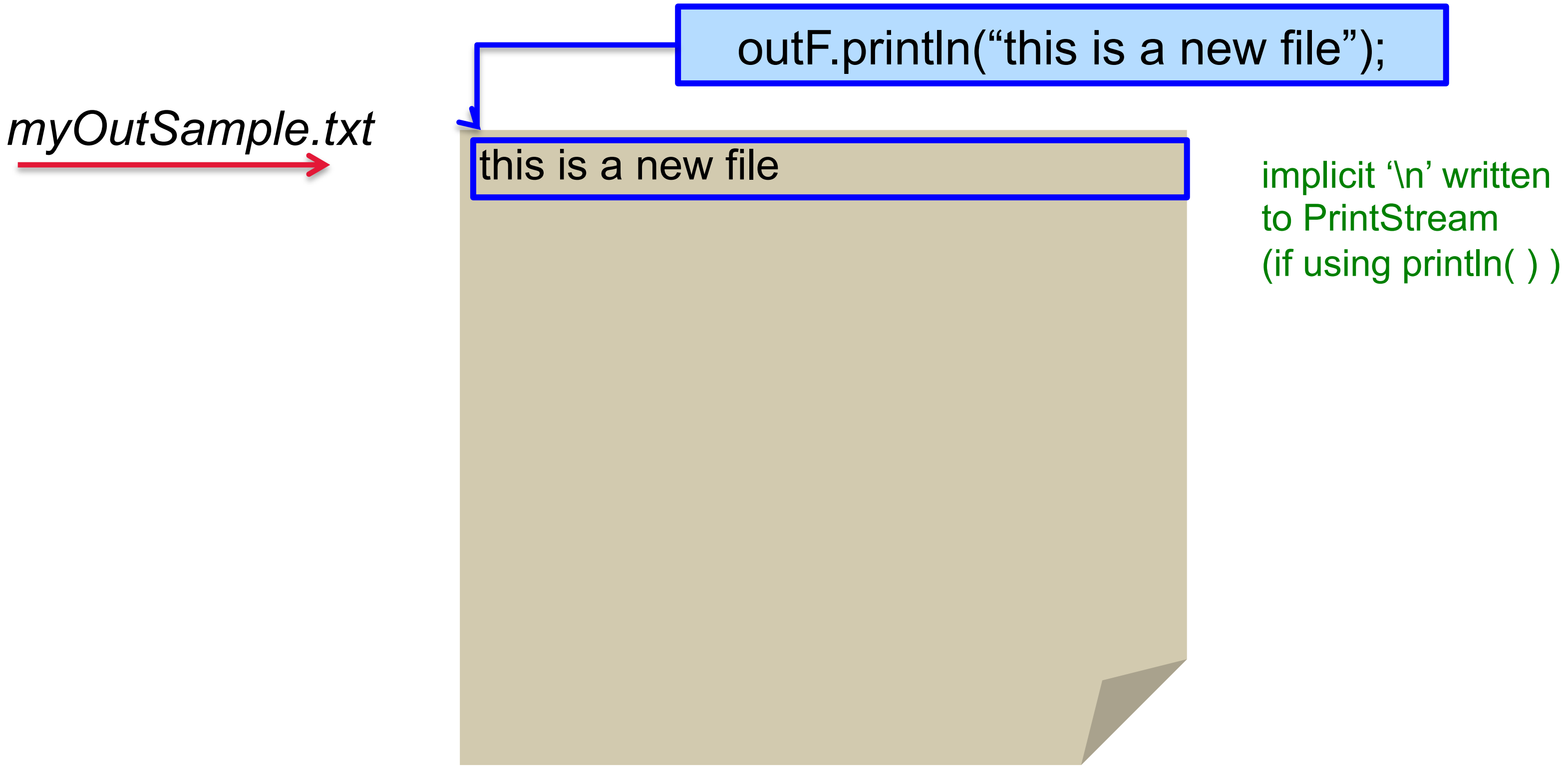
- Then use **`print()`**, **`println()`**, or **`printf()`**, as before
- main method requires throws IOException
- Use **`close()`** when done

File Writing



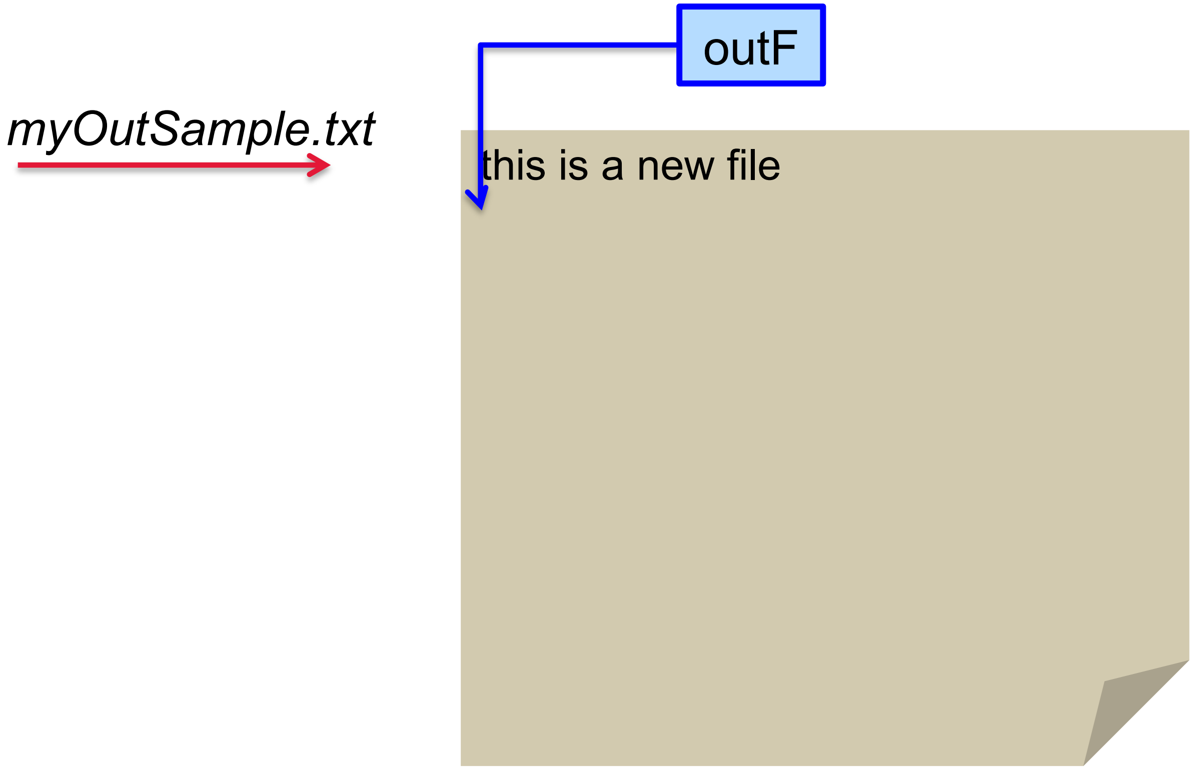
```
// assume file is in current directory
File outFile = new File(“./myOutSample.txt”);
PrintStream outF = new PrintStream(outFile);
```

File Writing


outF.println("this is a new file");
myOutSample.txt
this is a new file
implicit '\n' written to PrintStream (if using println( ))

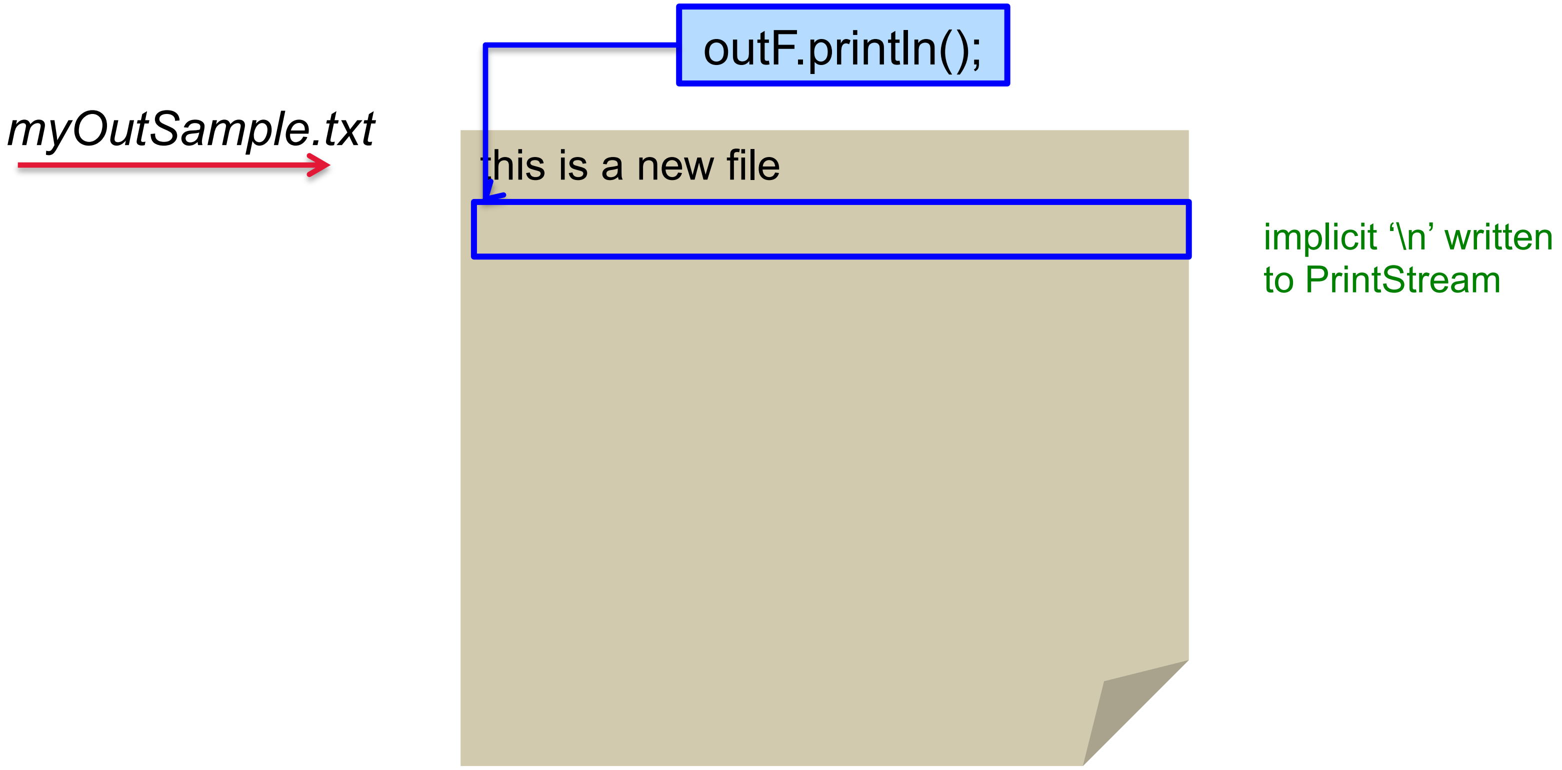
```
// assume file is in current directory
File outFile = new File("./myOutSample.txt");
PrintStream outF = new PrintStream(outFile);
outF.println("this is a new file");
```

File Writing



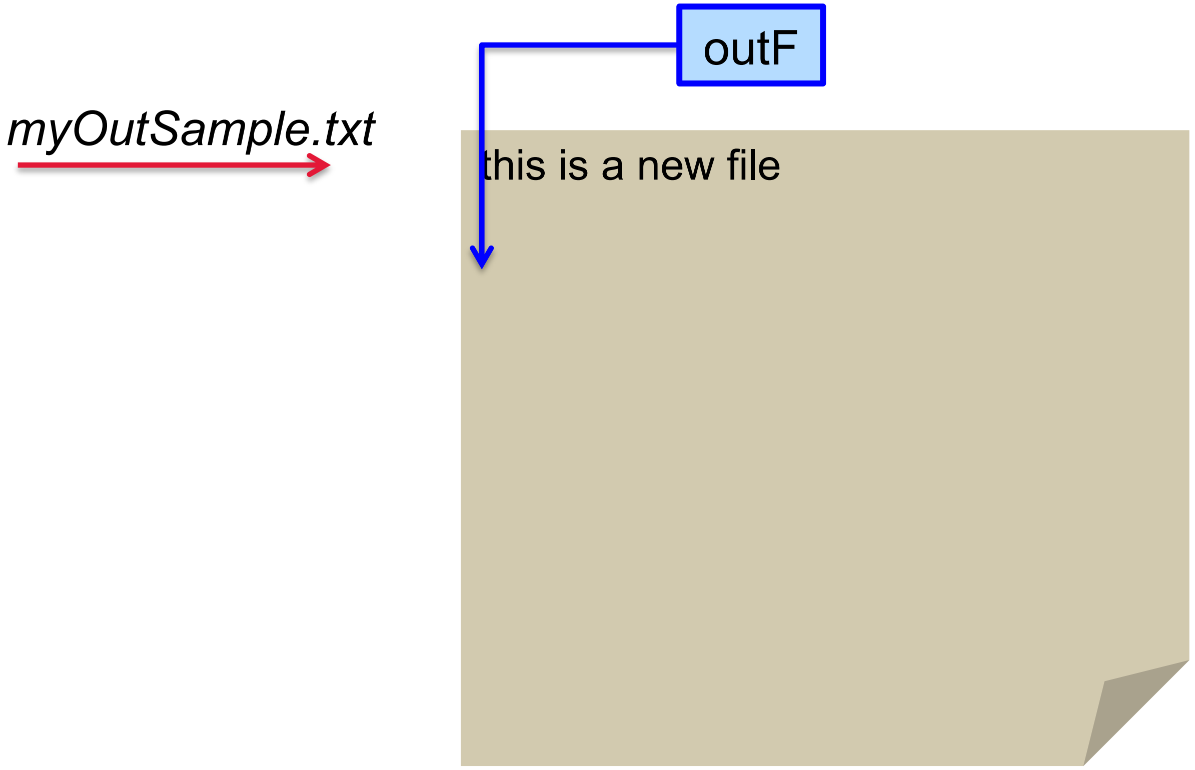
```
// assume file is in current directory
File outFile = new File("./myOutSample.txt");
PrintStream outF = new PrintStream(outFile);
outF.println("this is a new file");
```

File Writing



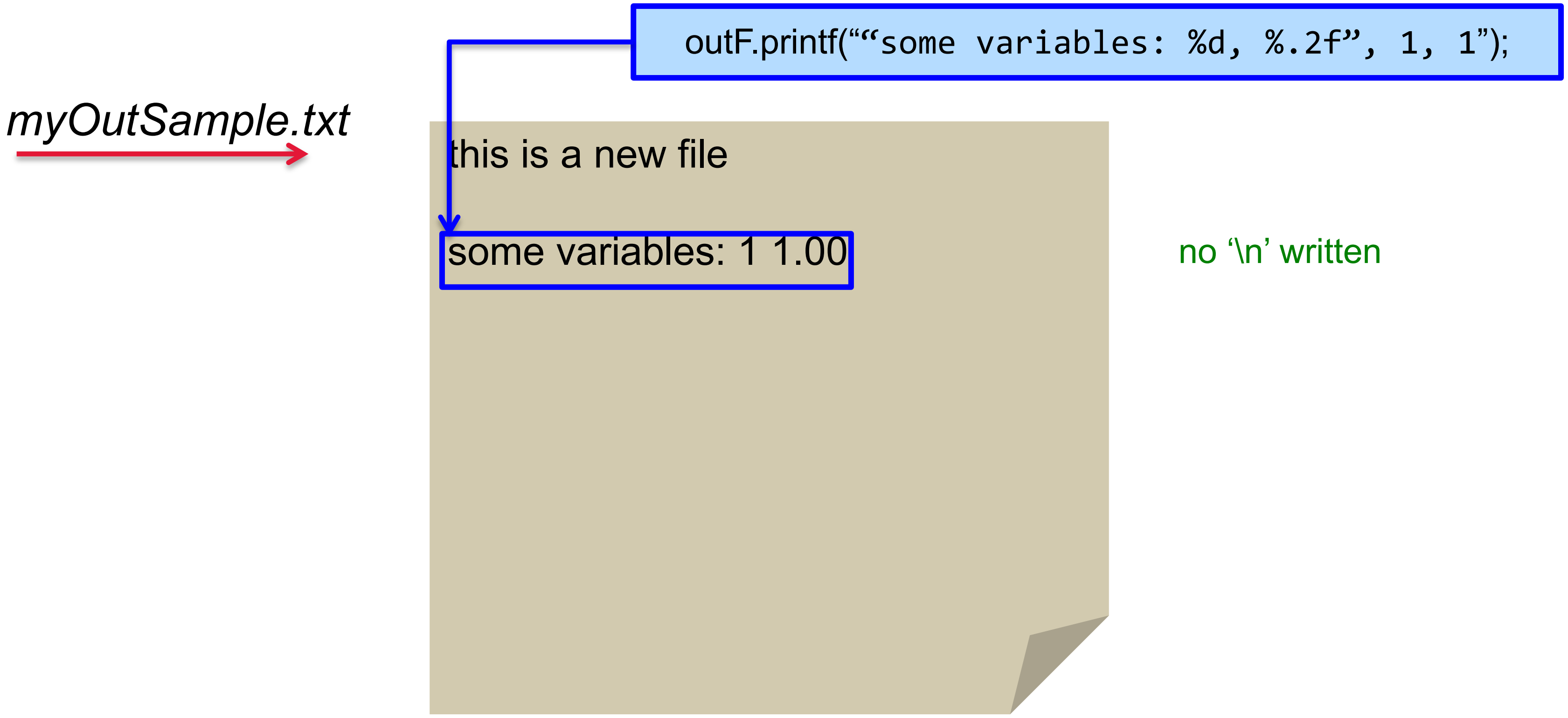
```
// assume file is in current directory
File outFile = new File(“./myOutSample.txt”);
PrintStream outF = new PrintStream(outFile);
outF.println(“this is a new file”);
outF.println();
```

File Writing



```
// assume file is in current directory
File outFile = new File(“./myOutSample.txt”);
PrintStream outF = new PrintStream(outFile);
outF.println(“this is a new file”);
outF.println();
```

File Writing



outF.printf(““some variables: %d, %.2f”, 1, 1”);

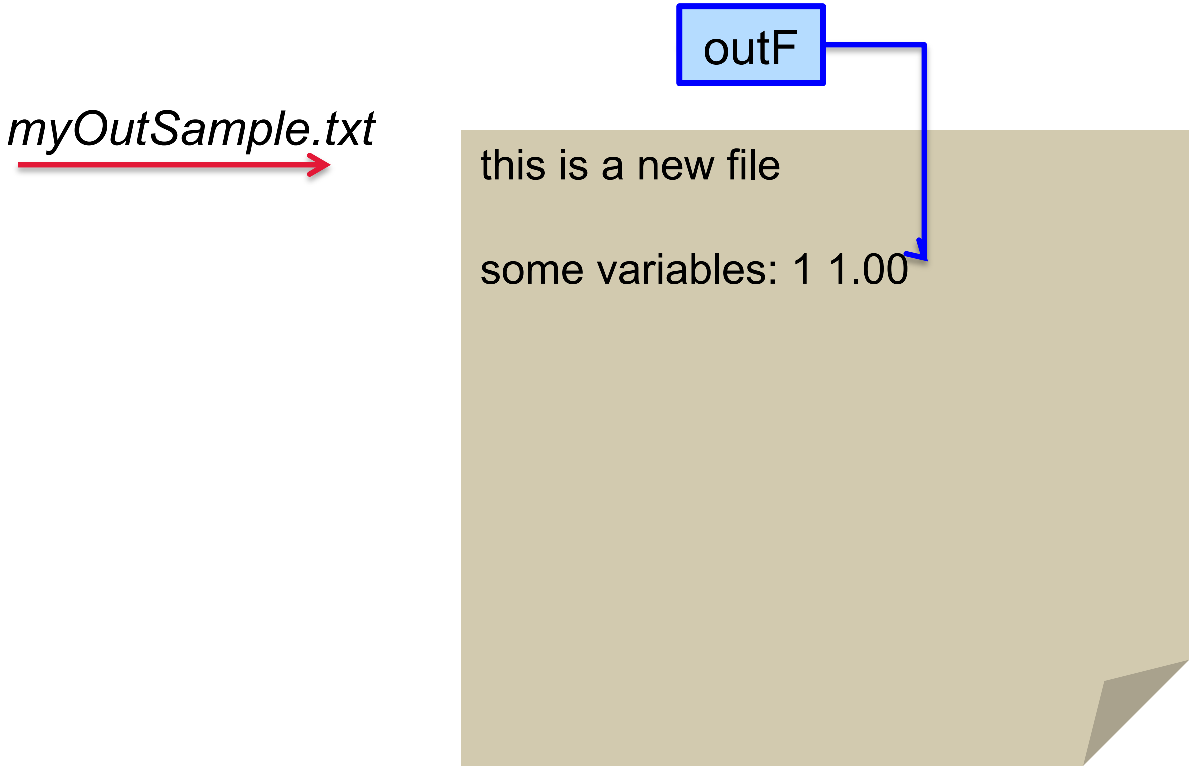
myOutSample.txt

```
this is a new file

some variables: 1 1.00
```

no ‘\n’ written

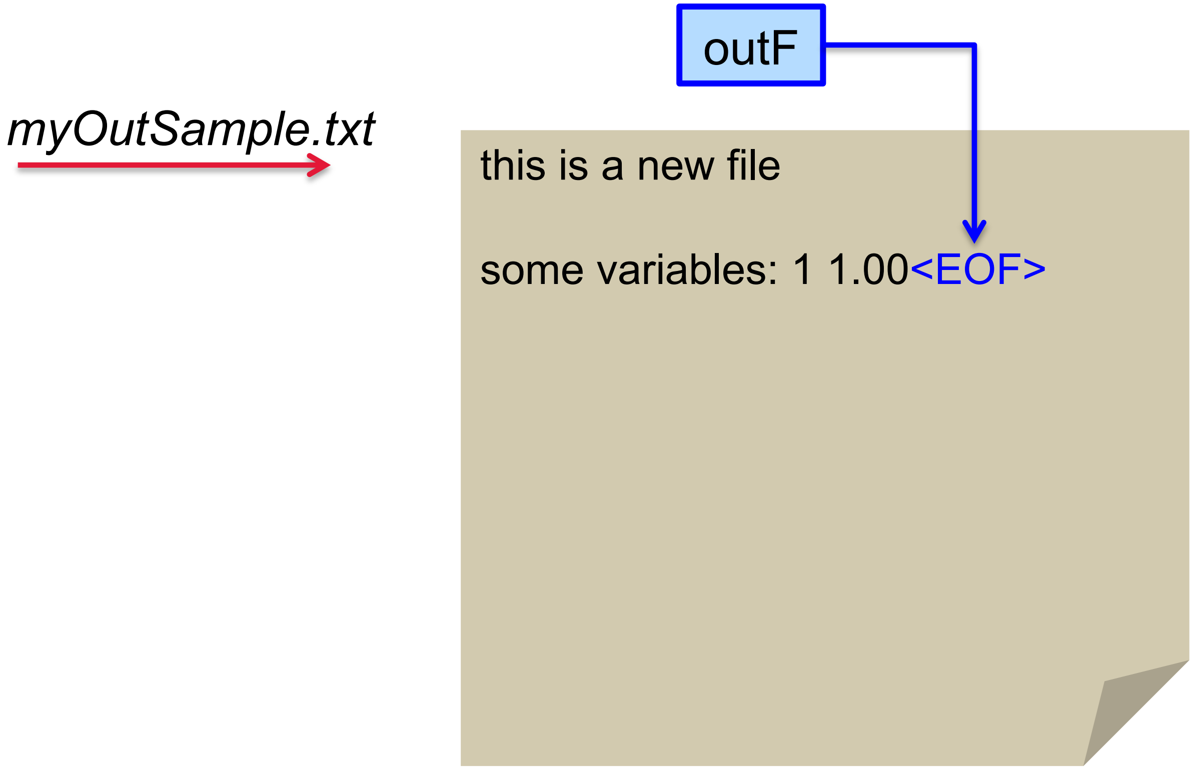
```
// assume file is in current directory
File outFile = new File(“./myOutSample.txt”);
PrintStream outF = new PrintStream(outFile);
outF.println(“this is a new file”);
outF.println();
outF.printf(“some variables: %d, %.2f”, 1, 1);
```

File Writing



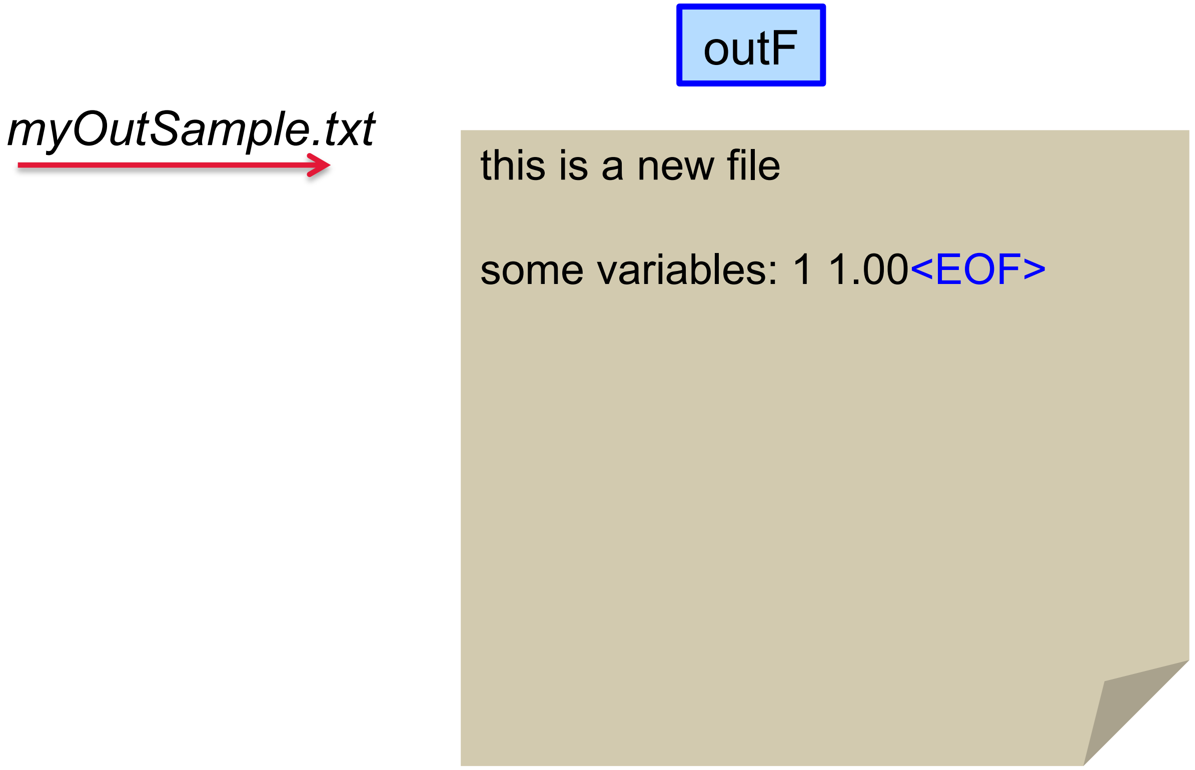
```
// assume file is in current directory
File outFile = new File("./myOutSample.txt");
PrintStream outF = new PrintStream(outFile);
outF.println("this is a new file");
outF.println();
outF.printf("some variables: %d, %.2f", 1, 1);
```

File Writing



```
File outFile = new File(“./myOutSample.txt”);
PrintStream outF = new PrintStream(outFile);
outF.println(“this is a new file”);
outF.println();
outF.printf(“some variables: %d, %.2f”, 1, 1);
outF.close();
```

File Writing



```
this is a new file

some variables: 1 1.00<EOF>
```

```
File outFile = new File(“./myOutSample.txt”);
PrintStream outF = new PrintStream(outFile);
outF.println(“this is a new file”);
outF.println();
outF.printf(“some variables: %d, %.2f”, 1, 1);
outF.close();
```

Example 5: reading from a file (exceptions)

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

public class FileIOError {

     public static void main(String[] args) {

          Scanner in = new Scanner(System.in);
          File inFile = new File("./sample.txt");

          // do an echo of input file (i.e. read all lines and output them to screen)
          Scanner inF = new Scanner(inFile);
          String oneLineText;

          System.out.println("Contents of file:");
          System.out.println("****************************");

          while (inF.hasNextLine()) {
               oneLineText = inF.nextLine();
               System.out.println(oneLineText);
          }

          inF.close();  // close the file after reading!!

     }

}
```

Example 5: where & what exception types can occur?

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

public class FileIOError {

     public static void main(String[] args) {

          Scanner in = new Scanner(System.in);
          File inFile = new File("./sample.txt");

          // do an echo of input file (i.e. read all lines and output them to screen)
          Scanner inF = new Scanner(inFile);
          String oneLineText;

          System.out.println("Contents of file:");
          System.out.println("****************************");

          while (inF.hasNextLine()) {
               oneLineText = inF.nextLine();
               System.out.println(oneLineText);
          }

          inF.close();  // close the file after reading!!

     }
}
```

# wont compile?!

Example 5: where & what exception types can occur?

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

public class FileIOError {

     public static void main(String[] args) {

          Scanner in = new Scanner(System.in);
          File inFile = new File("./sample.txt");

          // do an echo of input file (i.e. read all lines and output them to screen)
          Scanner inF = new Scanner(inFile);
          String oneLineText;

          System.out.println("Contents of file:");
          System.out.println("**************************

          while (inF.hasNextLine()) {
               oneLineText = inF.nextLine();
               System.out.println(oneLineText);
          }

          inF.close();  // close the file after reading!!

     }

}
```

NullPointerException

FileNotFoundException

IllegalStateException

NoSuchElementException / IllegalStateException

IllegalStateException

Example 5a: insert try/catch block

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class FileIOErrorHandled1 {

     public static void main(String[] args) {
          try {
               Scanner in = new Scanner(System.in);
               File inFile = new File("./sample.txt");

               // do an echo of input file (i.e. read all lines and output them to screen)
               Scanner inF = new Scanner(inFile);
               String oneLineText;
               System.out.println("Contents of file:");
               System.out.println("****************************");

               while (inF.hasNextLine()) {
                    oneLineText = inF.nextLine();
                    System.out.println(oneLineText);
               }
               inF.close();  // close the file after reading!!
          }
          catch (FileNotFoundException e) {
               // handle it
          }
     }
}
```

Example 5a: insert try/catch block

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.NoSuchElementException;
import java.util.Scanner;

public class FileIOErrorHandled1 {

     public static void main(String[] args) {
          try {
               Scanner in = new Scanner(System.in);
               File inFile = new File("./sample.txt");

               // code to read file not shown

               inF.close();  // close the file after reading!!
          }
          catch (FileNotFoundException e) {
               // handle it
          }
          catch (NullPointerException e) {
               // handle it
          }
          catch (NoSuchElementException e) {
               // handle it
          }

     }
}
```

Example 5b: indicate a method may throw an exception

```
import java.io.File;
import java.util.Scanner;
import java.io.FileNotFoundException;

public class FileIOErrorHandled2 {

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

               Scanner in = new Scanner(System.in);
               File inFile = new File("./sample.txt");

               // do an echo of input file (i.e. read all lines and output them to screen)
               Scanner inF = new Scanner(inFile);
               String oneLineText;
               System.out.println("Contents of file:");
               System.out.println("****************************");

               while (inF.hasNextLine()) {
                    oneLineText = inF.nextLine();
                    System.out.println(oneLineText);
               }

               inF.close();  // close the file after reading!!


     }
}
```

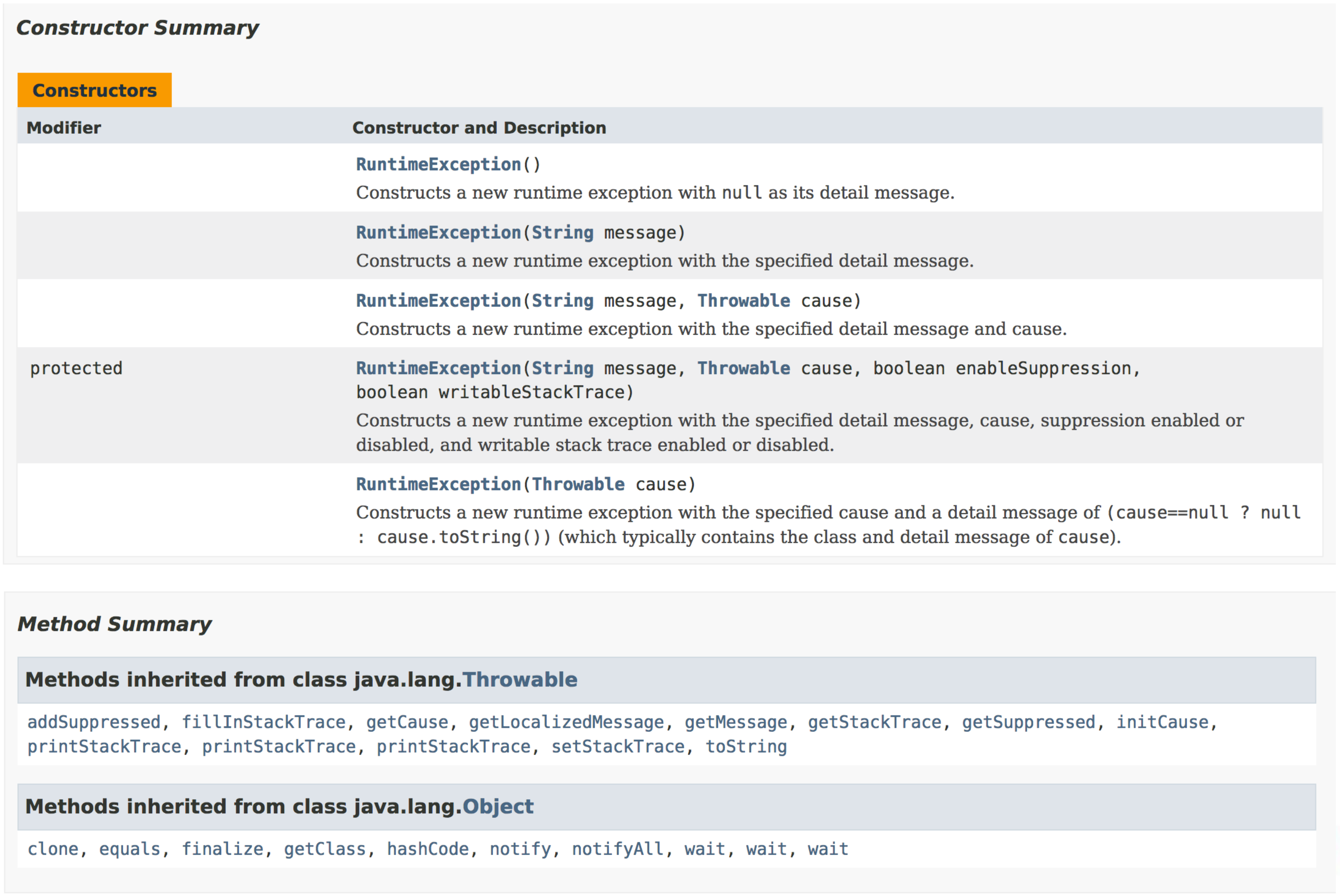
Assumption is that this will be handled higher up the call stack

Are exceptions always automatically thrown?

- No!
- Exceptions are objects after all …
  - **Thus, we can instantiate and throw them at will !!**

- Again, to throw an exception:
  - Instantiate and use the “throw” keyword

RuntimeException



Constructor Summary

Constructors

Modifier	**Constructor and Description**
	`RuntimeException()` Constructs a new runtime exception with `null` as its detail message.
	`RuntimeException(String message)` Constructs a new runtime exception with the specified detail message.
	`RuntimeException(String message, Throwable cause)` Constructs a new runtime exception with the specified detail message and cause.
`protected`	`RuntimeException(String message, Throwable cause, boolean enableSuppression, boolean writableStackTrace)` Constructs a new runtime exception with the specified detail message, cause, suppression enabled or disabled, and writable stack trace enabled or disabled.
	`RuntimeException(Throwable cause)` Constructs a new runtime exception with the specified cause and a detail message of `(cause==null ? null : cause.toString())` (which typically contains the class and detail message of `cause`).

Method Summary

Methods inherited from class java.lang.Throwable

`addSuppressed, fillInStackTrace, getCause, getLocalizedMessage, getMessage, getStackTrace, getSuppressed, initCause, printStackTrace, printStackTrace, printStackTrace, setStackTrace, toString`

Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait`

Example 6a

```
import java.util.Scanner;

public class ThrowAnException1{

    public static void main(String[] args) {

        System.out.println(”Here I will create and throw an exception.\n");

        RuntimeException myException = new RuntimeException();

        throw myException;


    }
}
```

Example 6b

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

public class ThrowAnException2{
    public static void main(String[] args) {

        System.out.println(”Here I will create and throw an exception.\n");

        FileNotFoundException myFNFE = new FileNotFoundException();

        throw myFNFE;


    }
}
```

# wont compile?!
=> checked Exception object,
therefore must be handled

Example 6c

```
import java.util.Scanner;

public class ThrowAnException3{
    public static void main(String[] args) {

        try {
            System.out.println(”Here I will create and throw an
            exception.\n");
            RuntimeException myException = new RuntimeException();
            throw myException;

        }
        catch (RuntimeException e) {
            System.out.println(“And here I have caught the exception.\n”);
        }



    }
}
```

Why create our own?

- Perhaps we want to address an unusual condition if it occurs. For example, in the (x/y) example.

- We consider that x and y should never be negative
  - this is an arbitrary reason to throw an exception..
  - Could be handled using validity test
  - but as an example.. Lets create our own Exception if this occurs

Example 7

lets say we don't want left or right to be negative ??

```
import java.util.Scanner;

public class NonNegativeFraction{
    public static void main(String[] args) {

        System.out.println("Enter a fraction (x/y) ");
        System.out.println("and I will give you the quotient");
        Scanner in = new Scanner(System.in);
        String str = in.nextLine();

        int slash = str.indexOf("/");
        String left = str.substring(0, slash);
        String right = str.substring(slash + 1);

        int numer = Integer.parseint(left);
        int denom = Integer.parseint(right);
        if (numer<0 || denom<0 )
                throw new IllegalArgumentException();
        int quotient = numer/denom;
        System.out.println("Quotient = " + quotient);

    }
}
```

NOTES on throwing a manual Exception

- Remember, if it is a checked Exception (we must handle), if not, then it is not mandatory to handle

- When handling, we have full access to the attributes or methods of the Exception object.
- Note that when instantiating, we can pass a message to the object (accessible later using getMessage())

Finally

- Exceptions are a powerful construct which can be employed to great benefit

- Exceptions are part of many client-implementer contracts
  - exceptions are often found within most API’s
  - exceptions notify of potential issues, but are left to the person using the class to decide how best to handle them

- Exceptions illustrate a fundamental principle of interactive systems (the throw/catch mechanism is quite similar to the way interruptions/events are handled) → e.g. mouseclicks /keypresses etc

- We will see more use of Exceptions when designing our own classes

Further Reading

The Java™ Tutorials, Essential Classes, Lesson: Exceptions

- https://docs.oracle.com/javase/tutorial/essential/exceptions/index.html

Java Programming, Wikibooks, Section "Exceptions"

- https://en.wikibooks.org/wiki/Java_Programming/Exceptions

*** some subtopics in the above will be covered in the next lecture and some later in this course as we learn more about classes*