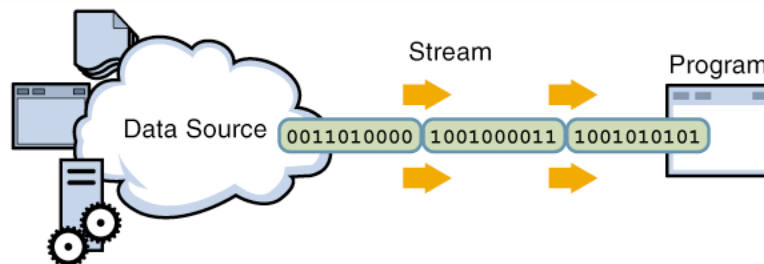
Objectives

- Streams
- Standard Error
- Compiling

1

Review: Streams

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



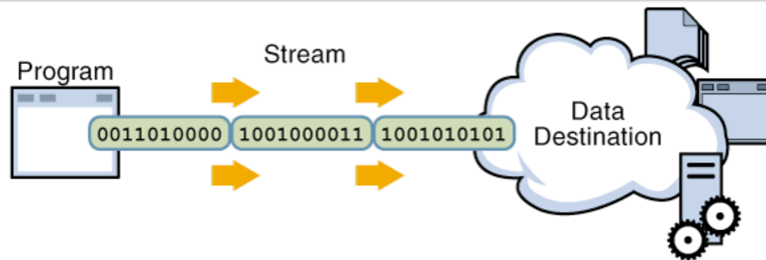
input stream: an object from which we can ***read*** a sequence of bytes

abstract class: `java.io.InputStream`

2

Review: 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`

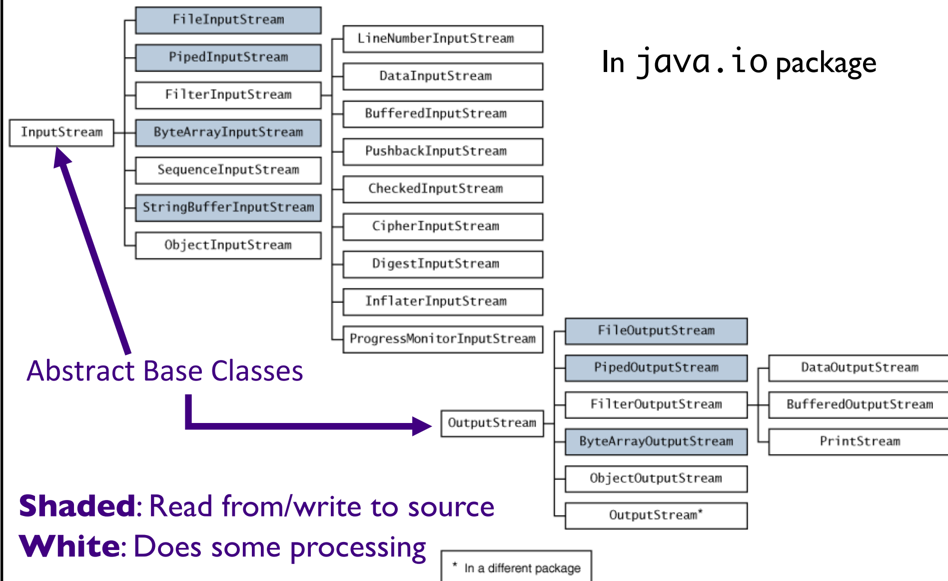
Java Streams

- 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
    - Create an `InputStream` or `OutputStream` of the appropriate type

java.io Classes Overview

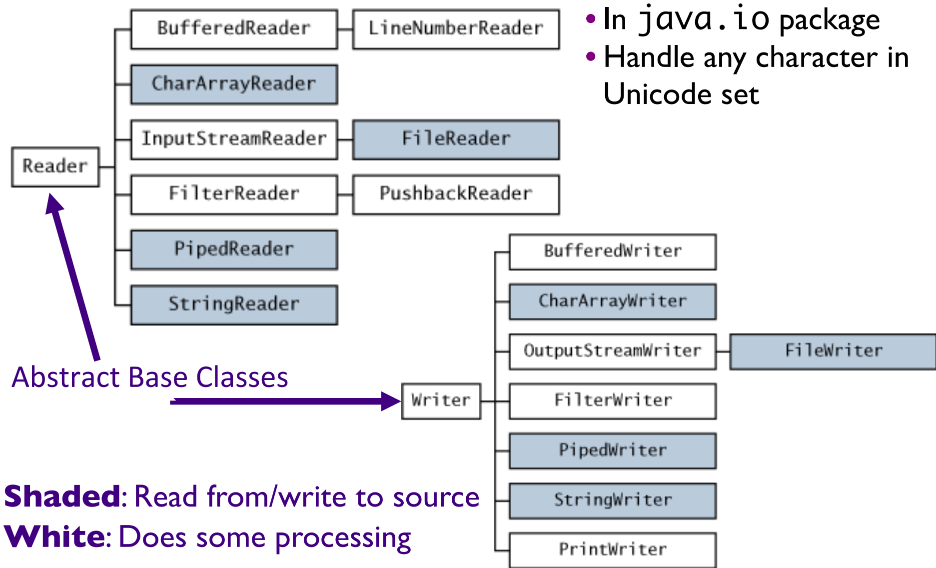
- Two categories of stream classes, based on datatype: Byte, Text
- Abstract base classes for **binary** data:
  InputStream OutputStream
- Abstract base classes for **text** data:
  Reader Writer

Byte Streams: For Binary Data



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

Character Streams: For Text



7

Console I/O

- Output:
  - `System.out` is a **`PrintStream`** object
- Input
  - `System.in` is an **`InputStream`** object
  - Throws exceptions if format of input data is not correct
    - Handle in `try/catch`

`SystemIO.java`

8

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 input stream 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:

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

  - Or, uses a **`File`** object …

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

Sept 28, 2020 Sprenkle - CSCI209 `FileTest.java` 12

12

More Powerful Stream Objects

- **`DataInputStream`**
  - Reads Java primitive types through methods such as `readDouble()`, `readChar()`, `readBoolean()`
- **`DataOutputStream`**
  - Writes Java primitive types with `writeDouble()`, `writeChar()`, …

Sept 28, 2020 Sprenkle - CSCI209 13

13

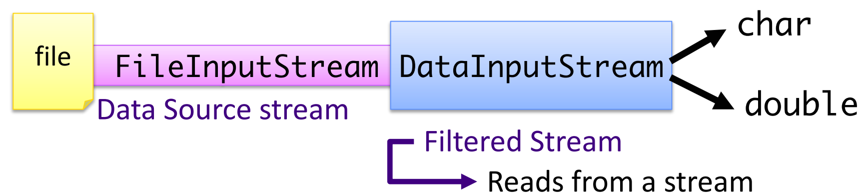
Connected Streams

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/wraps another stream
- Adds functionality to other stream, e.g.,
  - Automatically buffered IO
  - Automatic compression
  - Automatic encryption
  - Automatic conversion between objects and bytes

Buffered Streams

- Use a **`BufferedInputStream`** object to buffer your input streams
  - A pipe in the chain that adds buffering
  - Speeds up access

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

18

Connected Streams

Combine different types of streams to get functionality you want

- What are the tradeoffs for this design decision?
  - What would the alternative be?
  - Consider if you maintained the Java libraries
  - Consider as a user of those Java libraries

19

Connected Streams

Combine different types of streams
to get functionality you want

- Alternative: Creating a class for every combination would result in even more classes and a lot of redundant code
  - Consider what is required if some functionality must be updated
  - Tricky for user to pull together various streams BUT also would be hard to find the class you want that has the right combination of functionality

Connected Streams: Output

Combine different types of streams
to get functionality you want

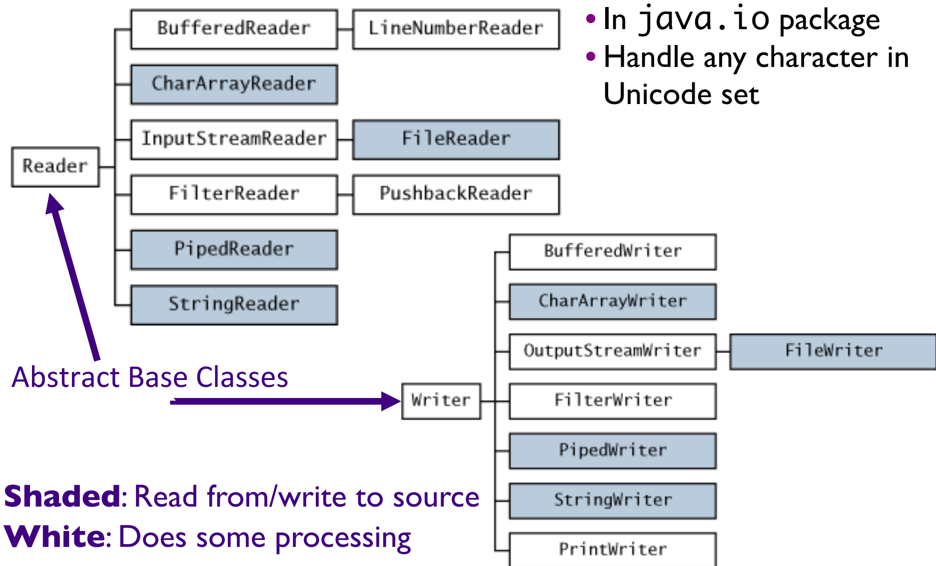
- Similar for output
  - For buffered output to the file and to write types
    - Create a `FileOutputStream`
    - Attach a `BufferedOutputStream`
    - Attach a `DataOutputStream`
    - Perform typed writing using methods of the `DataOutputStream` object

TEXT STREAMS

Text Streams

- Previous streams: operate on *binary* data
- Now: text streams!
- 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

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

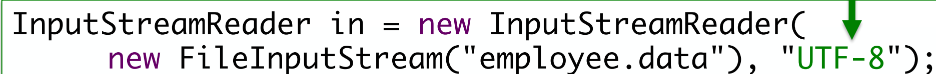
Text Streams and Encodings

- Attach an InputStreamReader to a FileInputStream

```
InputStreamReader in = new InputStreamReader(
      new FileInputStream("employee.data"));
```

  - Assumes file has been encoded in the default encoding of underlying OS

- You can specify a different *encoding* in constructor of InputStreamReader…



```
InputStreamReader in = new InputStreamReader(
      new FileInputStream("employee.data"), "UTF-8");
```


Convenience Classes

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

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

is equivalent to

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

PrintWriter

- Use for writing text output
  - Easiest writer to use
- Similar to a DataOutputStream, PrintStream → has methods for printing various data types
- 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.");
```

Review: Formatted Output

- printf or format
  - PrintStream added functionality in Java 1.5

```
double f1=3.14159, f2=1.45, total=9.43;
// simple formatting...
System.out.printf("%6.5f and %5.2f", f1, f2);
// getting fancy (%n = \n or \r\n)...
System.out.printf("%-6s%5.2f%\n", "Tax:", total);
```


Reading Text from a Stream

- There is no PrintReader class
- Use a BufferedReader
  - Constructor requires a Reader object

```
BufferedReader in = new BufferedReader(
      new FileReader("inputfile.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 also 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

Review: Scanners

- Scanners do not throw `IOExceptions`!
  - For a simple console program, `main()` does not have to deal with or throw `IOExceptions`
  - 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?

`Console` class

- Get a `Console` object using `System.console()`
- Has some useful methods for requesting passwords
- Issue: does not work through an IDE

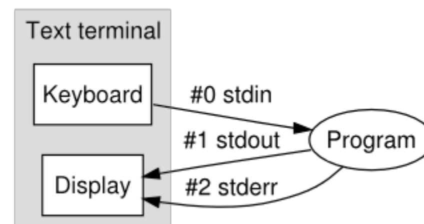
`ConsoleUsingConsoleDemo.java`

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)
- Can combine streams to get the custom functionality you want
  - Convenience classes for some common combinations

STANDARD ERROR

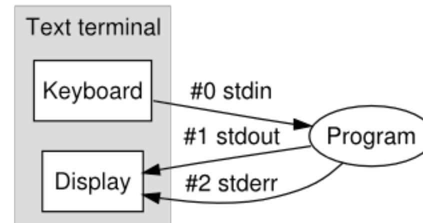
Standard Streams



- Preconnected streams
  - Standard Out: stdout
  - Standard In: stdin
  - *Standard Error: stderr*
    - For error messages and diagnostics
    - In Java: `System.err`

Benefits of two output streams (out and err)?

Standard Streams



- Preconnected streams
  - Standard Out: stdout
  - Standard In: stdin
  - *Standard Error: stderr*
    - For error messages and diagnostics
    - In Java: `System.err`
- Helpful to have separate streams for output and error messages
  - Can save outputs in two different files, e.g., error.log vs output.log
  - Eclipse (and other IDEs) differentiates between output (black text) and error (red text)

Redirecting Output

- Recall earlier this semester

```
$ java OlympicScore > score.out
```

  - Redirected `stdout` to `score.out`
  - `stderr` would still go to terminal

- To redirect `stderr` to same file as well

```
$ java OlympicScore 1> score.out 2>&1
```

`StandardStreamsExample.java`

COMPILATION

Review

- How is compiling different from interpreting?
  - What does the compiler do?

In pure forms

Summary: Compiled vs Interpreted Languages

Compiled

- – Spends a lot of time analyzing and processing the program
- • Resulting executable is some form of machine- specific binary code
- • Computer hardware interprets (executes) resulting code
- ✓ Program execution is fast
  - Efficient machine/byte code generation
  - Performance gains

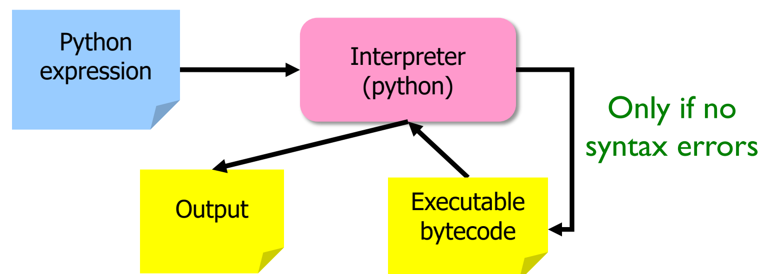
Interpreted

- ✓ Relatively little time spent analyzing and processing the program
- • Resulting code is some sort of intermediate code
- • Another program interprets resulting code
- – Program execution is relatively slow
- ✓ Faster development/prototyping

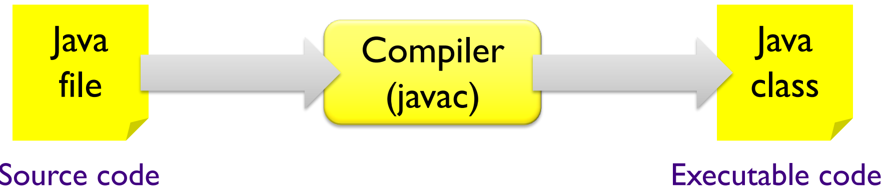
Python Interpreter

(not pure interpreting)

1. Validates Python programming language expression(s)
   - Enforces Python syntax rules
   - Reports syntax errors
2. Executes expression(s)



Java Compiler



Source code

Executable code

- Lexical analysis, parsing, semantic analysis, *code generation*, and *code optimization*
- Code optimization: dead code eliminator, inline expansion, constant propagation, …

Sept 28, 2020 Sprenkle - CSCI209 45

45

Compiling

- Translates high-level programming language to machine code or byte code
  - Java: .java → .class == bytecode
  - Holistic view of the program
- Compiler optimization techniques
  - Generate *efficient* bytecode/machine code
  - Examples: get rid of unused local variables, transform loops, inline method calls
  - In Java: static typing for additional gains
- Can execute generated code multiple times
  - Performance gain
  - Interpreted → have to re-verify the code each time executed

Sept 28, 2020 Sprenkle - CSCI209 46

46

Compiler Optimization Examples

- What is the optimization?
  - How does it make the code more efficient?
- For each optimization, should you do these optimizations yourself? Or, is it something that only the compiler should do?

Compiler Optimization Examples

Original:

```
for(int i = 0; i < 10; i++ ) {
     int j = 10;
     System.out.println(i + ", " + j);
}
```

Optimization 1

```
int j = 10;
for(int i = 0; i < 10; i++ ) {
     System.out.println(i + ", " + j);
}
```

Optimization 2

```
for(int i = 0; i < 10; i++ ) {
     System.out.println(i + ", " + 10);
}
```

Compiler Optimization Examples

```
for( int i = 0; i < 10; i++ ) {
	if( i == 0 ) {
		System.out.println("Do this");
	}
	else {
		System.out.println("Do that");
	}
}
```

```
System.out.println("Do this");

for( int i = 1; i < 10; i++ ) {
	System.out.println("Do that");
}
```

```
System.out.println("Do this");
System.out.println("Do that");
System.out.println("Do that");
System.out.println("Do that");
…
```