Objectives

- Files
- Streams

Review

- What are the two types of *exceptions*?
- What are the two ways to deal with exceptions?
- What are the benefits of exceptions?

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 f1 = 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

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

More File Constructors

- String for the path, String for filename

```
File f2 = new File(
   "/home/courses/cs209/datafiles","chicken.data");
```

- File for directory, String for filename

```
File dir= new File("/home/courses/cs209/datafiles");
File f4 = new File(dir, "chicken.data");
```


"Break" any of Java's Principles?

Not Portable

- Accessing the file system is inherently not portable
  - In Windows, paths are "c:\\dir"
  - In Unix, paths are "/home/courses/dir"
- Relies on underlying file system/operating system to perform actions

Handling Portability Issues

- Fields in `File` class
  - `static separator`
    - Unix: "/"
    - Windows: "\\"
  - `static pathSeparator`
    - For separating a list of paths
    - Unix: ":"
    - Windows: ";"
- Use relative paths, with separators

Why two \\?

java.io.File Class

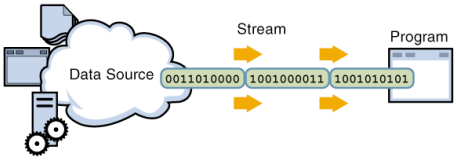
- 25+ methods
  - Manipulate files and directories
  - Creating and removing directories
  - Making, renaming, and deleting files
  - Information about file (size, last modified)
  - Creating temporary files
  - …
- See online API documentation

FileTest.java

STREAMS

Streams

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

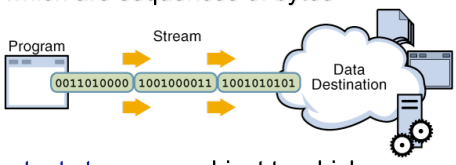


input stream: an object from which we can ***read*** a sequence of bytes

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`

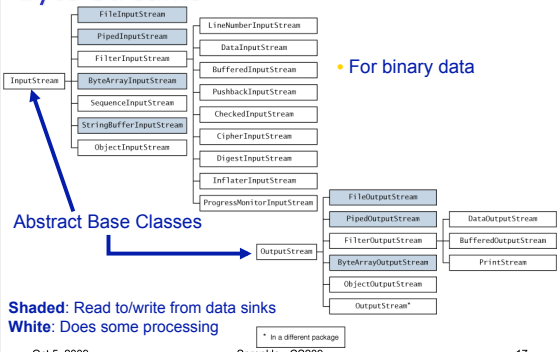
Java Streams

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

`java.io` Classes Overview

- Two types of stream classes
  - Based on datatype: Byte, Text

Byte Streams

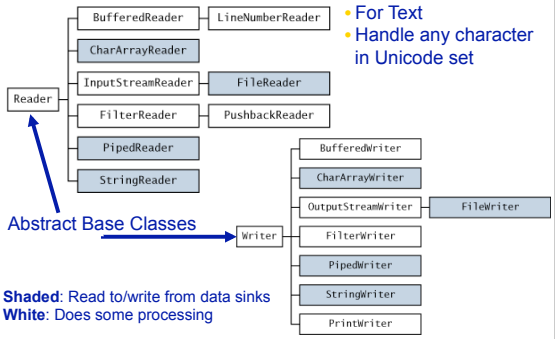


- For binary data

Abstract Base Classes

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

Character Streams



- For Text
- Handle any character in Unicode set

Abstract Base Classes

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

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`

Opening & Closing Streams

- Streams are *automatically opened* when created
- 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

Reading & Writing Bytes

- `read()` and `write()` are ***blocking*** operations
  - If a byte cannot be read from the stream, the method waits (does not return) until a byte is read
- `isAvailable()` allows you to check the number of bytes that are available for reading before calling `read()`

```
int bytesAvailable = System.in.isAvailable();
if (bytesAvailable > 0)
      System.in.read(byteBuffer);
```


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);
```


More Powerful Stream Objects

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

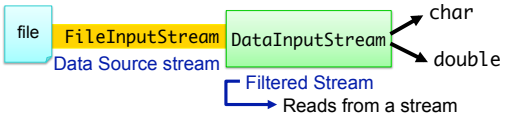
Connected Streams

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`

Data Source vs. Filtered Streams 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

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
- 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 the methods of the `DataOutputStream` object

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

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"), "ASCII");
```


Convenience Classes

- Reading and writing to text files is common
- 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.");
```


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);
```

- Can make formatted output easy
  - Before 1.5, required java.util.Formatter objects to generate String passed to System.out.println()

PrintWriters and Buffering

- PrintWriters are *always* buffered
- Option: autoflush mode
  - Causes any writes to be executed directly on target destination
    - In effect, defeats the purpose of buffering
  - Constructor with second parameter set to true

```
// create an autoflushing PrintWriter
PrintWriter out = new PrintWriter("output.txt",
                      true);
```


Reading Text from a Stream

- There is no PrintReader class
- Use a BufferedReader
  - 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 in2 = new BufferedReader(
      new InputStreamReader(System.in));
BufferedReader in3 = new BufferedReader(
      new InputStreamReader(url.openStream()));
```

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

java.util.Scanner

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

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

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

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

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

    // open the Scanner on the console input, System.in
    Scanner scan = new Scanner(System.in);

    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 +
".");
}
```

ConsoleIODemo.java

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
- Scanners are for Java 1.5 and up only

Scanners & Exceptions

- 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)

Assignment 7

- Modifying Olympic Score generator
  - Read difficulty score from console
  - Read execution scores from a file
    - Filename comes from console or command-line arguments