

COMP 1020 - Strings, exceptions, and files

UNIT 4

More String methods

- We've seen:
 - `length()`, `charAt()`, `equals()`, `equalsIgnoreCase()`
- Some other useful instance methods:
 - `String toLowerCase()` - letters become lower case
`String s = "Hello, Person."`
`s.toLowerCase()` would give `"hello, person."`
 - `String toUpperCase()` – letters become upper case
`s.toUpperCase()` would give `"HELLO, PERSON."`
 - `String trim()` – removes *leading/trailing* whitespace (including tabs and newlines)

More String methods

- Remember: these methods **don't** change the original
 - That's impossible. String objects are "**immutable**".
 - They create a **new String** and return a reference to it

```
System.out.println(s.toLowerCase( )); //s not changed  
s = s.toLowerCase( ); //changes s
```

Another String method

- `String1.compareTo(String2)`
- Gives
 - Some negative value if String1 is “less” than String2
 - A zero if they are the same
 - Some positive value if String1 is “greater” than String2

Another String method

- `String1.compareTo(String2)`
- Essentially, it tells you which is first “in alphabetical order”
 - “less” or “greater” is determined by the numeric codes (ASCII or Unicode) of the characters in that position
 - Letter codes are always in alphabetical order
 - Be careful: **uppercase is always less than lowercase**

Another String method

- `String1.compareTo(String2)`
- Essentially, it tells you which is first “in alphabetical order”
 - “less” or “greater” is determined by the numeric codes (ASCII or Unicode) of the characters in that position
 - Letter codes are always in alphabetical order
 - Be careful: **uppercase is always less than lowercase**

THE fundamental String methods

- Concatenation: + (we've already seen that)
- `String1.indexOf(String2)`
 - Finds the position of the first appearance of `String2` within `String1`
 - If not there at all, -1 is the result

THE fundamental String methods

- `String1.substring(start, end)`
 - Takes **two int parameters**
 - **start** can be considered as
 - The position of the first character you want, or
 - The number of characters you want to discard from the front
 - **end** is the position of the first character you DON'T want (end position is excluded)
- So it gives you from positions **start** to **end-1**
- The result is a String (the selected piece of `String1`)
- **end** can be omitted – then it takes everything to the end

Example

- Check to see if a String is a palindrome
 - Reads the same forward and backward
 - We want to do this while ignoring non-letters, leading/trailing space, and case
- Palindromes:
 - "detartrated"
 - "redivider"
 - "If I had a Hi-Fi"
 - " Was it a car or a cat I saw? "
 - "Able was I ere I saw Elba."

Example

- An email address has the form:
name@part1.part2.part3...
- You might want to break it up into its separate parts
 - name, part1, part2, part3, etc. to different strings
- Same type of situation:
 - A date may be yy/mm/dd
 - A big number may be 234,917,342
 - A file name may be H:\Documents\folder\stuff.txt
 - There are many things that might work this way

Example

- There is an easy way to do this
- `String[] split(String)` will break up a `String` into an array of `Strings` at the positions of the given substring
- `"Test 12 34".split(" ")` gives `{"Test", "12", "34"}`
- `.split("\\s+")` will split on any whitespace of any kind
- The parameter is actually a “regular expression”, in the form of a `String`. Beware!

Exceptions

- Exceptions in Java represent problems that arise during execution, which a program might want to deal with (manage / handle)
- Exceptions are objects
- There are multiple specific types of Exceptions (they actually form a hierarchy of objects)

Exceptions

- Whenever anything goes wrong in Java, an Exception is generated:
 - 5/0 gives an **ArithmeticException**
 - array[-1] = 0 gives an **ArrayIndexOutOfBoundsException**
 - Integer.parseInt("Not a number") gives a **NumberFormatException**

Exceptions

- Remember these are **objects**:
 - They contain information about **what went wrong** and **where in the program it went wrong**
 - They have **methods** you can use to find out this information
 - They're created automatically when problems occur (you can also define your own! → later)
- You can even define variables of these types:
`ArithmeticException ae = new ArithmeticException();`

Exception methods

- Every Exception object has these methods:
 - String getMessage()
 - Find out more about what happened
 - void printStackTrace()
 - Print out where the program was at the time this Exception object was created
 - String toString()
 - The name of the exception and its message

Exception methods

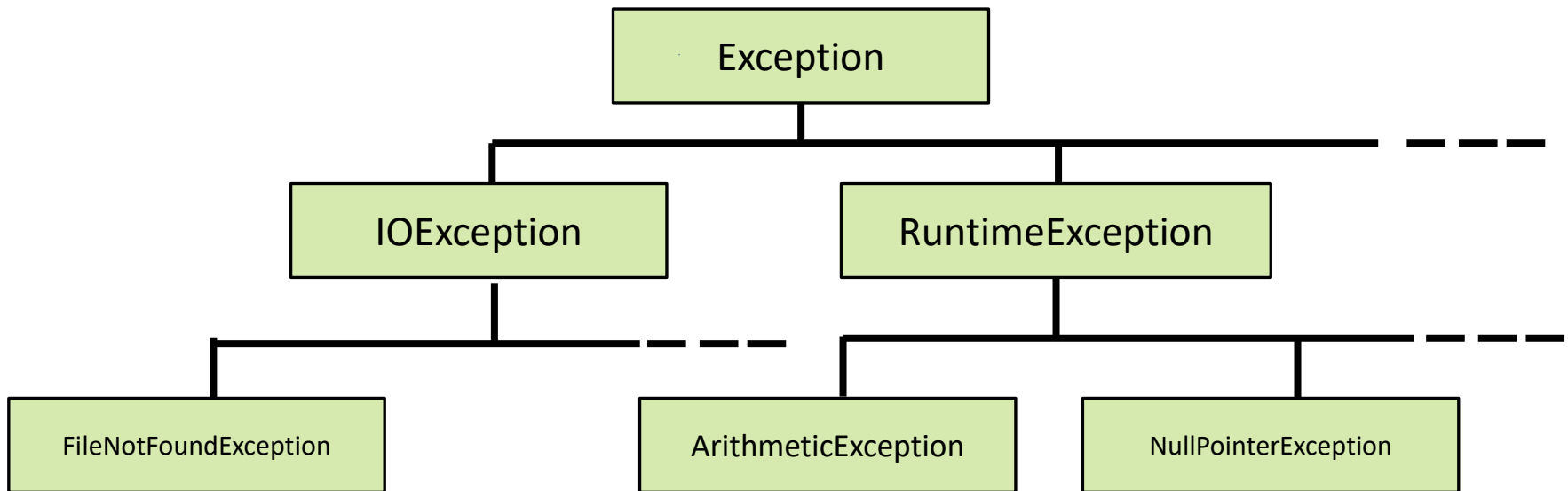
- You can set the message when you construct your own Exception object:
 - `Exception myBad = new Exception("I blew it!");`
- The "stack trace" is set automatically

Exceptions vs Errors

- There's another class of objects called "Error" in Java
- As opposed to Exceptions, Errors are more **serious problems** that a **program is not supposed to manage / handle**
 - when it happens, the program will terminate with a description of the error
- Example: OutOfMemoryError (when the virtual machine is out of memory)

Exception hierarchy

- The built-in types of Exceptions form a "hierarchy"



Catching exceptions

- Exceptions can be **caught and handled** without terminating the program



Catching exceptions

- Exceptions can be **caught and handled** without terminating the program:

```
try {  
    ...statements...  
}  
catch(OfTypeException e) {  
    ...do this if one of those Exceptions happens...  
}  
...continue here...  
...(unless some un-caught exception occurs)...
```

Catching specific exceptions

- You can choose to catch only one specific kind of Exception:

```
try {  
    ...statements...  
}  
catch(ArithmeticException ae) { .... }  
//Other kinds will not be caught at all
```

Catching specific exceptions

- Or handle different types in different ways:

```
try {  
    ...statements...  
}  
catch(ArithmeticException ae) { ..handle one way.. }  
catch(IOException ioe) { ..handle some other way.. }
```

Catching specific exceptions

- Or handle different types in different ways:

```
try {  
    ...statements...  
}  
catch(ArithmeticException ae) { ..handle one way.. }  
catch(IOException ioe) { ..handle some other way.. }
```

You can have as many “catch” blocks as you want.

When you have multiple catch blocks (as in the above example), only one catch block will handle the exception → the first one that is of the corresponding type (exception type matches thrown exception)

Catching specific exceptions

- Handling a type of Exception will also cover all other types beneath it in the hierarchy

```
try {  
    ...statements...  
}  
catch(Exception e) {  
    ...will catch absolutely everything... }
```

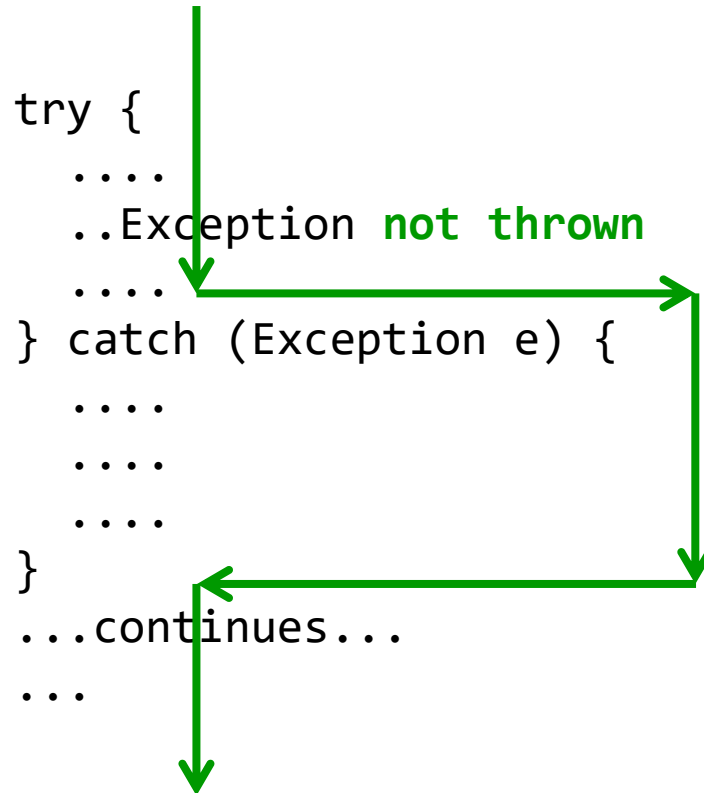

Catching specific exceptions

- Handling a type of Exception will also cover all other types beneath it in the hierarchy

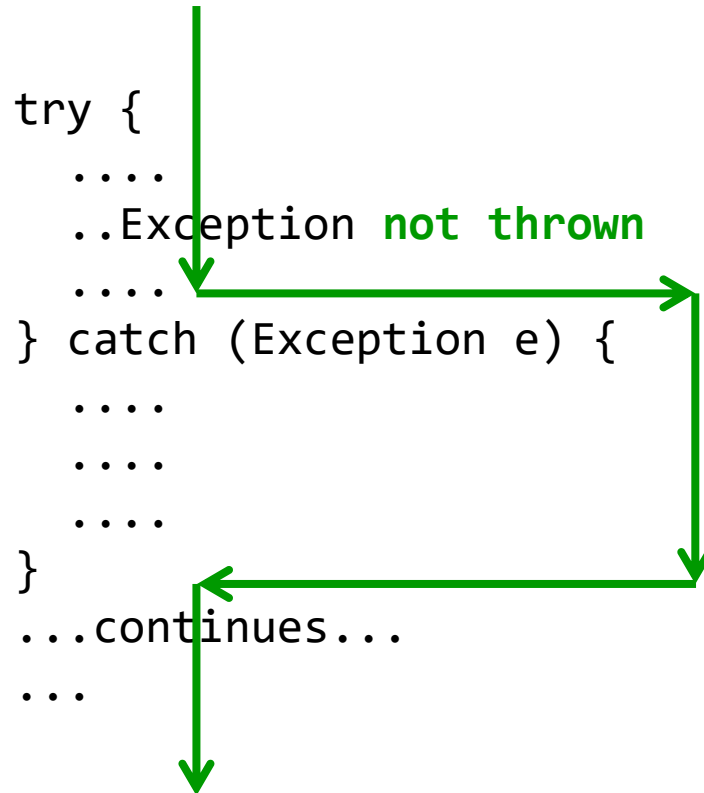
```
try {  
    ...statements...  
}  
catch(Exception e) {  
    ...will catch absolutely everything... }
```

Here, **Exception** is at the top of the hierarchy of exceptions, so it will always catch any type of exception thrown.
This means that if we added more specific catch blocks below it (with more specific type of exceptions), they would never receive anything!

Execution flow in try-catch

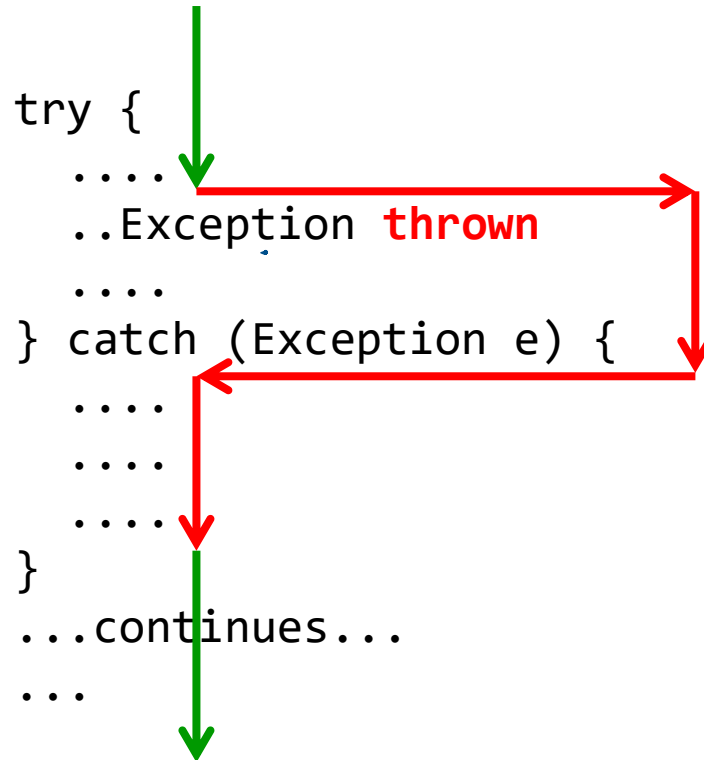


Execution flow in try-catch



When no exception is thrown in the try block, the program skips over the catch blocks and continues the execution of the code below the try-catch

Execution flow in try-catch



When an exception is thrown in the try block, the first catch block “catching” the exception will execute, and then the program continues the execution of the code below the try-catch

finally

- You can add a **finally** block at the end of your try-catch → this block is **always executed**, no matter what happens

```
try {  
    ...statements...  
}  
catch(Exception e) {  
    ...will catch absolutely everything... }  
finally{  
    ... do this every time, whether try successfully completed or  
    an exception was thrown and caught...  
}
```

Exceptions are not “returned”

- You don't normally pass Exceptions around in the normal way
 - You want methods to have their usual results (return values)
 - **AND** you want them to be able to report problems

Exceptions are “thrown”

- So when something nasty happens, you can “throw” an exception, using the `throw` keyword:
`throw new Exception("Bail out!!");`
- This object is now automatically returned up the “stack” or “call chain” – immediately terminating all of these methods as it goes.
 - Until it hits something that “catches” it...

Exceptions are “thrown”

- All methods that either throw an exception or call a method that throws an exception have this statement at the end of the signature:

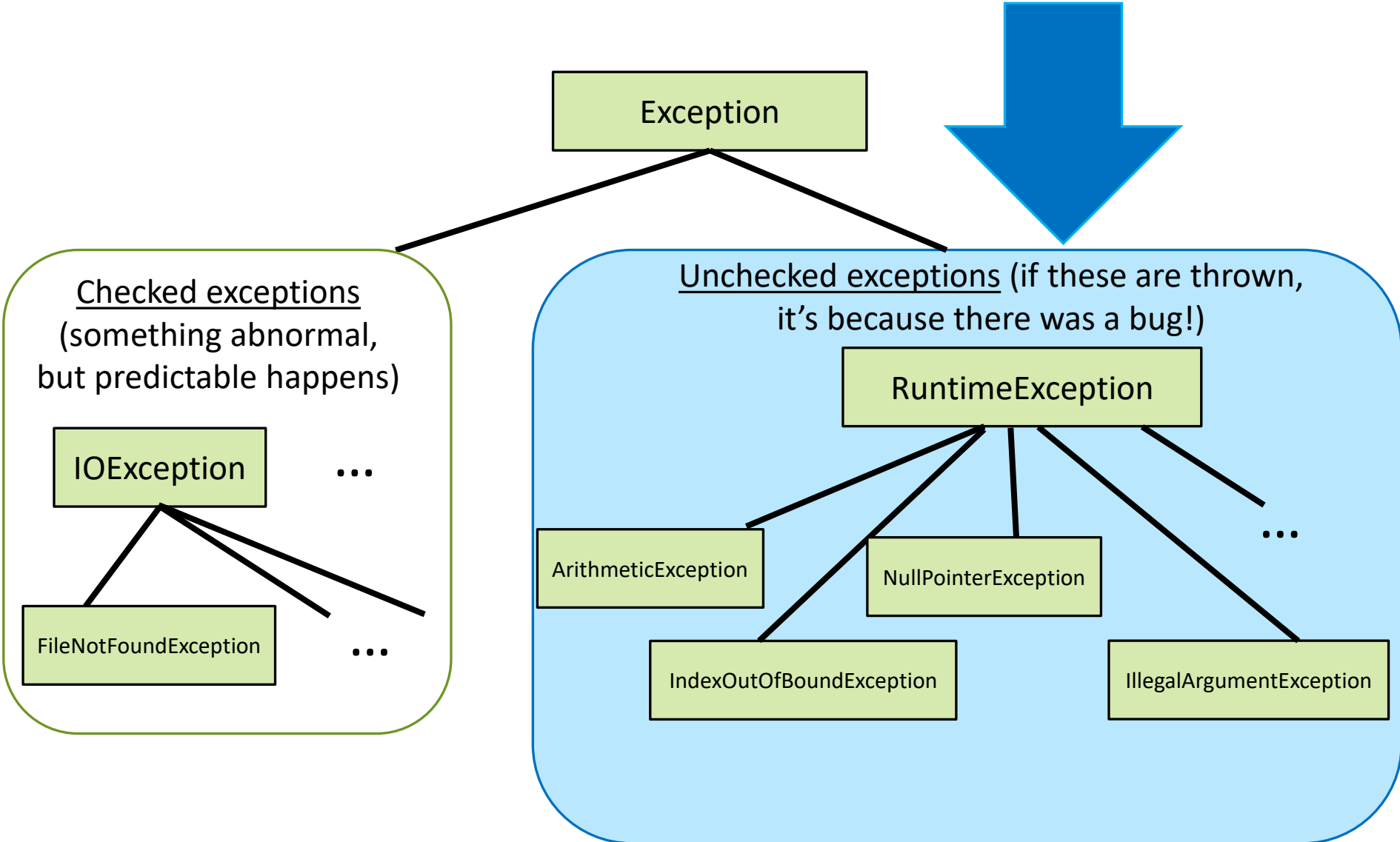
throws Exception //could be a sub-type of Exception

- This is necessary, unless the method itself “catches” the exception thrown by the callee

Some exceptions don't need to be handled

- You have probably noticed that some exceptions are popping up from time to time when you run your code
- Most notably, you have probably seen by now:
 - `NullPointerException`
 - `IndexOutOfBoundsException`
 - `IllegalArgumentException`
 - `ArithmeticException`
- Those exceptions form a hierarchy of exceptions under `RuntimeException` (which is under `Exception`)

Some exceptions **don't need to be handled**



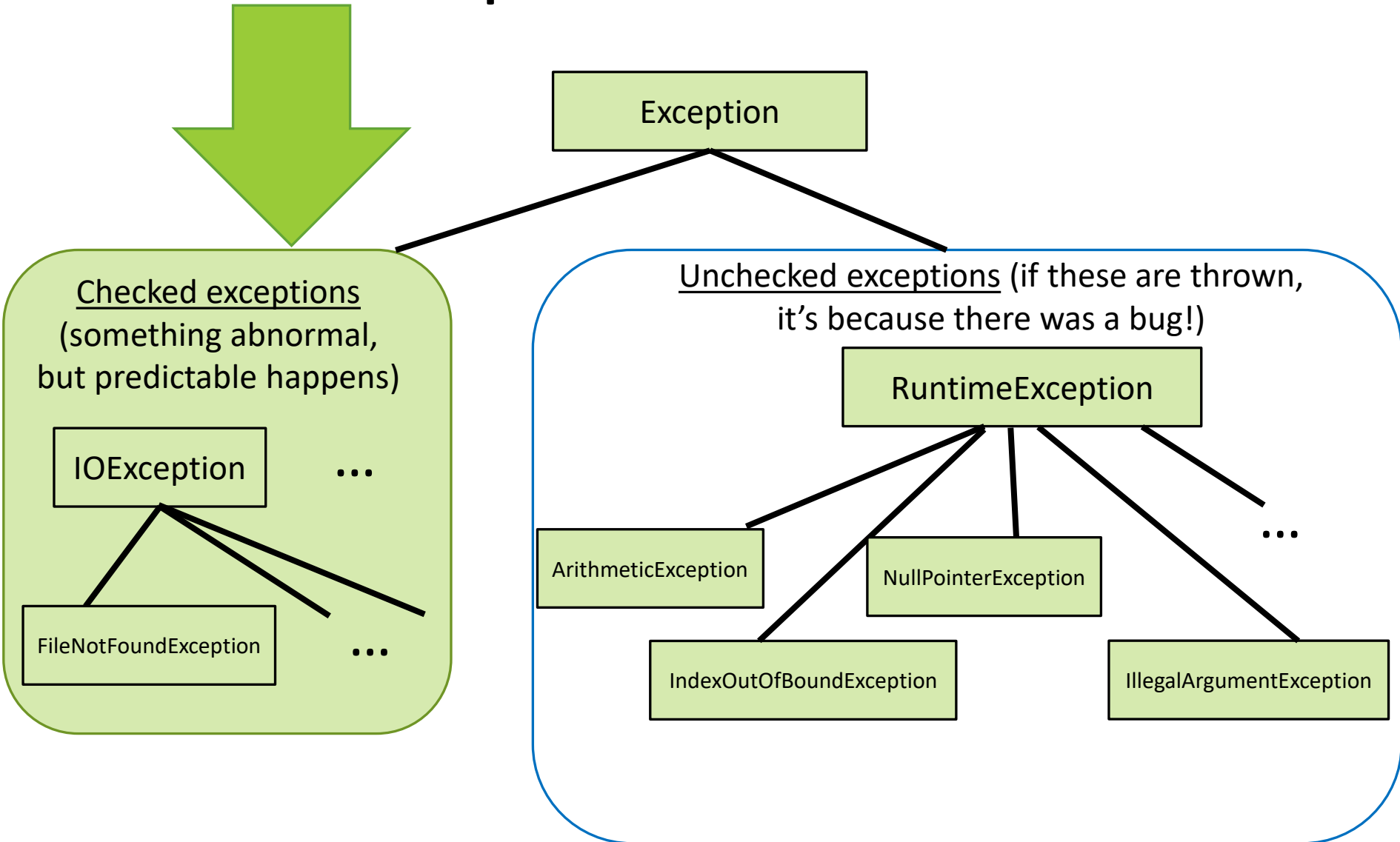
Some exceptions don't need be handled

- RuntimeExceptions are **unchecked**:
 - You don't need to catch them
 - You don't need to declare in your method signature that it could throw them
 - RuntimeExceptions are thrown when the programmer makes a mistake and calls a method incorrectly (NullPointerException is a good example of that...)

Some Exceptions **must be handled**

- **Checked** Exceptions **must be handled**
 - Everything under Exception (in the hierarchy) that is not a RuntimeException is a checked Exception
 - Most IO exceptions are of this type
 - Checked Exceptions must either be:
 - Caught by a try-catch block, or
 - Thrown from this method to the calling method (by declaring “throws ExceptionType” in the signature)

Some Exceptions **must be handled**



Checked Exception example

```
public static void methodX {  
    ....  
    readFile(); //Might throw IOException  
    ....  
} //methodX
```

- This is **not allowed**. Something must be done to handle that IOException.

Checked Exception example

```
public static void methodX {  
    ....  
    try {  
        readFile(); //Might throw IOException  
    } catch(IOException e) {  
        ....  
    }  
    ....  
} //methodX
```

- Solution 1: Catch it

Checked Exception example

```
public static void methodX throws IOException {  
    ....  
    readFile(); //Might throw IOException  
    ....  
} //methodX
```

- Solution 2: **Throw it**

Checked Exception example

```
public static void methodX throws IOException {  
    ....  
    readFile(); //Might throw IOException  
    ....  
} //methodX
```

- Solution 2: **Throw it**
 - If any method call or action inside this method causes an IOException of any kind, it's thrown back to the method that called this one.

Checked Exception example

```
public static void methodX throws IOException {  
    ....  
    readFile(); //Might throw IOException  
    ....  
} //methodX
```

- Solution 2: **Throw it**
 - If any method call or action inside this method causes an IOException of any kind, it's thrown back to the method that called this one.
 - This method is immediately terminated at that point.

Checked Exception example

```
public static void methodX throws IOException {  
    ....  
    readFile(); //Might throw IOException  
    ....  
} //methodX
```

- Solution 2: **Throw it**
 - If any method call or action inside this method causes an IOException of any kind, it's thrown back to the method that called this one.
 - This method is immediately terminated at that point.
 - Now any other method that uses methodX must also catch or throw IOExceptions.

Returning from a method - case 1

```
public static int methodX throws IOException {  
    ....  
    readFile(); //No IOException is thrown this time  
    ....  
    return 42;  
} //methodX
```

Returning from a method - case 2

```
public static int methodX throws IOException {  
    ....  
    ↓ readFile(); //IOException is thrown this time  
    ....  
    return 42;  
} //methodX
```

The diagram illustrates the flow of an exception. A green arrow points down to the `readFile()` call in the method body. A red arrow originates from the `readFile()` call, points right, and then turns upwards to point at the `IOException` in the `throws` clause of the method signature.

What throws what?

- You have to handle checked exceptions when using a method that throws a checked exception
- So... how do you know if a method throws something?

What throws what?

- You have to handle checked exceptions when using a method that throws a checked exception
- So... how do you know if a method throws something?
- KEEP CALM and read the MANUAL

What throws what?

- We haven't dealt with checked Exceptions yet in our in-class examples, so let's go back and look at a method that throws an unchecked Exception
 - Note that **you can** catch unchecked Exceptions if you want to (but you don't have to)

What throws what? → Read the manual

- Search for “Java Integer class” online
 - Go to the “SE8” (or “SE7”) one
- Scroll down to the "parseInt()" description
 - Right at the top you see
 - "throws NumberFormatException". Aha!

<https://docs.oracle.com/javase/8/docs/api/java/lang/Integer.html>

What throws what? → Read the manual

- Search for “Java Integer class” online
 - Go to the “SE8” (or “SE7”) one
- Scroll down to the "parseInt()" description
 - Right at the top you see
 - "throws NumberFormatException". Aha!
- What kind of Exception is that?
 - Click on "NumberFormatException"
 - You can immediately see that:
 - It's a kind of "IllegalArgumentException"
 - which is a kind of "RuntimeException"
 - which is a kind of "Exception"
 - Catching any of these will work.

<https://docs.oracle.com/javase/8/docs/api/java/lang/Integer.html>

Stupid try-catch tricks

- Technically, you can catch `RuntimeException`s:

```
int[] mySmallArray = {1,2,3,4,5,6,7,8,9};
try{
    for (int i = 0; i < 1000; i++)
    {
        System.out.println(mySmallArray[i]);
    }
}
catch (ArrayIndexOutOfBoundsException e){
    System.out.println("End of the array");
}
```

Will print:

1

2

3

4

5

6

7

8

9

End of the array

Stupid try-catch tricks

- Technically, you can catch `RuntimeException`s:

```
int[] mySmallArray = {1,2,3,4,5,6,7,8,9};
try{
    for (int i = 0; i < 1000; i++)
    {
        System.out.println(mySmallArray[i]);
    }
}
catch (ArrayIndexOutOfBoundsException e){
    System.out.println("End of the array");
}
```

Will print:

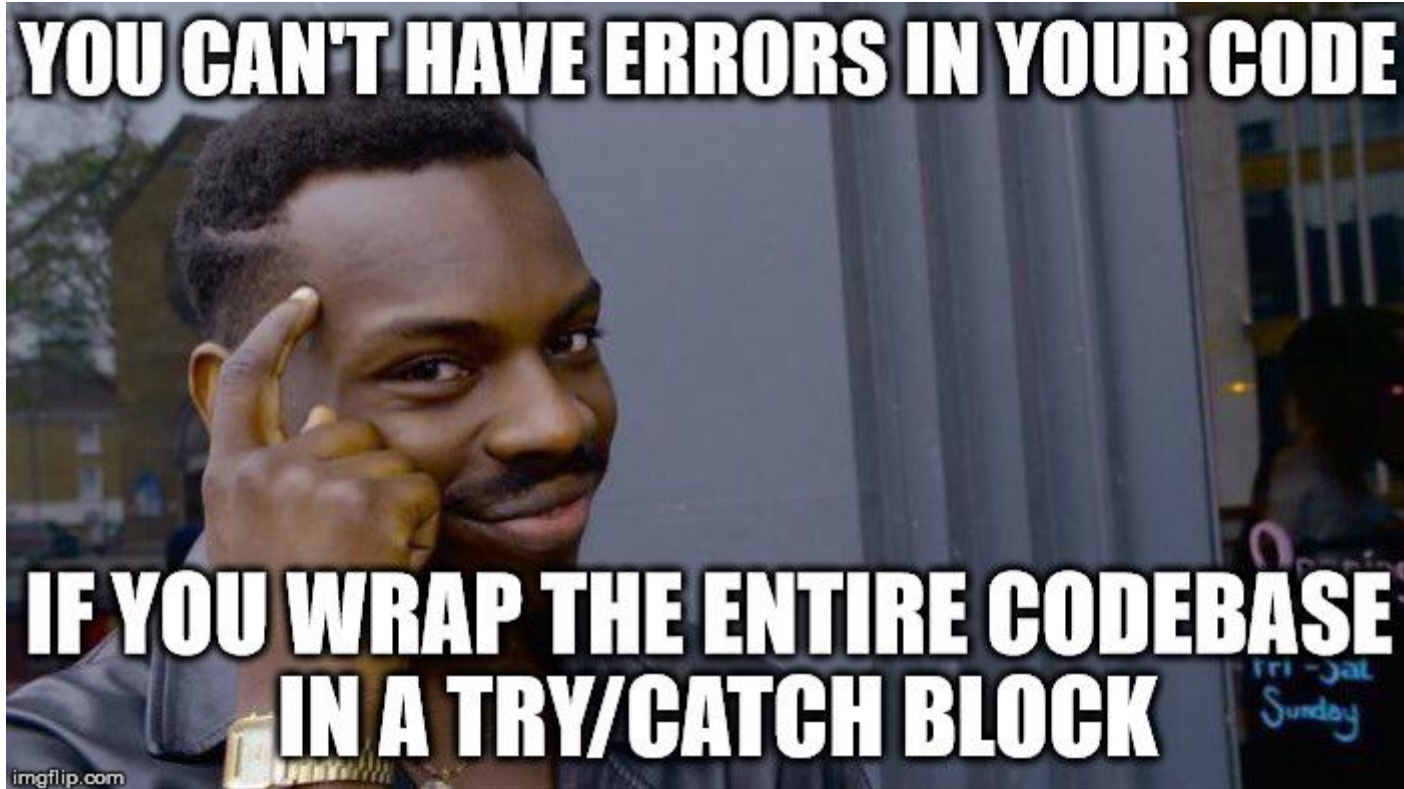
```
1
2
3
4
5
6
7
8
9
End of the array
```

- But don't do that. This is bad programming. Write the loop correctly in the first place.

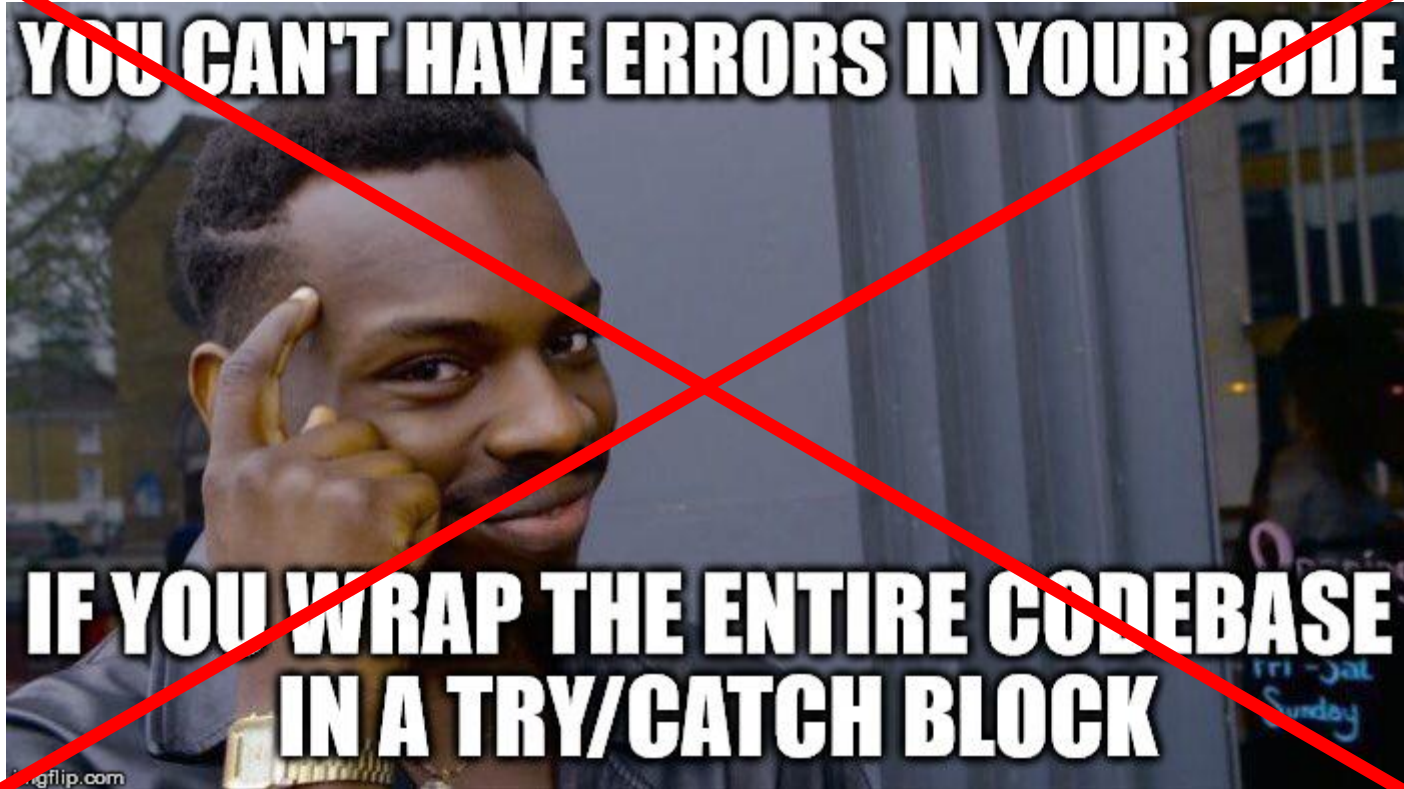
Do not use try-catch as an if

- You're **not supposed** to use try-catch to deal with unchecked exceptions and emulate if statements
- Most unchecked exceptions result from bugs and/or missing safety mechanisms in your code
 - don't be lazy and make the appropriate checks instead of using try-catch for those
 - e.g. check if your reference is null before calling a method on it, instead of catching the `NullPointerException`

Do not use try-catch as an if



Do not use try-catch as an if



Don't think like that!

Import may be needed

- These Exceptions are just types of objects like any other.
 - Some are automatically available
 - Some need the correct "import" at the top
- Example:
 - IOException is really java.io.IOException so
 - Use `import java.io.*;`
 - Or use `import java.io.IOException;`

File Input / Output

- It is pretty useful to be able to read or write to a file
 - Allows you to analyse information stored in a file
 - Allows you to print the results of an analysis in a file
- All programming languages allow you to do that
- Let's see how this works in Java

File Input / Output

- Java has multiple built-in classes for reading and writing to a file
 - They also form a hierarchy of classes, and range from low-level to high-level classes:
 - Low-level classes are simple and basic, they don't offer many methods and are not very user-friendly
 - Mid or high-level classes are often “wrapped” around lower-level classes, they offer more methods and are easier to use

File Input / Output

- Java has multiple built-in classes for reading and writing to a file
 - You can also divide them into 2 groups
 - Those who read / write bytes
 - Those who read / write characters

Stream

- Java reads or writes through “streams” of data (either in bytes or characters)

Reading a file



Writing to a file

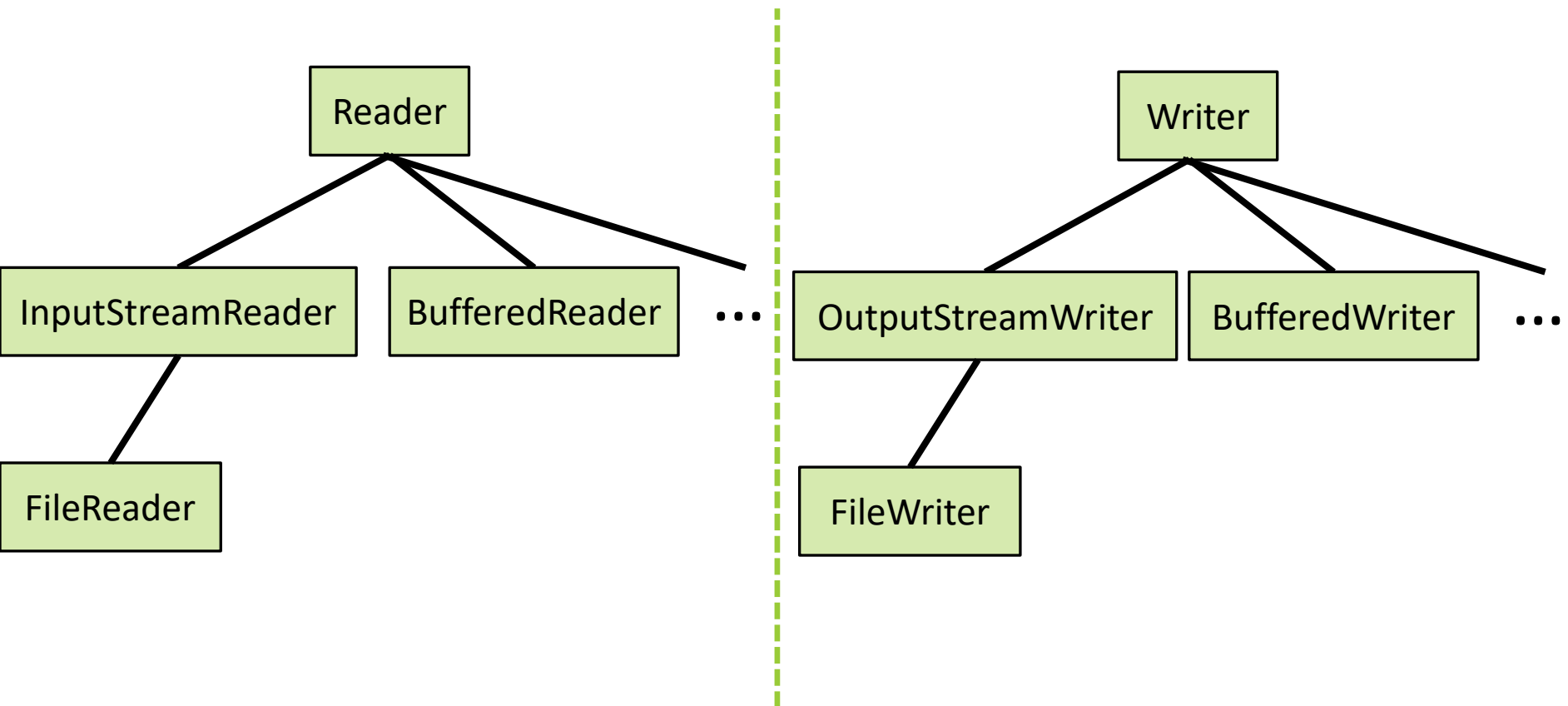


Typical use

- For reading:
 - Open a stream (building an object)
 - Use the object's methods to get data
 - Close the stream (calling close() on the object)
- For writing:
 - Open a stream (building an object)
 - Use the object's methods to write data
 - Close the stream (calling close() on the object, which also “flushes” the stream → emptying it in the file)

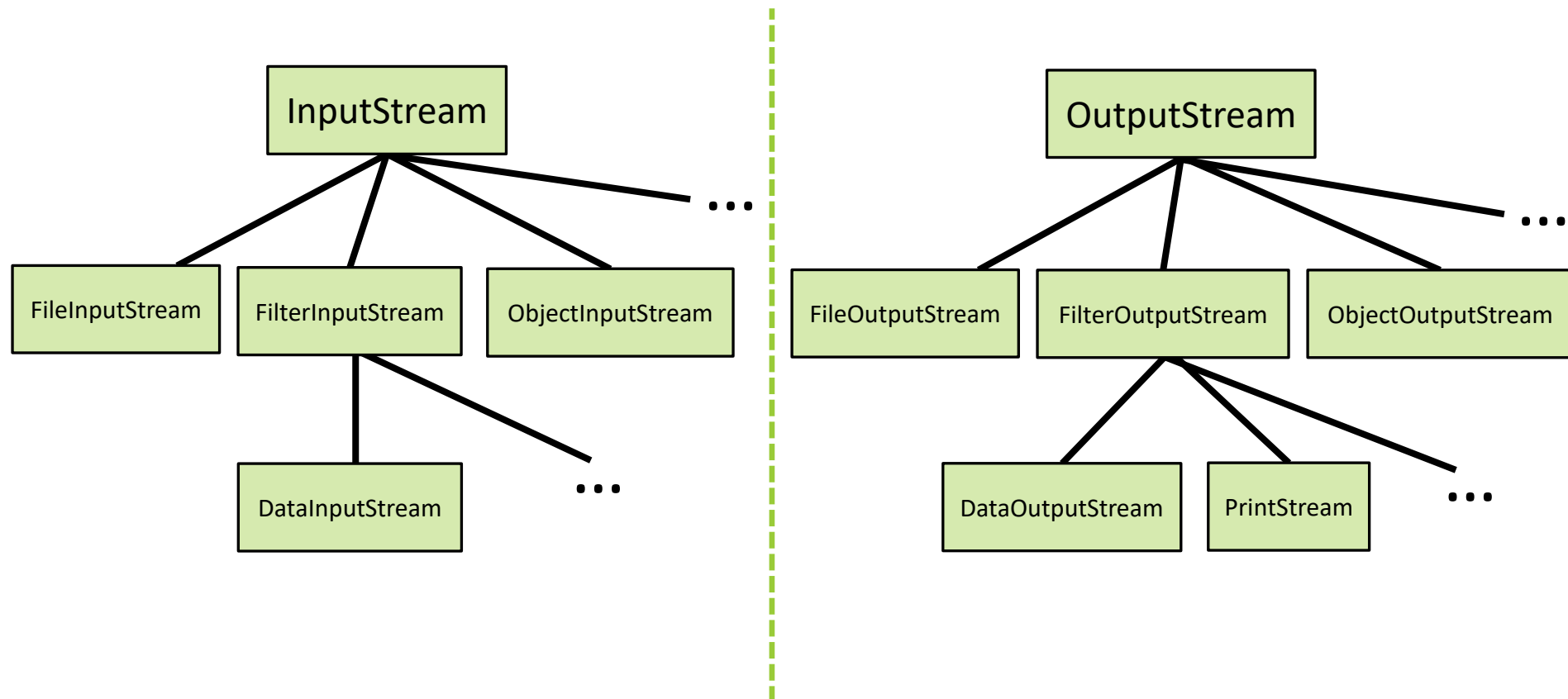
Some class examples for I/O

- I/O of characters:



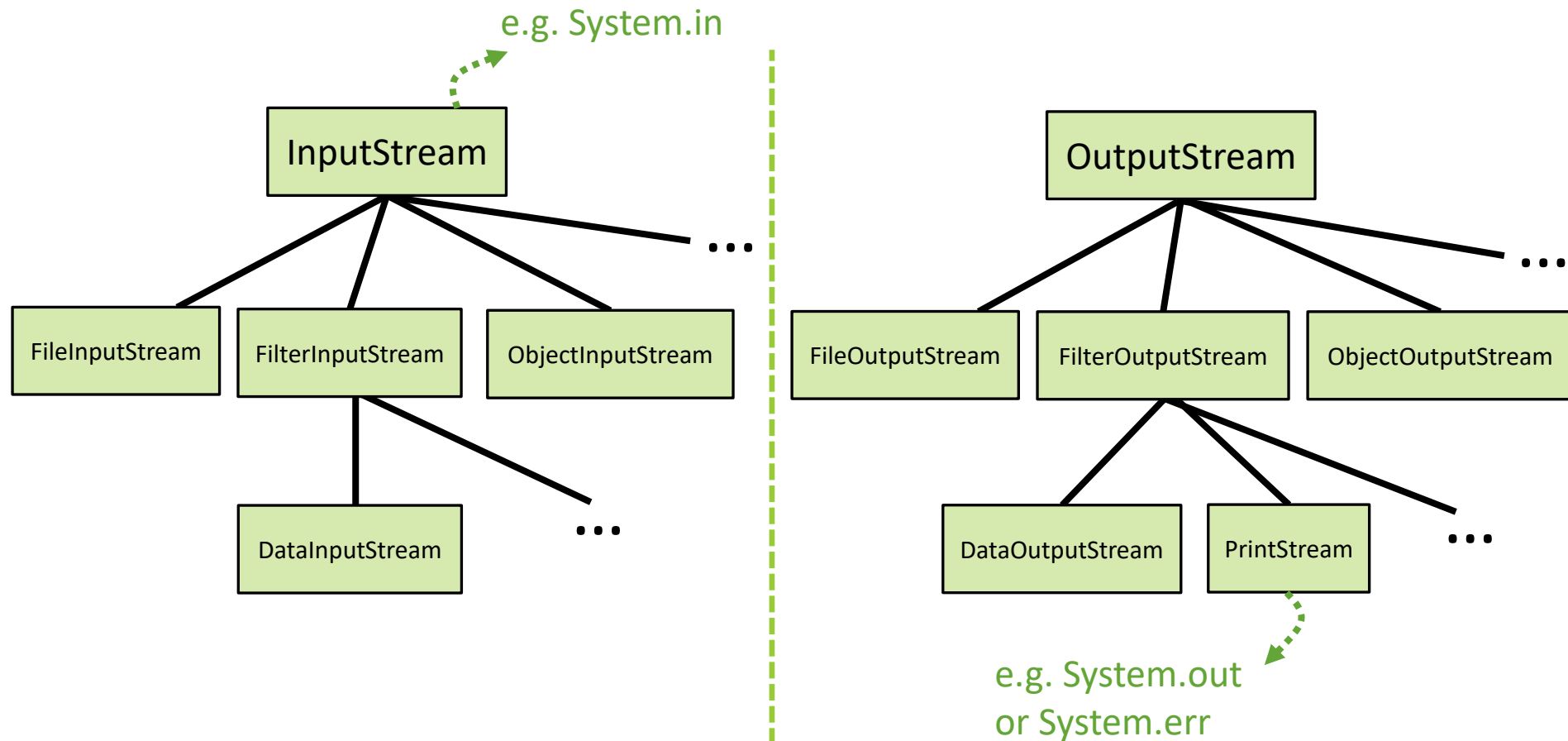
Some class examples for I/O

- I/O of bytes:



Some class examples for I/O

- I/O of bytes:



Reading from files - low level

- FileReader is an example of a low-level, basic reader
 - It's not really meant to be used directly
 - Will usually be “wrapped” inside another class (another type of reader)

Reading from files - low level

- FileReader is an example of a low-level, basic reader
 - It's not really meant to be used directly
 - Will usually be “wrapped” inside another class (another type of reader)
- Constructors for FileReader:
 - new FileReader(String)
 - example: new FileReader("data.txt");
 - Or new FileReader("C:\\Users\\Me\\Documents\\data.txt");
 - new FileReader(File)
 - We'll cover File objects later.

Reading from files - mid level

- We'd need to write some methods to help us actually use a `FileReader`... of course, someone has already done that:
- A `BufferedReader`
 - Handles the `FileReader` for us
 - Provides more user-friendly methods
 - Still not very fancy (won't read an `int` or a `double`)

Reading from files - mid level

- We'd need to write some methods to help us actually use a FileReader... of course, someone has already done that:
- A BufferedReader
 - Handles the FileReader for us
 - Provides more user-friendly methods
 - Still not very fancy (won't read an int or a double)
- Constructor:
 - `new BufferedReader(FileReader)`
 - This is **known as “wrapping”** a FileReader inside a BufferedReader

BufferedReader methods

The fundamental methods:

- `String readLine()`
 - Reads the next entire line from the file
 - Returns null if there are no more lines
- `int read()`
 - Reads one character from the file
 - But returns its character code as an int, not a char.
 - Cast it to (char) if you want to, but not if...
 - It returns -1 for “end of file” (no more chars)

BufferedReader methods

The fundamental methods:

- `void close()`
 - Release control of the file
 - Should always be done after any file I/O of any type
 - After the stream has been closed, calling methods on the `BufferedReader` will throw an `IOException`

Notes on file reading

- For any type of file I/O you need
 - to use `import java.io.*;`
 - to catch many kinds of `IOExceptions`

Notes on file reading

- BufferedReader will still only give you characters, or a complete line
- To read int, double, tokens, etc. use a Scanner as usual

BufferedReader inFile; *//open it normally (not shown)*

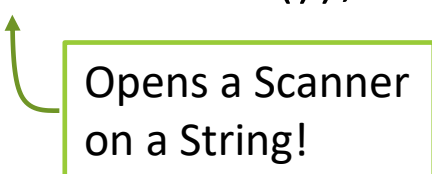
...

Scanner lineReader = new Scanner(inFile.readLine());

int i = lineReader.nextInt();

double x = lineReader.nextDouble();

lineReader.close(); *//Clean up afterwards. Be nice.*



Opens a Scanner
on a String!

Advanced Scanner

- When a Scanner reads a line or String, it treats it as a sequence of “tokens” (any consecutive non-blank characters)
- Example: If it gets the line or String

34.2 Fred false -3

it sees it as

"34.2" then "Fred" then "false" then "-3"

Advanced Scanner

- You can see if there are any more tokens, and, if so, get the next one (as a String) with:

`scannerObject.hasNext()` //gives a boolean answer

- “Is there another token?”

`scannerObject.next()` // gives a String

- “Give me the next token.”

Advanced Scanner

- If you want to read tokens as int, long, float, double, or boolean (but not char):

`scannerObject.hasNextInt()` //or Long or Float or ...

- Returns true if the next token is OK for that type
- **BUT DOES NOT READ IT**

`scannerObject.nextInt()` //or Long or Float or ...

- Actually reads it, as that type

Advanced Scanner

- The `nextLine()` method is different
 - It ignores tokens and just gives you the whole line as one big String
 - It's your problem what to do with it

Another approach for parsing lines

- When you get a String from `readLine()`, sometimes it can be useful to “split” the String
- Let’s say that your file has multiple rows (each line), and items on each line (each column) is separated by a specific character (e.g. a space, a dash, or a comma)
 - Use `split(String separator)` from the String class → returns a `String[]`, with each item separately in each cell of the array

Writing to files - low level

- The low-level object is a `FileWriter`
 - Also not really meant to be used directly
- Constructors:
 - `new FileWriter(String);` //Give it the file name (the String)
 - `new FileWriter(File);` //We'll cover "File" objects later
- These two will immediately erase all old contents of the file, and create new content → **for overwriting**

Writing to files - low level

- The low-level object is a `FileWriter`
 - Also not really meant to be used directly
- Constructors:
 - `new FileWriter(String,true);`
 - `new FileWriter(File,true);`
- These two will append data to the end of a file, leaving its current contents untouched → **for appending**

Writing to files - high level

- A `PrintWriter` will control a `FileWriter` for you, and give you the usual convenient methods:
- Constructor: `new PrintWriter(FileWriter)`

(You can also build the `PrintWriter` on a `String` representing the filename, or a `File` object)

Writing to files - high level

- A `PrintWriter` will control a `FileWriter` for you, and give you the usual convenient methods:
- Methods:
 - `void print(..any type..)`
 - `void println(..any type..)`
 - `void printf(String, ...others...)` *//we'll discuss later*
 - `void format(String, ...others...)` *//same as printf*
 - `void close()` *//should always be done for files*

Writing to files - high level

- A `PrintWriter` will control a `FileWriter` for you, and give you the usual convenient methods:
- Methods:
 - `void print(..any type..)`
 - `void println(..any type..)`
 - `void printf(String, ...others...)` *//we'll discuss later*
 - `void format(String, ...others...)` *//same as printf*
 - `void close()` *//should always be done for files*

Again, you must:

- Use `import java.io.*;`
- catch different types of `IOExceptions`

END of Quiz 2 Syllabus

Extra material - File object

- A File object does not do any I/O
 - Instead, it contains all of the information needed to identify a particular file
 - Its name, the "path" to find it, what disk it's on, etc.
 - Usually very system-dependent
- One constructor: `new File("file name")`
- You can use one of these to create a `FileReader`, a `FileWriter`, a `PrintWriter` or even a `Scanner` directly

Extra material - File Chooser

- A JFileChooser object can be used to
 - give the user an ordinary file dialog box
 - obtain a File object
 - which can be used to create a FileReader, FileWriter, etc.
 - which in turn can be used to create a BufferedReader or PrintWriter

File Chooser - advanced

- You can also add features to:
 - Default to the current directory
 - Restrict the choice to a particular type of file
 - Handle the "Cancel" button properly
 - Etc.
- This is beyond the scope of this course
 - Read over the JFileChooser documentation if you need to do this

Cutting out the intermediary

- If you want to read non-character data (int, double, etc.) from a text file, you can use a Scanner directly
 - A Scanner can read directly from a file all by itself
 - You don't need a BufferedReader
 - But you do need a File object to select the file:

```
Scanner in = new Scanner(new File("input.txt"));
```


(What happens if you leave out the new File() part?)

Extra material: formatting

- Instead of `print()` or `println()` you can use `printf()` or `format()` to control output exactly
 - `printf` and `format` are two names for the same method

Extra material: formatting

- Here's how to use them:

```
System.out.format("Casting %f to int gives %d %n",  
                  23.8, (int)23.8 );
```

- The first parameter is a String that indicates exactly how you want the data printed
 - The red codes that start with % are where the data goes
 - Except %n which just gives a newline character
- There can be any number of other parameters
 - These supply the actual data to print (in blue)

Formatting codes

- Commonly used codes:
 - `%d` – print a decimal integer here (base 10 integer)
 - `%6d` – use at least 6 characters to do that
 - `%f` – print a floating-point value here
 - `%6f` – use at least 6 characters to do that
 - `%6.2f` – with exactly 2 of them after the decimal point
 - `%s` – print a String here
 - `%n` – print a newline (`\n` character) here
- There must be one additional parameter (after the String) for each code used (except `%n`), and it must be the correct type

Formatting codes

- Previous style:

```
System.out.println(a+" plus "+b+" is "+(a+b)+".\n");
```

- Formatted style:

```
System.out.printf("%d plus %d is %d.%n",a,b,a+b);
```

- Most useful to

- Line up decimal points – perhaps use %7.2f
- Round off a number
 - Use %4.1f to get 98.6 and not 98.5999999999999999

Extra material: Binary files (bytes)

- We saw earlier that there are two types of file I/O:
 - char (text) → human readable with your favorite text editor
 - byte (binary) → not human readable
- You are going to use I/O of characters most of the time, but know that you can output in bytes too
 - See slide [53](#) for different classes allowing you to read/write in bytes

Extra material: Binary files (bytes)

- If you do

```
int x=987654321; //A 4-byte number
```

```
outFile.print(x); //printing to a text file, using a PrintWriter (chars)
```

- It will not print 4 bytes (the size of an int). It will print the 9 characters:

```
'9' '8' '7' '6' '5' '4' '3' '2' and '1'
```

- But you can write the actual 4 bytes of raw memory (for x) into a file
 - The file will be more compact, but not human-readable

Extra material: Binary files (bytes)

- Same sort of constructors as before (using wrapping):

- E.g. for input:

```
DataInputStream in = new DataInputStream(  
    new FileInputStream("rawData.xxx"));
```

- E.g. for output:

```
DataOutputStream out = new DataOutputStream(  
    new FileOutputStream("rawData.xxx"));
```

Extra material: Binary files (bytes)

- Methods for reading and writing:
 - `readInt()`, `readDouble()`, `readLong()`, `readUTF()`, ...
 - `writeInt(x)`, `writeDouble(x)`, `writeLong(x)`,
`writeUTF(x)`, ...
- UTF??
 - That is a standard way to write and read `String` data: 2 bytes for the length, then 1 or 2 bytes per character

Extra material: Binary files (bytes)

- You can also use the `ObjectOutputStream` and `ObjectInputStream` classes to write / read byte versions of objects to / from a file
 - FYI: your objects actually need to support the `java.io.Serializable` interface in order to do that
 - This is also beyond the scope of this course