Objectives

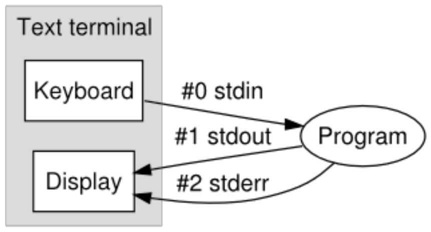
- Standard Error
- Streams
  - Byte Streams
  - Text Streams

Review

- What are benefits of exceptions
- What principle of Java do files break if we're not careful?
- What class did we use to read from standard in?
- What abstraction do we use to "do" I/O in Java?
- What are some ways to categorize streams?
  - When would you use these streams?

STANDARD ERROR

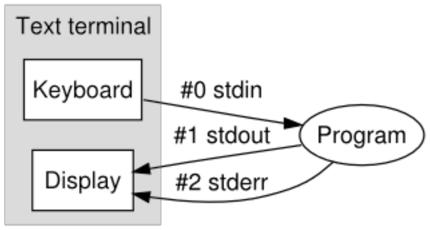
Standard Streams



- 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`

- Benefits of two output streams
  - Redirect to different places
    - Example: separate log files for info and for errors

RETURNING TO SCANNER

java.util.Scanner

- New(er) class for handling input
  - Since Java 1.5
- Many constructors
  - Read from file, input stream, string …

```
Scanner sc = new Scanner(System.in);
```

- Many methods
  - `nextXXXX` (int, long, line)
  - Skipping patterns, matching patterns, etc.

Scanners

- Breaks its input into tokens using a delimiter pattern, which matches whitespace

What is "delimiter pattern"?
What is "whitespace"?

- Converts resulting tokens into values of different types using `nextXXX`()
- Can change token delimiter from default of whitespace
- Assumes numbers are input as decimal
  - Can specify a different radix

Using Scanners

- Use *nextXXX()* to read from it…

```
long templong;

// create the scanner for the console
Scanner sc = new Scanner(System.in);

// read in an integer and a String
int i = sc.nextInt();
String restOfLine = sc.nextLine();

// read in a bunch of long integers
while (sc.hasNextLong()) {
      tempLong = sc.nextLong();
}
```

Using Scanner

Simplified version of online example

```
public static void main(String[] args) {

    // open the Scanner on the console input, System.in
    Scanner scan = new Scanner(System.in);
    scan.useDelimiter("\n"); // breaks up by lines, useful for
                             // console I/O

    System.out.print("Please enter the width of a rectangle: ");
    int width = scan.nextInt();

    System.out.print("Please enter the height of a rectangle: ");
    int length = scan.nextInt();

    System.out
       .println("The area of your square is " + length * width +
                 ".");
}
```

ConsoleUsingScannerDemo.java

Output

Read in as one token

```
This program calculates the area of a rectangle.

Please enter the width of a rectangle (as an integer):
the number is 1
Incorrect input.
Please enter the width of a rectangle (as an integer):
1 2
Incorrect input.
Please enter the width of a rectangle (as an integer):
2
Please enter the height of a rectangle (as an
integer): 3
The area of your rectangle is 6.
```

Scanners & Exceptions

- 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

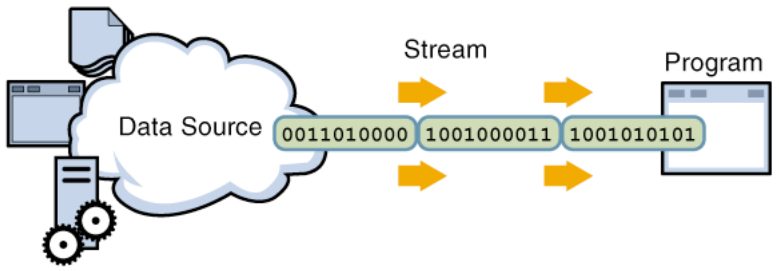
- Get a `Console` object using `System.console()`
- Has some useful methods for requesting passwords
- Issue: does not work through an IDE

`ConsoleUsingConsoleDemo.java`

STREAMS

Streams

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

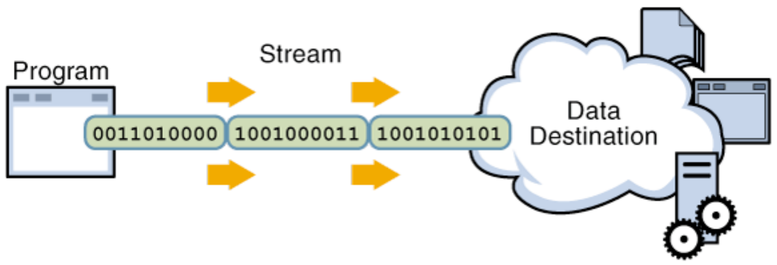


input stream: an object from which we can ***read*** a sequence of bytes

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

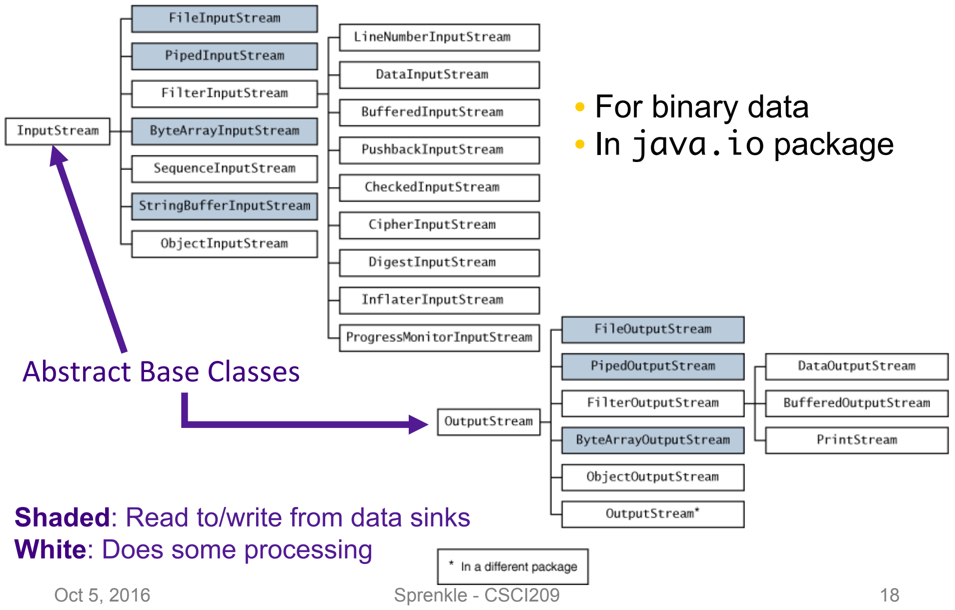
abstract class: `java.io.OutputStream`

java.io Classes Overview

- Two types of stream classes, based on type of data: Byte, Text
- Abstract base classes for binary data:
  `InputStream` `OutputStream`
- Abstract base classes for text data:
  `Reader` `Writer`

How will you know what type of data you have?

Byte Streams



- For binary data
- In `java.io` package

Shaded: Read to/write from data sinks
White: Does some processing

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);
```

`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()`, …

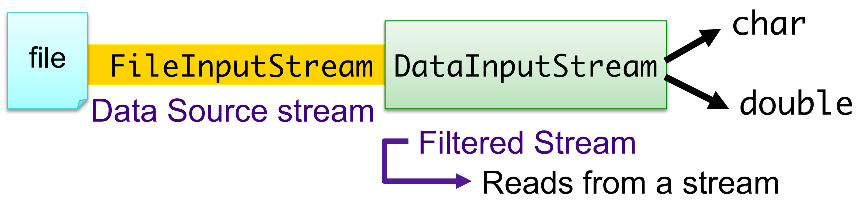
Connected Streams

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

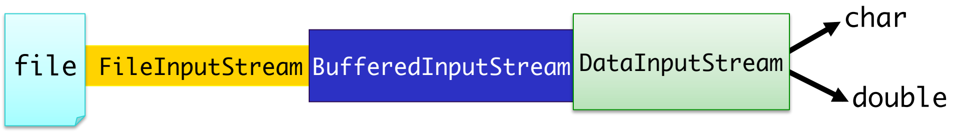
Filtered Streams

- Subclasses of `FilterInputStream` or `FilterOutputStream`
- *Always* contains another stream
- *Adds functionality* to other stream
  - 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")));
```



What functionality does each stream add?

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

Connected Streams

Combine different types of streams
to get functionality you want

- What are the tradeoffs for this design decision?

Connected Streams

Combine different types of streams
to get functionality you want

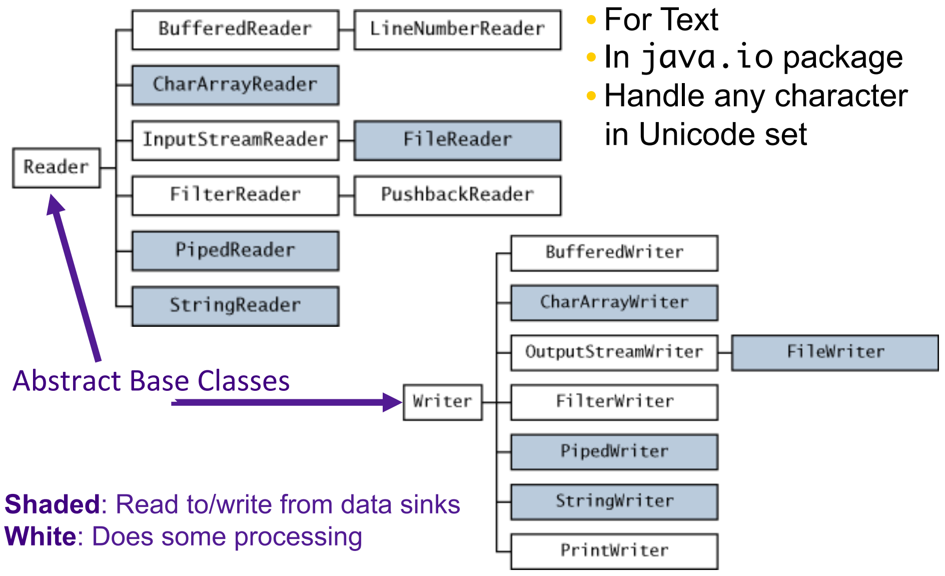
- Creating a class for every class would result in even more classes and a lot of redundant code
  - Consider what is required if some functionality must be updated

TEXT STREAMS

Text Streams

- Previous streams: 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

Shaded: Read to/write from data sinks
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

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 new functionality since 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

- 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

Practice

- Reading from and writing to a file – PetSurvey.java

Looking Ahead

- Exam: Friday
  - Faculty Rep to Board of Trustees – not available most of the day outside of class
- Assign 7: Wednesday
  - Modifying Olympic Score generator
  - Read difficulty score from console
  - Read execution scores from a file
  - Filename comes from console
- Monday
  - Java vs Python – capstone to this part of the course