Software Development (cs2500)

Lectures 34 & 35: I/O and Exception Handling

M. R. C. van Dongen

January 8, 2014

Starting this Friday

- Friday lectures will be from 11 a.m. – 12 m.
- The venue will be WGB01.

Outline

- Read from and write to files.
- Learn how to read from different kinds of sources.
- Understand the concept of runtime exceptions.
- Do basic exception handling.
- Process command line arguments.

Motivation

- Most modern programs use GUIs.
- They use event-driven processes to interact with the user.
- In the olden days most programs were input-output driven.
- At the basic level information was exchanged using *files*.
  - A sequence of bytes, really.
- Files are sequential in nature.
  - You read them from beginning to end.
  - You create them from start to finish.
  - (Random access may also be possible.)
- Memory is volatile, so computers store permanent data in files.
- This makes file I/O an important topic.

The `File` Class

- Files have (path) names.
- Path names have different representations on different OSs:
  - Differences in path separators:
    - `directory/file`
    - `directory\file`
  - Differences in the root of filesystem:
    - `/`
    - `C:\\`
  - ....
- The `File` class provides abstract file/path names and operations.
- The following are some constructors:
  - `File(String parent, String child)`
  - `File(File parent, String child)`
  - `File(String pathname)`

`File` Instance Methods

- `boolean canExecute( )`
- `boolean canRead( )`
- `boolean canWrite( )`
- `boolean exists( )`
- `String getAbsolutePath( )`
- `String getName( )`
- `File getParentFile( )`
- `boolean isDirectory( )`
- `boolean isFile( )`
- `boolean isHidden( )`
- `String[] list( )`
- `File[] listFiles( )`
- `boolean mkdir( )`

File Instance Methods

- `boolean canExecute( )`
- `boolean canRead( )`
- `boolean canWrite( )`
- `boolean exists( )`
- `String getAbsolutePath( )`
- `String getName( )`
- `File getParentFile( )`
- `boolean isDirectory( )`
- `boolean isFile( )`
- `boolean isHidden( )`
- `String[] list( )`
- `File[] listFiles( )`
- `boolean mkdir( )`

File Instance Methods

- `boolean canExecute( )`
- `boolean canRead( )`
- `boolean canWrite( )`
- `boolean exists( )`
- `String getAbsolutePath( )`
- `String getName( )`
- `File getParentFile( )`
- `boolean isDirectory( )`
- `boolean isFile( )`
- `boolean isHidden( )`
- `String[] list( )`
- `File[] listFiles( )`
- `boolean mkdir( )`

File Instance Methods

- boolean canExecute( )
- boolean canRead( )
- boolean canWrite( )
- boolean exists( )
- String getAbsolutePath( )
- String getName( )
- File getParentFile( )
- boolean isDirectory( )
- boolean isFile( )
- boolean isHidden( )
- String[] list( )
- File[] listFiles( )
- boolean mkdir( )

File Instance Methods

- `boolean canExecute( )`
- `boolean canRead( )`
- `boolean canWrite( )`
- `boolean exists( )`
- `String getAbsolutePath( )`
- `String getName( )`
- `File getParentFile( )`
- `boolean isDirectory( )`
- `boolean isFile( )`
- `boolean isHidden( )`
- `String[] list( )`
- `File[] listFiles( )`
- `boolean mkdir( )`

File Instance Methods

- `boolean canExecute( )`
- `boolean canRead( )`
- `boolean canWrite( )`
- `boolean exists( )`
- `String getAbsolutePath( )`
- `String getName( )`
- `File getParentFile( )`
- `boolean isDirectory( )`
- `boolean isFile( )`
- `boolean isHidden( )`
- `String[] list( )`
- `File[] listFiles( )`
- `boolean mkdir( )`

File Instance Methods

- `boolean canExecute( )`
- `boolean canRead( )`
- `boolean canWrite( )`
- `boolean exists( )`
- `String getAbsolutePath( )`
- `String getName( )`
- `File getParentFile( )`
- `boolean isDirectory( )`
- `boolean isFile( )`
- `boolean isHidden( )`
- `String[] list( )`
- `File[] listFiles( )`
- `boolean mkdir( )`

File Instance Methods

- `boolean canExecute( )`
- `boolean canRead( )`
- `boolean canWrite( )`
- `boolean exists( )`
- `String getAbsolutePath( )`
- `String getName( )`
- `File getParentFile( )`
- `boolean isDirectory( )`
- `boolean isFile( )`
- `boolean isHidden( )`
- `String[] list( )`
- `File[] listFiles( )`
- `boolean mkdir( )`

File Instance Methods

- `boolean canExecute( )`
- `boolean canRead( )`
- `boolean canWrite( )`
- `boolean exists( )`
- `String getAbsolutePath( )`
- `String getName( )`
- `File getParentFile( )`
- `boolean isDirectory( )`
- `boolean isFile( )`
- `boolean isHidden( )`
- `String[] list( )`
- `File[] listFiles( )`
- `boolean mkdir( )`

File Instance Methods

- `boolean canExecute( )`
- `boolean canRead( )`
- `boolean canWrite( )`
- `boolean exists( )`
- `String getAbsolutePath( )`
- `String getName( )`
- `File getParentFile( )`
- `boolean isDirectory( )`
- `boolean isFile( )`
- `boolean isHidden( )`
- `String[] list( )`
- `File[] listFiles( )`
- `boolean mkdir( )`

File Instance Methods

- `boolean canExecute( )`
- `boolean canRead( )`
- `boolean canWrite( )`
- `boolean exists( )`
- `String getAbsolutePath( )`
- `String getName( )`
- `File getParentFile( )`
- `boolean isDirectory( )`
- `boolean isFile( )`
- `boolean isHidden( )`
- `String[] list( )`
- `File[] listFiles( )`
- `boolean mkdir( )`

File Instance Methods

- `boolean canExecute( )`
- `boolean canRead( )`
- `boolean canWrite( )`
- `boolean exists( )`
- `String getAbsolutePath( )`
- `String getName( )`
- `File getParentFile( )`
- `boolean isDirectory( )`
- `boolean isFile( )`
- `boolean isHidden( )`
- `String[] list( )`
- `File[] listFiles( )`
- `boolean mkdir( )`

File Instance Methods

- `boolean canExecute( )`
- `boolean canRead( )`
- `boolean canWrite( )`
- `boolean exists( )`
- `String getAbsolutePath( )`
- `String getName( )`
- `File getParentFile( )`
- `boolean isDirectory( )`
- `boolean isFile( )`
- `boolean isHidden( )`
- `String[] list( )`
- `File[] listFiles( )`
- `boolean mkdir( )`

Types of Reading

- There are several ways you can read from files.
- Difference in:

Location:	On hard disk, on a remote machine;
Access:	Sequential vs. random access;
Buffering:	Byte-by-byte versus buffered I/O.
...	

- Different classes/constructors provide different readers.

Good ol' Scanner

Basic Work Flow

- Create the `Scanner` object;
- Use the `Scanner` for sequential reading;
- Close the `Scanner`.

Good ol’ Scanner (Continued)

- Low-level `Scanner`s read from `InputStream`:
  - `final Scanner input = new Scanner( System.in );`
  - `final String word = input.next( );`
  - ...
  - `input.close( );`
- A `Scanner` can read from a `String`:
  - `final Scanner input = new Scanner( "hello world" );`
  - `final String hello = input.next( );`
  - ...
  - `input.close( );`
- A `Scanner` can also deal with a regular file:
  - ...
  - `final File file = new File( "letter.txt" );`
  - ...
  - `final Scanner input = new Scanner( file );`
  - ...
  - `input.close( );`

Oops

- When you read from file many things may go wrong:
  - You may not have read permission;
  - The file may not exist;
  - The disk crashes;
  - The file is corrupt;
  - ...
- We cannot risk programs with inpredictable outcomes.
- We must deal with all these *exceptions*.
- `Java` forces you to *handle* or *ignore* these exceptions.
- Today we shall ignore them.
  - We do this by *throwing* them with a `throws` declaration.
- We shall handle exceptions in Friday's lecture.

Throwing Exceptions

Java

```
public static void main( String[] args ) throws FileNotFoundException {
    final File file = new File( "input.txt" );
    final Scanner input = new Scanner( file ); // may fail
    final String word = input.next( );
    ...
}
```

Running the Program

- If the input file exists, all should go well.
- If not, the `jvm` will terminate the program. with the exception.

Unix Usage

```
Exception in thread "main" java.io.FileNotFoundException: input.txt
                     ... (No such file or directory)
at java.io.FileInputStream.open(Native Method)
at java.io.FileInputStream.<init>(FileInputStream.java:138)
at java.util.Scanner.<init>(Scanner.java:656)
at Main.main(tmp.java:8)
```

Adding it Up

Java

```
import java.io.*;
import java.util.Scanner;

public class Adder {
    private static final String FORMAT = "%10s%8.2f\ n";
    private static final String TOTAL  = "Total:";
    public static void main( String[] args ) throws FileNotFoundException {
        final File file = new File( "input.txt" );
        final Scanner in = new Scanner( file );

        double total = 0.0;
        while (in.hasNextDouble( )) {
            final double next = in.nextDouble( );
            total += next;
            System.out.printf( FORMAT, "", next );
        }
        in.close( ); // always close the scanner
        System.out.printf( FORMAT, TOTAL, total );
    }
}
```

Running the Program

Unix Session

```
$
```

Running the Program

Unix Session

```
$ cat input.txt
```

Running the Program

Unix Session

```
$ cat input.txt
1.20
 2.30
3.40 4.50
$
```

Running the Program

Unix Session

```
$ cat input.txt
1.20
 2.30
3.40 4.50
$ java Main
```

Running the Program

Unix Session

```
$ cat input.txt
1.20
 2.30
3.40 4.50
$ java Main
          1.20
          2.30
          3.40
          4.50
Total:   11.40
$
```

Introduction

- Needless to say, writing to files is also important.
- This time we use an object that writes to the file.
- `Java` has many file writer classes.
  - The differences are similar to differences with reader classes.
- This time the work flow is:
  - Create the writer object;
  - Use the object to write the file;
  - Close the writer object.
- Closing the object is important:
  - Other processes can’t use the file util the writer is closed.
  - If the program terminates without closing, data may be lost.

The `PrintWriter` Class

- The `PrintWriter` class provides an easy API for writing files.
- You create the `PrintWriter` object:
  - `final File file = new File( "output.txt" );`
  - `final Printwriter out = new PrintWriter( file );`
- You use the `PrintWriter` object to write to the file:
  - `out.println( "important stuff" );`
- You close the `PrintWriter` object:
  - `out.close( );`

Example

Java

```
private static final String FORMAT = "%10s%8.2f\ n";
private static final String TOTAL  = "Total:";
public static void main( String[] args ) throws FileNotFoundException {
    final File inputFile = new File( "input.txt" );
    final File outputFile = new File( "output.txt" );
    final Scanner in = new Scanner( inputFile );
    final PrintWriter out = new PrintWriter( outputFile );

    double total = 0.0;
    while (in.hasNextDouble( )) {
        final double next = in.nextDouble( );
        total += next;
        out.printf( FORMAT, "", next );
    }
    out.printf( FORMAT, TOTAL, total );
    in.close( );
    out.close( );
}
```

Reading Lines

- Many programs use text-based database tables.
- Each line is a record.
- Each record is a delimiter-separated sequence of values.
- E.g. `Unix` password file:

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
...
```

Reading Lines (Continued)

- `nextLine( )` read the `Scanner`'s next line.
  - `final Scanner input = ...;`
  - `String line = input.nextLine( );`
- Simplifies processing.

Java

```
while (tableScanner.hasNext( )) {
    final String record = tableScanner.nextLine( );
    final Scanner recordScanner = new Scanner( record );
    while (recordScanner.hasNext( )) {
        final String next = recordScanner.next( );
        ...
    }
}
```

Reading Words

- A Scanner splits its input into words.
- The words in the input are separated by *delimiter sequences*.
- By default a Scanner uses whitespace sequences as delimiter.
  - Whitespaces are normal spaces, tabs, and newlines.
- The next( ) method returns the next word from the Scanner's input.
  - `final String word = input.next( );`

Scanner Delimiters

- The `useDelimiter( )` method changes the `Scanner`'s delimiter.
- `useDelimiter( )` method takes one argument that specifies the delimiter.
- The argument is based on a *regular expression*.
- They can describe structured text, for example:
  - A single space: `" "`;
  - A sequence of one or more spaces: `"␣+"`;
  - A sequence of spaces: `"␣+"`;
  - A sequence of whitespace characters: `"\\\\s+"`;
  - A single colon: `":"`;
  - A colon or a semicolon: `"[:;]"`;
  - A letter: `"[a-ZA-Z]"`;
  - A sequence of letters: `"[a-ZA-Z]+"`;
  - A word: `"\<[a-ZA-Z]+\>"`;
  - A word at the start of the line: `"$[a-ZA-Z]+\>"`;
  - ....

Reading Characters

- An empty delimiter lets you read one character at a time.
- `input.setDelimiter( "" );`
- `String next = input.next( );`
- `char c = next.charAt( 0 );`

Classifying Characters

- Many applications require character classification:
  - Is the character whitespace?
  - Is the character a digit?
  - Is the character a letter?
  - Is the character a lowercase letter?
  - ...
- The `Character` class provides class functions for doing this:
  - `isWhiteSpace( )`
  - `isDigit( )`
  - `isLetter( )`
  - `isLowercase( )`

Implementing isDigit( )

Java

```
public static boolean isDigit( final char character ) {

}
```

Implementing `isDigit( )`

Java

```
public static boolean isDigit( final char character ) {
    return ('0' <= character) && (character <= '9');
}
```

Converting a `String` to `int`

- Converting `String`s to numbers is a common task.
- The `Integer` class provides class methods to do this:
  - `parseInt( )`
  - `parseLong( )`
  - `parseDouble( )`
  - ...

Converting a String to int (Continued)

Java

```
private static final int NUMBER_BASE = 10;
private static final int MINUS_SIGN = '-';
private static final int UNICODE_VALUE_OF_ZERO = (int)'0';

/**
 * Convert a {@code String} to an {@code int}.
 * <p>The {@code String} should represents a valid {@code int}.
 * It may contain an optional '-' at the start.
 * It shouldn't contain spaces.</p>
 *
 * @param string The input {@code String}.
 * @return The {@code int} representation of {@code string}.
 */
public static int parseInt( final String string ) {
    final boolean isNegative = string.charAt( 0 ) == MINUS_SIGN;
    int nextDigitPosition = isNegative ? 1 : 0;
    int result = 0;

    while (nextDigitPosition != string.length( )) {
        final char nextChar = string.charAt( nextDigitPosition++ );
        final int nextContribution = ((int)nextChar) - ZERO;
        result = (result * NUMBER_BASE) + nextContribution;
    }

    return isNegative ? -result : result;
}
```

What Happens when a Method you Call is Risky?

- Let's say you call a method/constructor.

Java

```
final File file = new File( "skblzzz" );
final Scanner scanner = new Scanner( file );
```

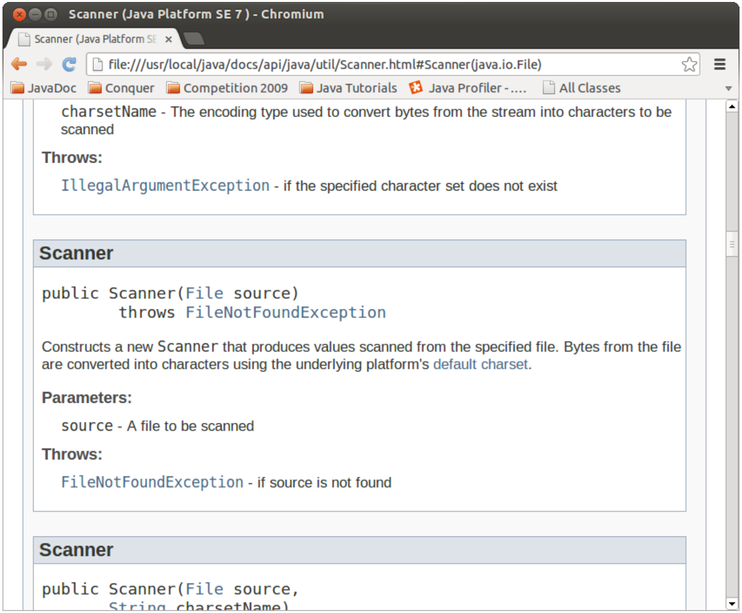
- The method does something risky: may not work at runtime.

Java

```
public Scanner( final File file ) {
   if (!file.exist( )) {
       System.explode( );  // runtime error
   } else {
       ⟨Construct Scanner⟩ // grand
   }
}
```

- You need to know the method you're calling is risky.
- You then write code that *catches* errors if they happen.
- The result is a safe and robust application.

Finding Risky Code


Scanner (Java Platform SE 7) - Chromium
Scanner (Java Platform SE
file:///usr/local/java/docs/api/java/util/Scanner.html#Scanner(java.io.File)
JavaDoc
Conquer
Competition 2009
Java Tutorials
Java Profiler - ....
All Classes

charsetName - The encoding type used to convert bytes from the stream into characters to be scanned

Throws:

IllegalArgumentException - if the specified character set does not exist

Scanner

```
public Scanner(File source)
        throws FileNotFoundException
```

Constructs a new Scanner that produces values scanned from the specified file. Bytes from the file are converted into characters using the underlying platform's default charset.

Parameters:

source - A file to be scanned

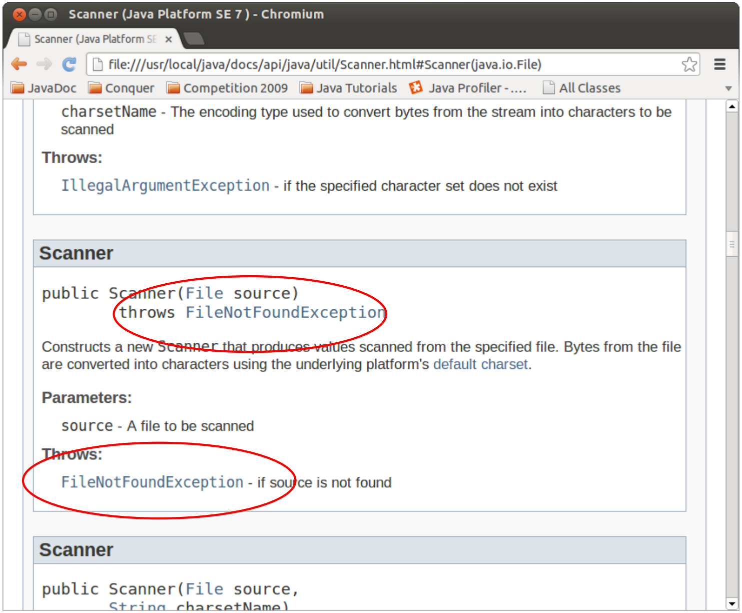
Throws:

FileNotFoundException - if source is not found

Scanner

```
public Scanner(File source,
        String charsetName)
```

Finding Risky Code


Scanner (Java Platform SE 7) - Chromium
Scanner (Java Platform SE
file:///usr/local/java/docs/api/java/util/Scanner.html#Scanner(java.io.File)
JavaDoc
Conquer
Competition 2009
Java Tutorials
Java Profiler - ....
All Classes

charsetName - The encoding type used to convert bytes from the stream into characters to be scanned

Throws:

IllegalArgumentException - if the specified character set does not exist

Scanner

```
public Scanner(File source)
        throws FileNotFoundException
```

Constructs a new Scanner that produces values scanned from the specified file. Bytes from the file are converted into characters using the underlying platform's default charset.

Parameters:

source - A file to be scanned

Throws:

FileNotFoundException - if source is not found

Scanner

```
public Scanner(File source,
        String charsetName)
```

Exceptions

- *An exception* informs caller when something bad has happened.
- Caller must *catch* the exception or *ignore* it.

Catching	Means acknowledging with the exception.
Ignoring	Passing the exception on.

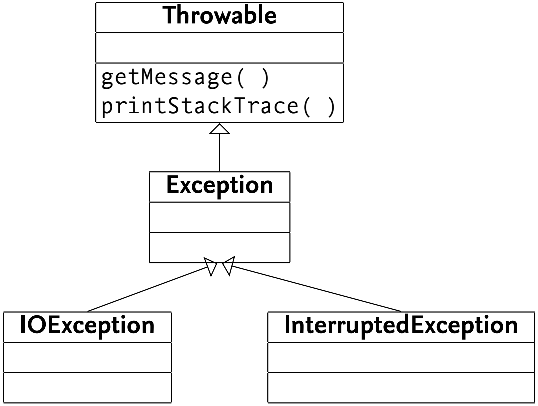
  - This is done by *declaring* the exception.
  - Ignoring an exception doesn't solve it.
  - Eventually, some code must catch the exception.

Try the Method and Catch the Exceptions

Java

```
public Scanner createSCanner( ) {
    Scanner scanner = null;
    try {
        final File file = new File( "skblzzz" );
        scanner = new Scanner( file );
    } catch( FileNotFoundException exception ) {
        System.err.println( "MyClass: scanner creation failed!" );
        ...
    }
}
```

Exceptions are Objects



Creating New Exceptions

Java

```
public class MotherOfAllExceptions extends Exception {
    ...
}
```

Properly Dealing with Exceptions

Java

```
public void handleException( Exception exception ) {
    final String cause = exception.getMessage( );
    if (cause != null) {
        System.err.println( cause );
    }
    exception.printStackTrace( );
    // call System.exit( exitStatus ) if you want to terminate the application.
}
```

Properly Dealing with Exceptions (Continued)

Java

```
public void safe( ) {
    try {
        risky( );
    } catch (FatherOfAllExceptions exception) {
        handleException( exception );
    } catch (MotherOfAllExceptions exception) {
        handleException( exception );
    }
}
```

Output

Unix Session

```
$
```

Output

Unix Session

```
$ java Risky
```

Output

Unix Session

```
$ java Risky
MotherOfAllExceptions
        at Risky.risky(Risky.java:10)
        at Risky.safe(Risky.java:26)
        at Risky.main(Risky.java:4)
$
```

Throwing Exceptions

Uses First Matching Exception from Top

Java

```
public void risky( ) throws MotherOfAllExceptions,
                            FatherOfAllExceptions {
    if (allFails( )) {
        throw new MotherOfAllExceptions( );
    } else if (stillDesperate( )) {
        throw new FatherOfAllExceptions( );
    }
}
```

Ignoring Exceptions

Java

```
public void safeIsh( ) throws FatherOfAllExceptions {
    try {
       risky( );
    } catch (MotherOfAllExceptions exception) {
       // Deal with it.
    }
}
```

Finally

Java

```java
final Oven oven = new Oven( );
try {
    oven.on( );
    Dish dish = new Dish( );
    dish.bake( );
} catch (BakingException exception) {
    exception.printStackTrace( );
} finally {
    oven.off( );
}
```

Old-Style Try-Catch Block

Java

```
import java.io.FileWriter;

public class WriteFile {
    public static void main( String[] args ) {
        PrintWriter writer = null; // cannot make it final
        try {
            writer = new PrintWriter( "output.txt" );
            writer.println( "My first line of text." );
            writer.println( "My second line of text?" );
        } catch( FileNotFoundException exception ) {
            // handle exception
        } finally {
            writer.close( );
        }
    }
}
```

Modern-Style Try-with-Resources Block

Class must Implement AutoClosable Interface

Java

```
import java.io.FileWriter;

public class WriteFile {
    public static void main( String[] args ) {
        try ( final PrintWriter writer = new PrintWriter( "output.txt" ); ) {
            writer.println( "My first line of text." );
            writer.println( "My second line of text?" );
        } catch( FileNotFoundException exception ) {
            // handle exception
        }
    }
}
```

Command Line Arguments

- Many `Unix` commands take one or more arguments.
  - Regular arguments:
    - `echo Hello world!`
  - Flags:
    - `ls -l`
  - Application/os-specific “stuff”:
    - `sort < input.txt > output.txt`
- `Java` programs may also take command line parameters.
  - The os passes them to the JVM.
  - This is done in an array of `String`.
  - The JVM passes the array to the `main`.

Case Study

Imports

Java

```
import java.io.*;
import java.util.Scanner;
```

Case Study (Continued)

Class Constants

Java

```
public class NumberAdder {
    private static final String DEFAULT_OUTPUT = "output.txt";
    private static final String DEFAULT_INPUT = "input.txt";
    private static final String OUTPUT_FLAG = "-o";
    private static final String INPUT_FLAG = "-i";
    private static final String USAGE
        = "java NumberAdder [" + INPUT_FLAG + " input]"
                        + " [" + OUTPUT_FLAG + " output]";

    private static final String FORMAT = "%10s%8.2f\ n";
    private static final String TOTAL  = "Total:";
    private static final int ERROR_EXIT_STATUS = 1;

    ...
}
```

Case Study (Continued)

Instance Variables and Constructor

Java

```
private final File outputFile;
private final File inputFile;

private NumberAdder( final String output, final String input ) {
    this.outputFile = new File( output );
    this.inputFile = new File( input );
}
```

Case Study (Continued)

The `main( )`

Java

```
public static void main( String[] args ) {
    final NumberAdder parameters = parseArguments( args );

    if ((parameters == null) || (!parameters.inputFile.exists( ))) {
        usage( );
    } else {
        parameters.process( );
    }
}
```

Case Study (Continued)

The process( ) Method

Java

```
private void process( ) throws FileNotFoundException {
    final Scanner in = new Scanner( inputFile );
    final PrintWriter out = new PrintWriter( outputFile );

    double total = 0;
    while (in.hasNextDouble( )) {
        final double next = in.nextDouble( );
        total += next;
        out.printf( FORMAT, "", next );
    }
    out.printf( FORMAT, TOTAL, total );
    in.close( );
    out.close( );
}
```

Case Study (Continued)

The `usage( )` Method

Java

```
private static void usage( ) {
    System.err.println( USAGE );
    System.exit( ERROR_EXIT_STATUS );
}
```

Case Study (Continued)

The parseArguments( ) Method

Java

```java
private static NumberAdder parseArguments( final String[] args ) {
    String input = DEFAULT_INPUT;
    String output = DEFAULT_OUTPUT;
    boolean error = false;

    int position = 0;
    while ((!error) && (position != args.length)) {
        final String candidateFlag = args[ position++ ];
        if (position == args.length) {
            error = true;
        } else if (candidateFlag.equals( INPUT_FLAG )) {
            input = args[ position++ ];
        } else if (candidateFlag.equals( OUTPUT_FLAG )) {
            output = args[ position++ ];
        } else {
            error = true;
        }
    }

    return error ? null : new NumberAdder( output, input );
}
```

For Monday

- Study [Horstmann 2013, Chapter 12].

Acknowledgements

- This lecture corresponds to [Horstmann 2013, Chapter 12].

About this Document

- This document was created with `pdflatex`.
- The LaTeX document class is `beamer`.