Objectives

- Wrap up exceptions
- Representing Files
- Streams
  - Byte Streams
  - Text Streams
  - Connected Streams

A Few Words on Assignment 5

- May be the opposite of Assignment 4
- Not as much thinking, more practicing Eclipse

Review

1. Why can Eclipse do all that it can do for Java? (as opposed to what's possible with a Python IDE)
2. Why did I wait until now to show you Eclipse?
3. If your code calls a method that can throw an exception, how can you handle it?
    - (Two options)
4. How do we make a block of code execute regardless of whether some code threw an exception or not?
5. What are benefits of exceptions?

Benefits of Exceptions

- Force error checking/handling
  - Otherwise, won’t compile
  - Does not guarantee “good” exception handling
- Ease debugging
  - Stack trace
- Separates error-handling code from “regular” code
  - Error code is in catch blocks at end
  - Descriptive messages with exceptions
- Propagate methods up call stack
  - Let whoever “cares” about error handle it
- Group and differentiate error types

Exceptions Summary

- Exception handling should be exceptional
  - Exception handling is expensive
- Try to prevent Runtime Exceptions
- Throw Exceptions in your code for improved error handling/robustness
- If your code calls a method that throws an exception
  - Catch the exception if you can handle it well OR
  - Throw the exception to whoever called you and let them handle it

FILES

`java.io.File` Class

- Represents a file or directory
- Provides functionality such as
  - Storage of the file on the disk
  - Determine if a particular file exists
  - When file was last modified
  - Rename file
  - Remove/delete file
  - …

Making a File Object

- Simplest constructor takes full file name (including path)
  - If don’t supply path, Java assumes current directory (.)

    ```
    File myFile = new File("chicken.data");
    ```

  - Creates a File object representing a file named “chicken.data” in the current directory
  - Does not create a file with this name on disk
- Similar to Python:

  ```
  myFile = open("chicken.data")
  ```

Files, Directories, and Useful Methods

- A `File` object can represent a file **or** a directory
  - Directories are special files in most modern operating systems
- Use `isDirectory()` and/or `isFile()` for type of file `File` object represents
- Use `exists()` method
  - Determines if a file exists on the disk

In Python, functionality are in the `os.path` module

More File Constructors

- String for the path, String for filename

```
File myFile = new File("/csdept/courses/cs209/handouts",
"chicken.data");
```

- File for directory, String for filename

```
File myDir = new File("/csdept/courses/cs209/handouts");
File myFile = new File(myDir, "chicken.data");
```

Does this “break” any of Java’s principles?

File Paths Break Java's Portability Principle

- Principle of Portability
  - Write and Compile Once, Run Anywhere
- Problem: file paths are OS-specific
- `java.io.File.separator`
  - OSX/Linux: /
  - Windows: \
- Takeaways:
  - Use *relative* paths
  - Use configuration files (text files, not Java files) to set paths

`java.io.File` Class

- 25+ methods
  - Manipulate files and directories
  - Creating and removing directories
  - Making, renaming, and deleting files
  - Information about file (size, last modified)
  - Creating temporary files
  - …
- See online API documentation

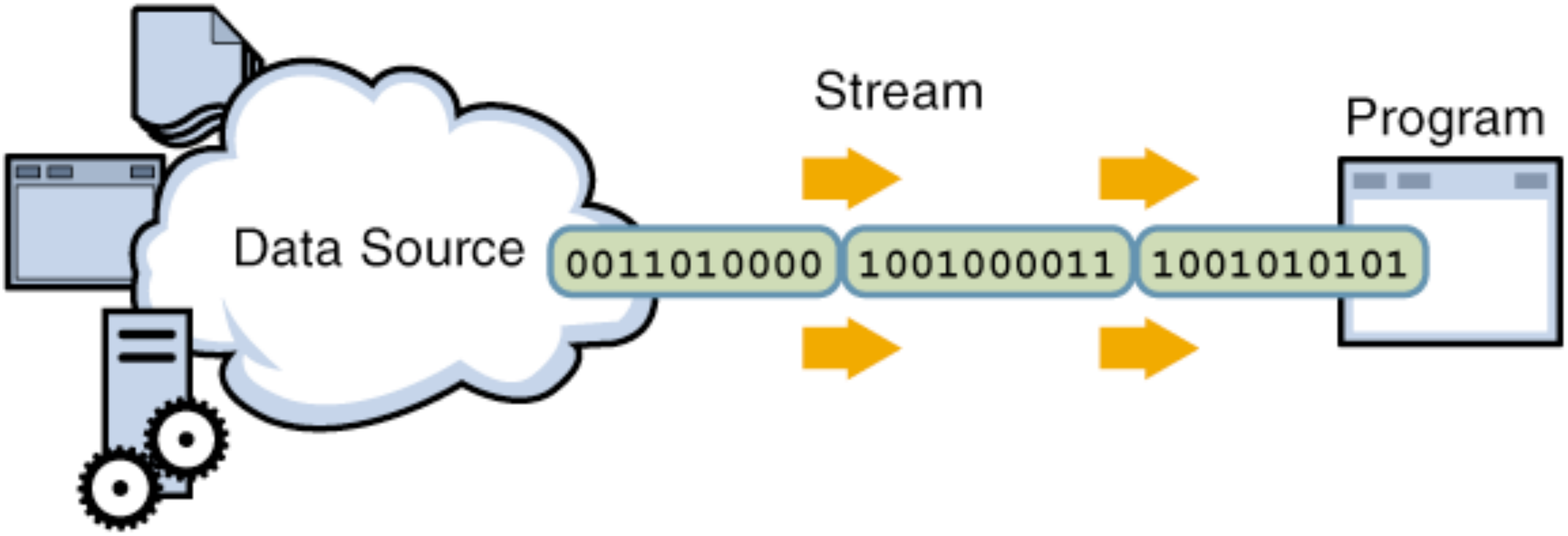
`FileTest.java`

A design case study

STREAMS

Streams

Java handles input/output using ***streams***,
which are sequences of bytes

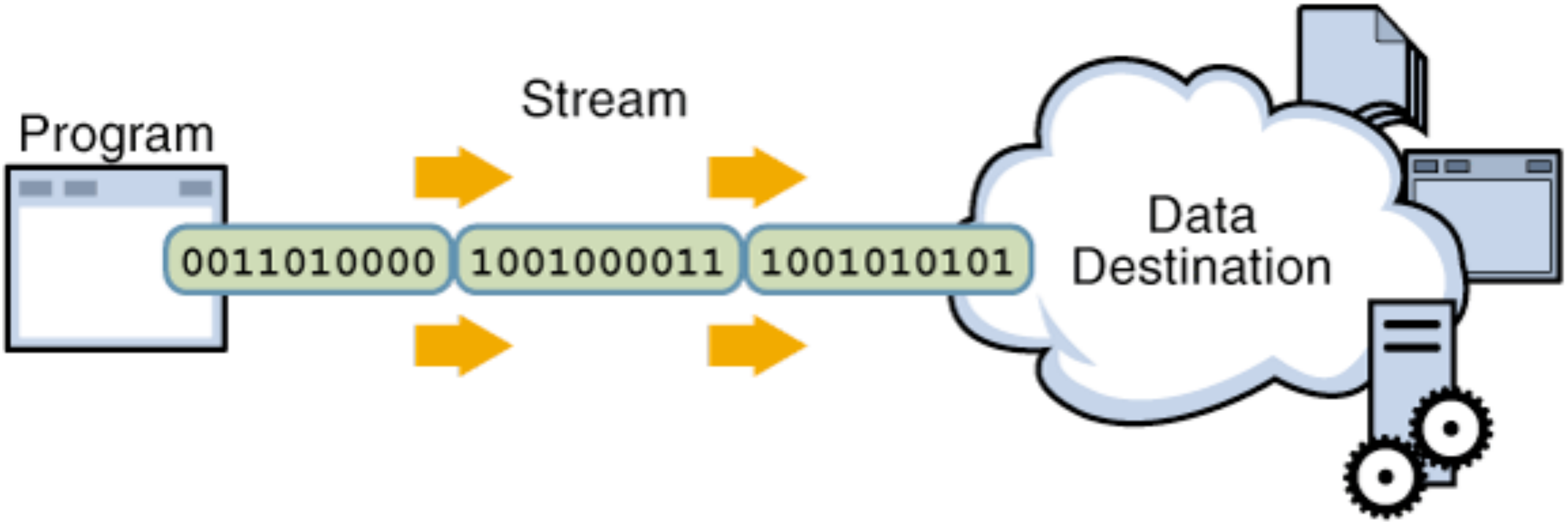


input stream: an object from which we can ***read*** a sequence of bytes

abstract class: `java.io.InputStream`

Streams

Java handles input/output using ***streams***,
which are sequences of bytes



output stream: an object to which we can ***write*** a sequence of bytes

abstract class: `java.io.OutputStream`

Java Streams

- MANY (80+) types of Java streams
- In `java.io` package
- Why **stream** abstraction?
  - Information stored in different sources is accessed in essentially the same way
    - Example sources: file, on a web server across the network, string
  - Allows same methods to read or write data, regardless of its source
    - Simply create an `InputStream` or `OutputStream` of the appropriate type

`java.io` Classes Overview

Two categories of stream classes, based on datatype

- Abstract base classes for **binary** data (bytes)

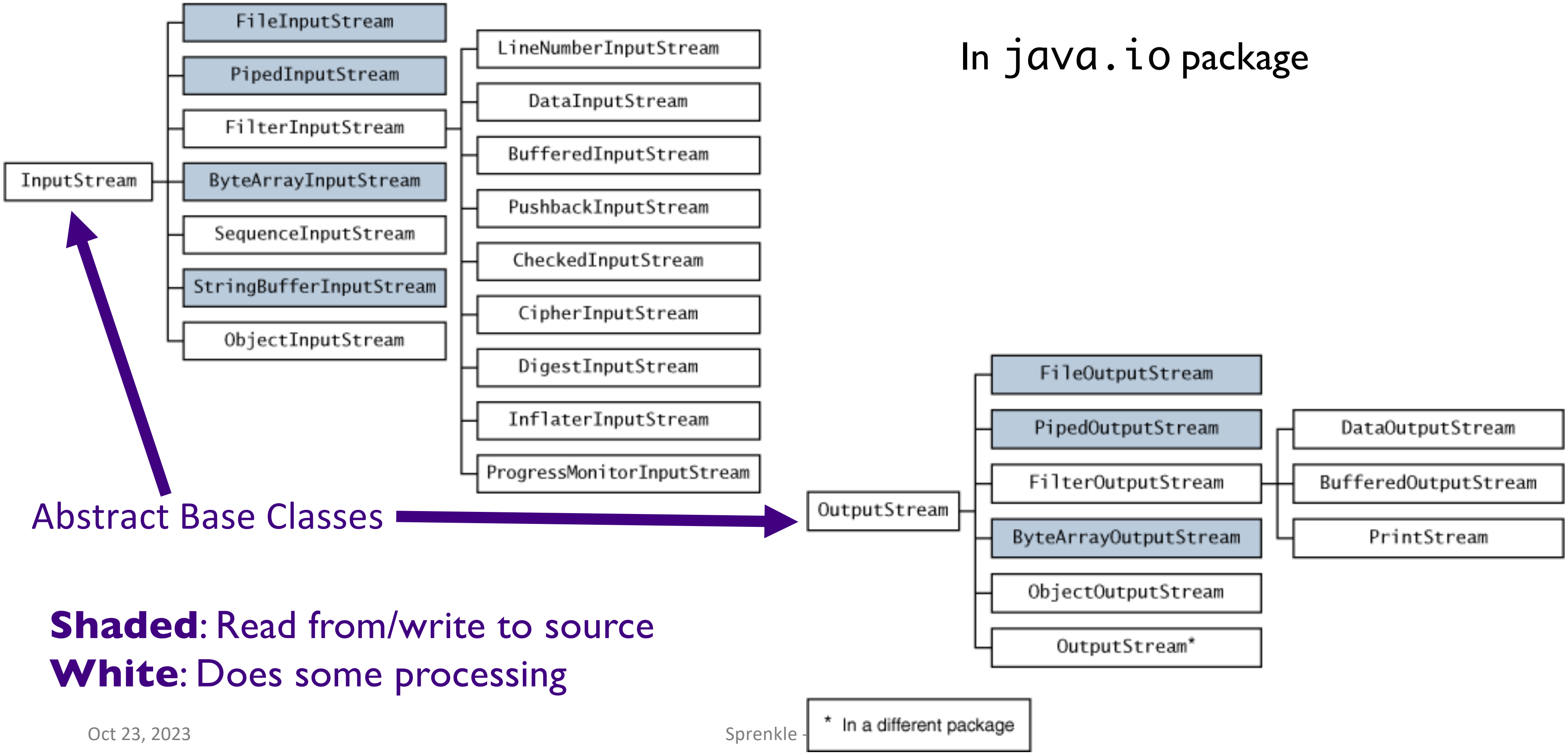
`InputStream` `OutputStream`

- Abstract base classes for **text** data:

`Reader` `Writer`

Byte Streams: For Binary Data

In `java.io` package



InputStream

- FileInputStream
- PipedInputStream
- FilterInputStream
  - LineNumberInputStream
  - DataInputStream
  - BufferedInputStream
  - PushbackInputStream
  - CheckedInputStream
  - CipherInputStream
  - DigestInputStream
  - InflaterInputStream
  - ProgressMonitorInputStream
- ByteArrayInputStream
- SequenceInputStream
- StringBufferInputStream
- ObjectInputStream

OutputStream

- FileOutputStream
- PipedOutputStream
- FilterOutputStream
  - DataOutputStream
  - BufferedOutputStream
  - PrintStream
- ByteArrayOutputStream
- ObjectOutputStream
- OutputStream*

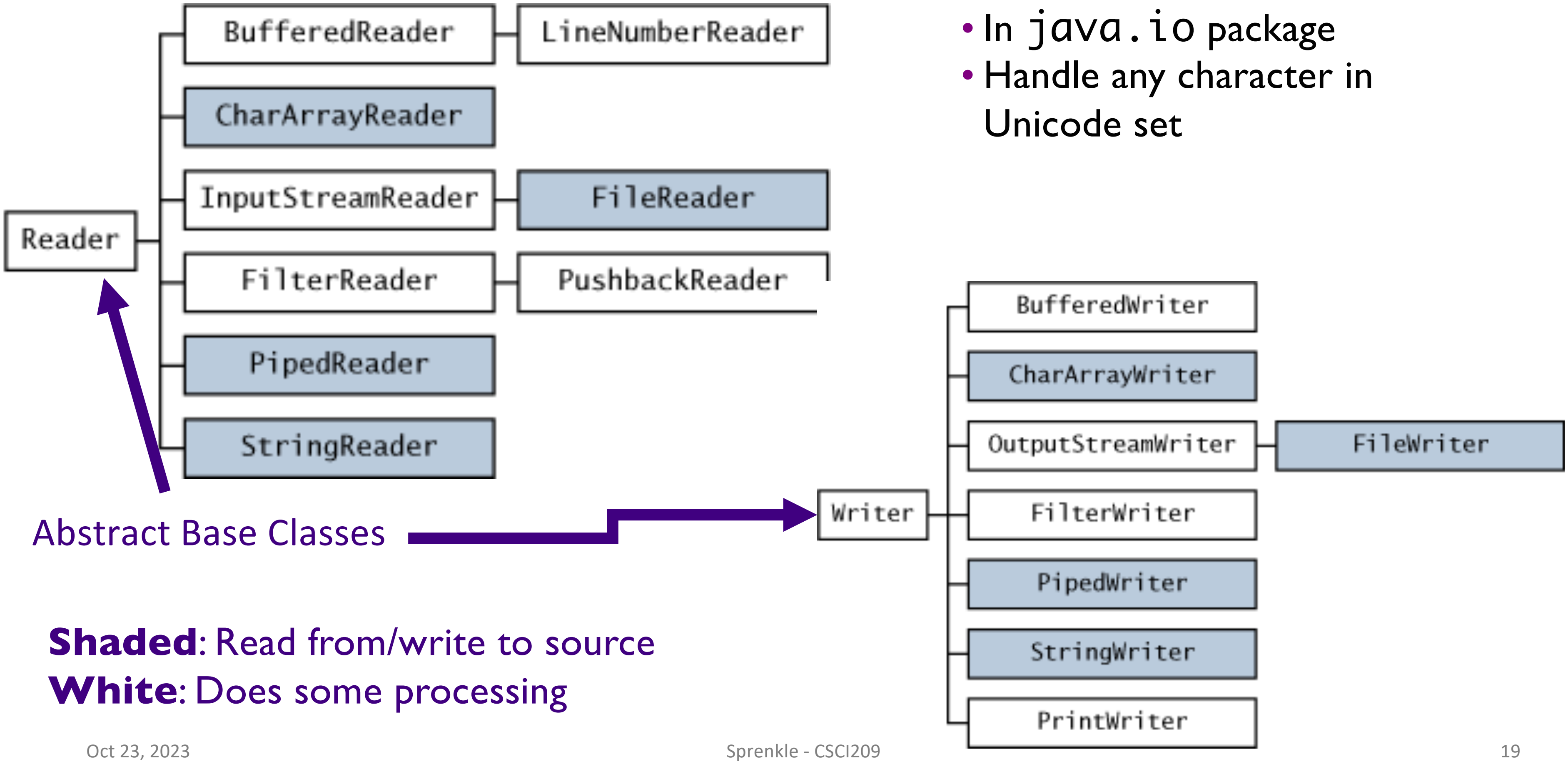
* In a different package

Abstract Base Classes

Shaded: Read from/write to source
White: Does some processing

Character Streams: For Text

- In `java.io` package
- Handle any character in Unicode set



Abstract Base Classes

Shaded: Read from/write to source

White: Does some processing

Console I/O: Streams!

- Output:
  - `System.out` and `System.err` are **`PrintStream`** objects
- Input
  - `System.in` is an **`InputStream`** object
  - Throws exceptions if errors when reading
    - Must handle in `try/catch`
    - Reason we instead used Scanner to read data

Opening & Closing Streams

- Streams are *automatically opened* when constructed

- Close a stream by calling its `close()` method
  - Close a stream as soon as object is done with it
  - Free up system resources

Reading & Writing Bytes

- Abstract parent class: `InputStream`
  - `abstract int read()`
    - reads one byte from the stream and returns it
  - Concrete child classes override `read()` to provide appropriate functionality
    - e.g., `FileInputStream`'s `read()` reads one byte from a *file*
- Similarly, `OutputStream` class has abstract `write()` to write a byte to the stream

File Input and Output Streams

- **`FileInputStream`**: provides an input stream that can read from a file
  - Constructor takes the name of the file:

```java
FileInputStream fin = new FileInputStream("chicken.data");
```

  - Or, uses a **`File`** object ...

```java
File inputFile = new File("chicken.data");
FileInputStream fin = new FileInputStream(inputFile);
```

FileTest.java

More Powerful Stream Objects

- `DataInputStream`
  - Reads Java primitive types through methods such as `readDouble()`, `readChar()`, `readBoolean()`

- `DataOutputStream`
  - Writes Java primitive types with `writeDouble()`, `writeChar()`, `writeBoolean()`, …

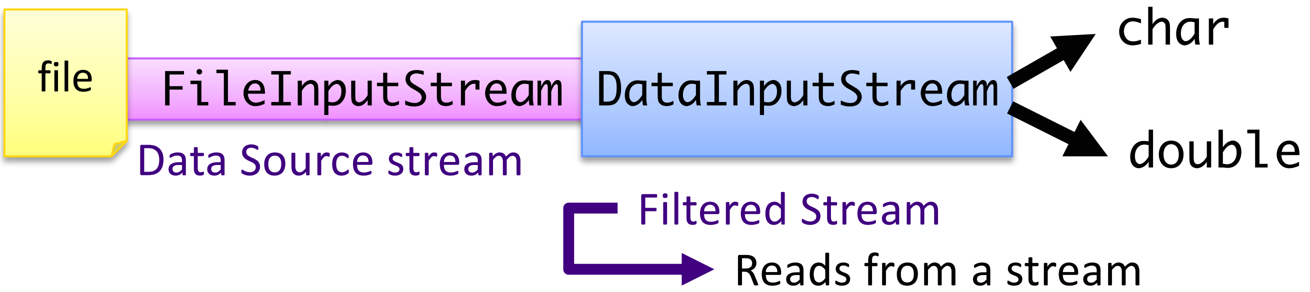
Connected Streams

Our goal: read numbers from a file

- `FileInputStream` can read from a file but has no methods to read numeric types
- `DataInputStream` can read numeric types but has no methods to read from a file
- Java allows you to **combine** two types of streams into a ***connected stream***
  - `FileInputStream` → chocolate
  - `DataInputStream` → peanut butter

Connected Streams

- Think of a stream as a pipe
- `FileInputStream` knows how to read from a file
- `DataInputStream` knows how to read an `InputStream` into useful types
- Connect **out** end of `FileInputStream` to **in** end of `DataInputStream`…



Connecting Streams

- If we want to read numbers from a file
  - FileInputStream reads bytes from file
  - DataInputStream handles numeric type reading
- Connect the DataInputStream to the FileInputStream
  - FileInputStream gets the bytes from the file and DataInputStream reads them as assembled types

```
FileInputStream fin = new FileInputStream("chicken.data");
DataInputStream din = new DataInputStream(fin);

double num1 = din.readDouble();
```

“wrap” fin in din

DataIODemo.java

Data Source vs. Filtered Streams

Data Source Streams

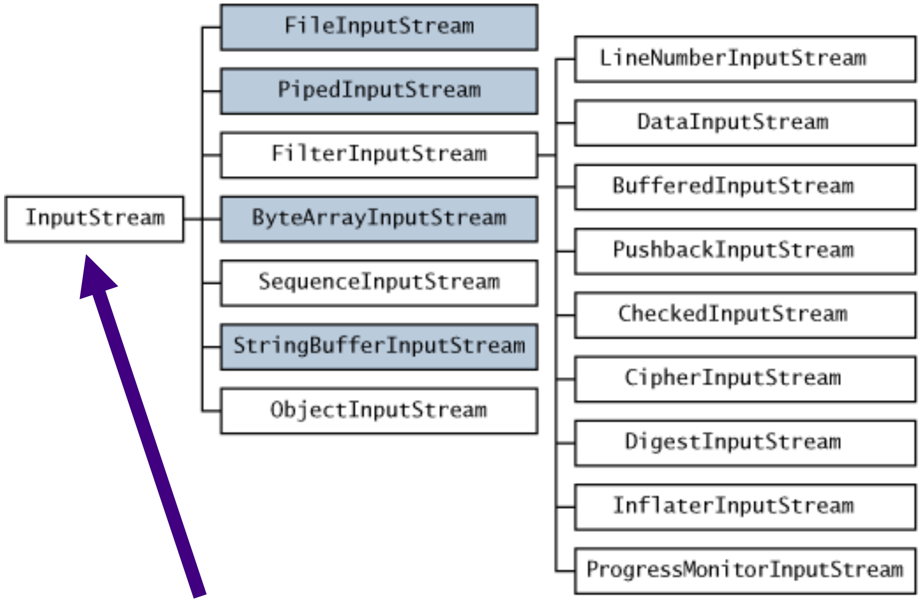
- Communicate with a data source
  - file, byte array, network socket, or URL

Filtered Streams

- Subclasses of `FilterInputStream` or `FilterOutputStream`
- Always contains/connects to another stream
- Adds functionality to other stream
  - Automatically buffered IO
  - Automatic compression
  - Automatic encryption
  - Automatic conversion between objects and bytes

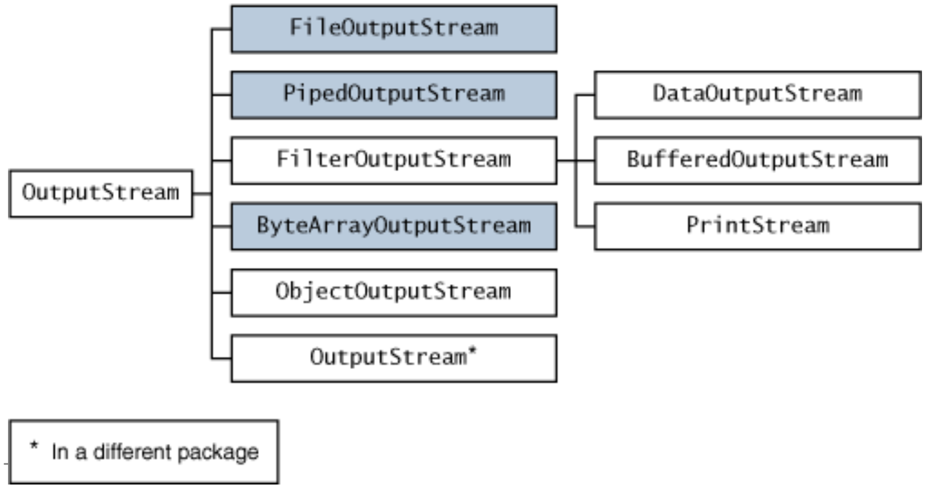
Byte Streams: For Binary Data

In `java.io` package



- InputStream
  - FileInputStream
  - PipedInputStream
  - FilterInputStream
    - LineNumberInputStream
    - DataInputStream
    - BufferedInputStream
    - PushbackInputStream
    - CheckedInputStream
    - CipherInputStream
    - DigestInputStream
    - InflaterInputStream
    - ProgressMonitorInputStream
  - ByteArrayInputStream
  - SequenceInputStream
  - StringBufferInputStream
  - ObjectInputStream

Abstract Base Classes



- OutputStream
  - FileOutputStream
  - PipedOutputStream
  - FilterOutputStream
    - DataOutputStream
    - BufferedOutputStream
    - PrintStream
  - ByteArrayOutputStream
  - ObjectOutputStream
  - OutputStream*

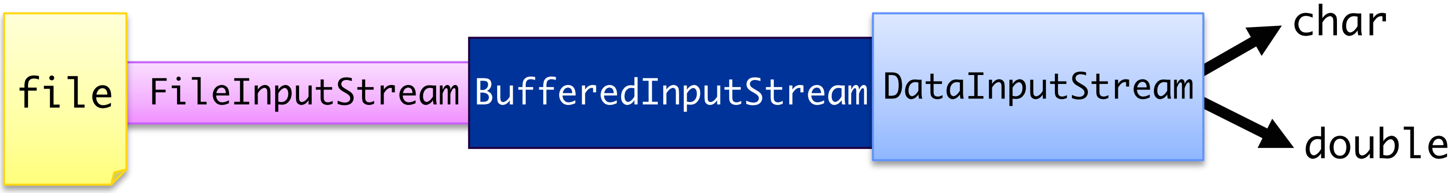
* In a different package

Shaded: Read from/write to source
White: Does some processing

Another Filtered Stream: Buffered Streams

- **`BufferedInputStream`** buffers your input streams
  - A pipe in the chain that adds *buffering* → speeds up access

```
DataInputStream din = new DataInputStream (
  new BufferedInputStream (
      new FileInputStream("chicken.data")));
```



Review: What functionality does each stream add?

Connected Streams: Similar for Output

- Example: for buffered output to the file and to write types
  - Create a `FileOutputStream`
  - Attach a `BufferedOutputStream`
  - Attach a `DataOutputStream`
  - Perform typed writing using methods of the `DataOutputStream` object

Combine different types of streams
to get functionality you want

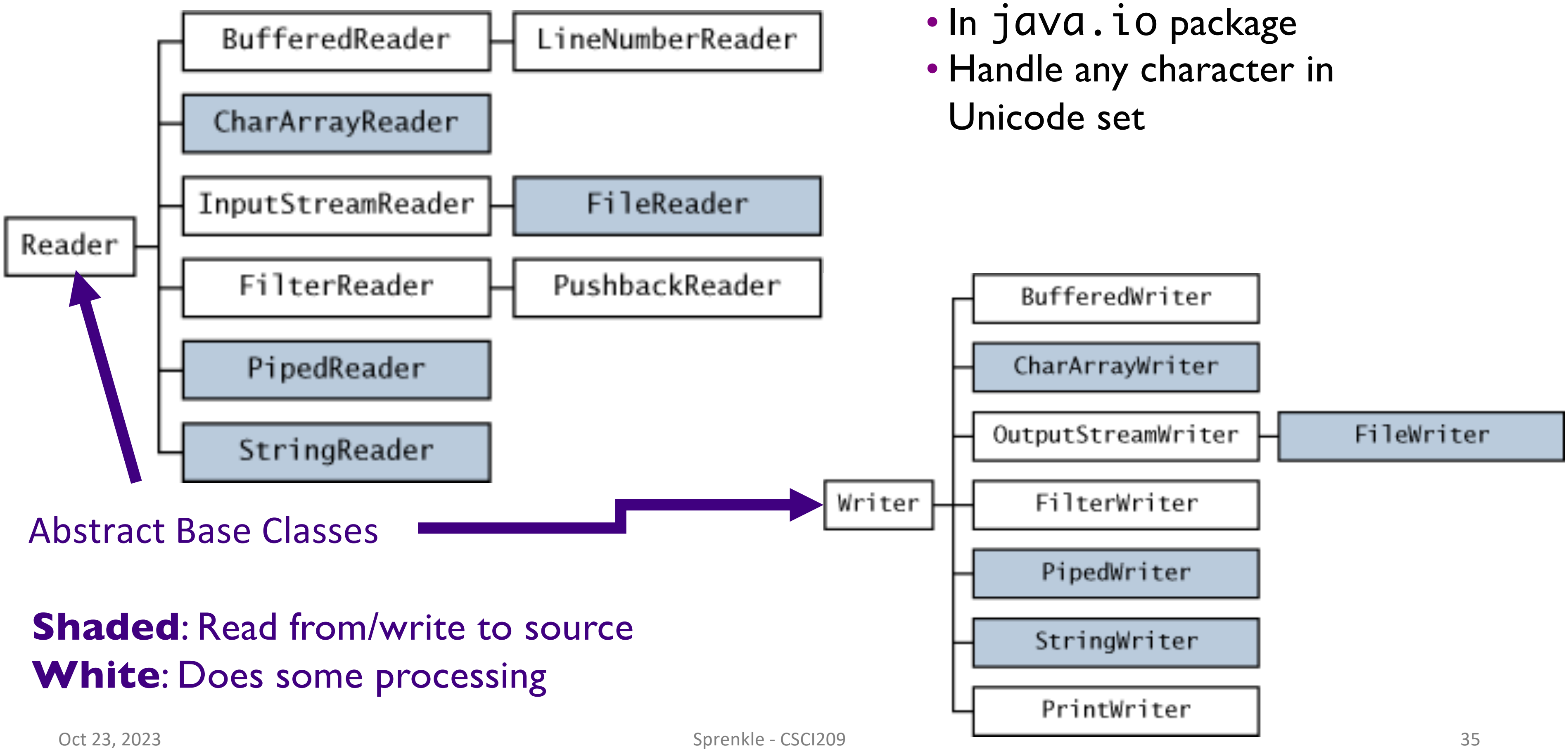
TEXT STREAMS

Text Streams

- Streams so far: operate on *binary* data, not text
- Java uses Unicode to represent characters/strings and some operating systems do not
  - Need something that converts characters from Unicode to whatever encoding the underlying operating system uses
  - Luckily, this is mostly hidden from you

Character Streams: For Text

- In `java.io` package
- Handle any character in Unicode set



Reader

- BufferedReader
  - LineNumberReader
- CharArrayReader
- InputStreamReader
  - FileReader
- FilterReader
  - PushbackReader
- PipedReader
- StringReader

Writer

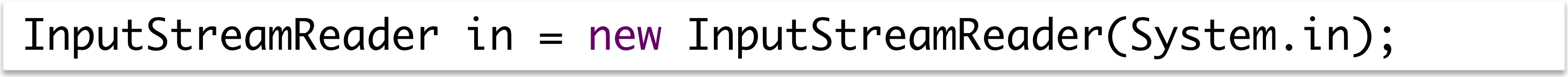
- BufferedWriter
- CharArrayWriter
- OutputStreamWriter
  - FileWriter
- FilterWriter
- PipedWriter
- StringWriter
- PrintWriter

Abstract Base Classes

Shaded: Read from/write to source
White: Does some processing

Text Streams

- Derived from **Reader** and **Writer** classes
  - Reader and Writer generally refer to **text** I/O
- Example: Make an input reader of type **InputStreamReader** that reads from keyboard



```
InputStreamReader in = new InputStreamReader(System.in);
```

  - in reads characters from keyboard and converts them into Unicode for Java

Convenience Classes: Common Combinations

- Reading and writing to text files is common
- **FileReader**
  - Convenience class *combines* a InputStreamReader with a FileInputStream
- Similar for output to text file

```
FileWriter out = new FileWriter("output.txt");
```

is equivalent to

```
OutputStreamWriter out = new OutputStreamWriter(
        new FileOutputStream("output.txt"));
```

PrintWriter

- Easiest writer to use for writing text output
- Has methods for printing various data types
  - similar to a DataOutputStream, PrintStream
- Methods: print, printf and println
  - Similar to System.out (a PrintStream) to display strings

PrintWriter Example

File to write to

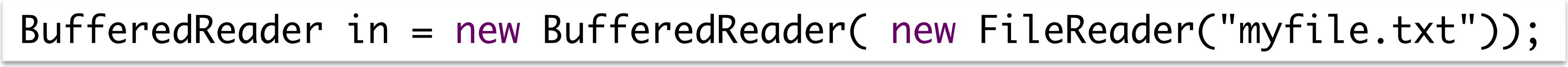
```
PrintWriter out = new PrintWriter("output.txt");

String myName = "Homer Simpson";
double mySalary = 35700;

out.print(myName);
out.print(" makes ");
out.print(salary);
out.println(" per year.");
       or
out.println(myName + " makes " + salary +
               " per year.");
```

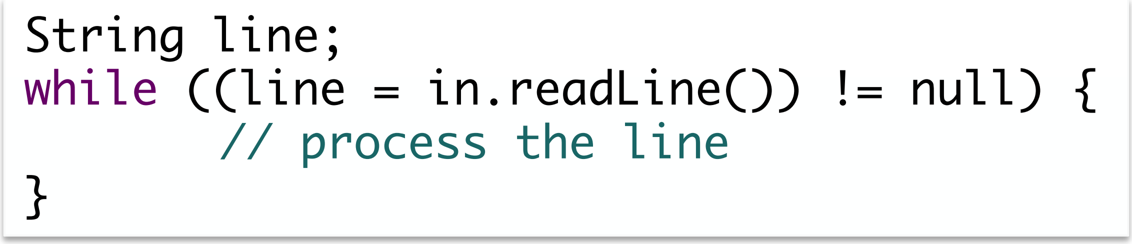
Reading Text from a Stream: `BufferedReader`

- There is no PrintReader class
- Constructor requires a `Reader` object



```
BufferedReader in = new BufferedReader( new FileReader("myfile.txt"));
```

- Read file, line-by-line using `readLine()`
  - Reads in a line of text and returns it as a `String`
  - Returns null when no more input is available



```
String line;
while ((line = in.readLine()) != null) {
        // process the line
}
```

Reading Text from a Stream

- You can attach a `BufferedReader` to an `InputStreamReader`:

```
BufferedReader consoleReader= new BufferedReader(
       new InputStreamReader(System.in));
BufferedReader webpageReader = new BufferedReader(
       new InputStreamReader(url.openStream()));
```

Note how easy it is to read from different sources

- *Used* to be the best way to read from the console

Scanners

- Scanners do not throw IOExceptions!
  - For a simple console program, main() does not have to deal with or throw IOExceptions
  - Handling those exceptions is required with BufferedReader/InputStreamReader combination
- Throws InputMismatchException when token doesn't match pattern for expected type
  - e.g., nextLong() called with next token "AAA"
  - No catching required

Meaning it is what type of exception?
How do you prevent errors in Scanner?

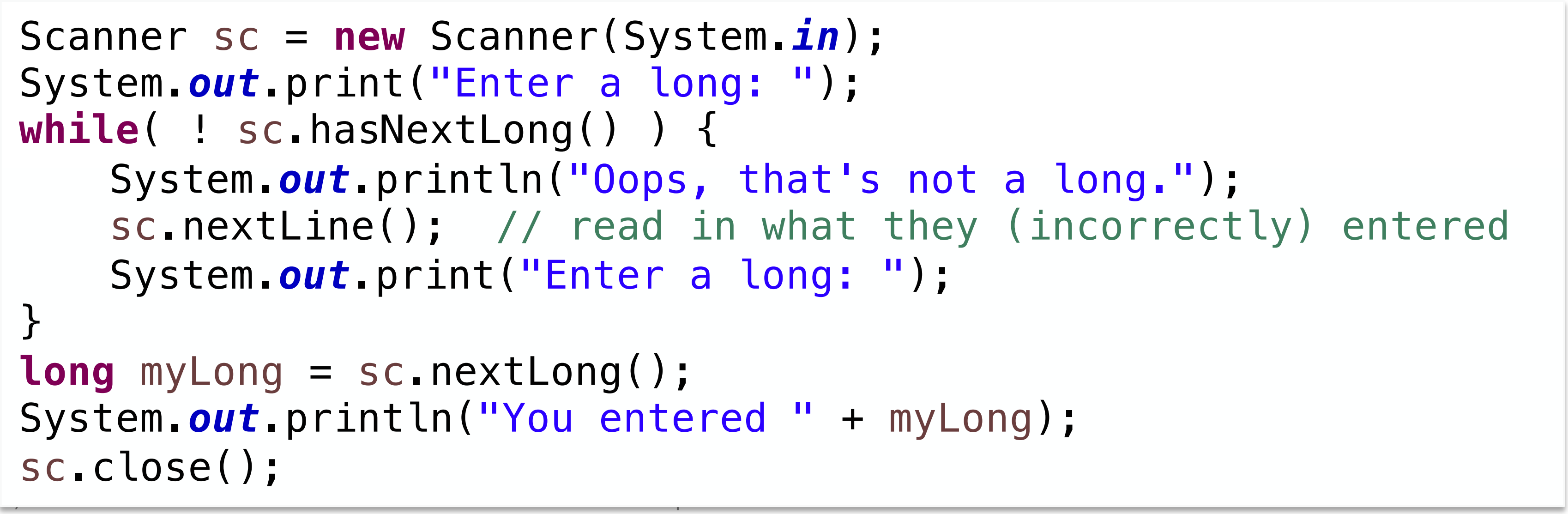
Scanners

- Scanners do not throw `IOExceptions`!
  - For a simple console program, `main`() does not have to deal with or throw `IOExceptions`
  - Handling those exceptions is required with `BufferedReader/InputStreamReader` combination
- Throws `InputMismatchException` when token doesn't match pattern for expected type
  - e.g., `nextLong`() called with next token "AAA"
  - `RuntimeException` (no catching required)

How do you prevent such errors?

Preventing Scanner Runtime Exceptions

- Methods to check before reading, e.g. hasNextLong()
- Example code excerpt



```
Scanner sc = new Scanner(System.in);
System.out.print("Enter a long: ");
while( ! sc.hasNextLong() ) {
    System.out.println("Oops, that's not a long.");
    sc.nextLine();  // read in what they (incorrectly) entered
    System.out.print("Enter a long: ");
}
long myLong = sc.nextLong();
System.out.println("You entered " + myLong);
sc.close();
```

Summary: Streams

- Abstraction: ***streams*** – sequences of data
- Two categories of classes based on type of data they handle
  - Bytes: `InputStream OutputStream`
  - Text: `Reader Writer`
- Two categories of classes based on their source
  - Data Source (primary source)
  - Filtered (another stream)

Summary: Using Streams

- Can combine streams to get the custom functionality you want
  - Convenience classes for some common combinations
- Development decisions: What do I want this stream to do?
  - What kind of data is it dealing with?
  - What filtering/functionality do I want?
- Select the streams that provide that functionality and connect them (or use convenience class)

Discussion: Stream Design Decisions

- Java’s Streams
  - Combine different types of streams to get functionality you want
  - Provide convenience classes for common functionality

What are the tradeoffs for this design decision?

- What would the alternatives be?
- Consider if you maintained the Java libraries
- Consider as a user of those Java libraries

Assignment 5

- Practicing with Eclipse
- Inheritance, Collections
- Due Monday