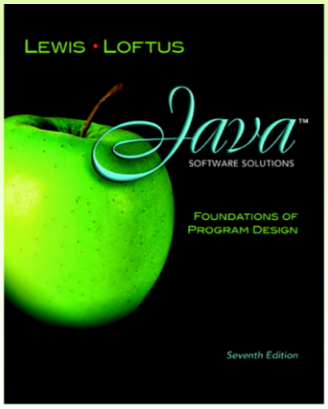
Chapter 11
Exceptions



Java Software Solutions
Foundations of Program Design
Seventh Edition

John Lewis
William Loftus

Addison-Wesley
is an imprint of
PEARSON

-In addition to Exceptions, we also take the time to learn about basic input/output (I/O) processing in Java

-We look at this topic now due to the fact that I/O processing can often create run-time errors or exceptions

-In our study of I/O, we'll first look at the topic of streams which form the foundation for all I/O processing

-We'll then look at Java API classes to perform basic file I/O operations such as reading and writing to files

-Since I/O tasks can often create exceptions, we'll then look at how to handle such exceptions from I/O

Exceptions

- Exception handling is an important aspect of object-oriented design
- Chapter 11 focuses on:
  - the purpose of exceptions
  - exception messages
  - the try-catch statement
  - propagating exceptions
  - the exception class hierarchy
  - GUI mnemonics and tool tips
  - more GUI components and containers

Outline

- Exception Handling
- The try-catch Statement
- Exception Classes
- I/O Streams
- File I/O
- I/O Exceptions

I/O Streams

- Input and output can be viewed as data (bytes) moving between a source and a destination
- Data flowing *in* to a device describes an **input** operation (such as bytes moving from a file to memory)
- Data flowing *out* of a device describes an **output** operation (such as bytes moving from memory to a file)
- We can describe this movement or flow of data as a **stream**
- A *stream* is a sequence of bytes that flow from a source to a destination

I/O Streams

- Streams can flow between a wide variety of different sources and destinations including such devices as a keyboard, disk, memory, or network
- We often find the need to write programs that manage such streams of data
- We may need to **read** a file from disk and load the contents into memory we've reserved (e.g. in class objects)
- We might want to **write** the contents in memory (e.g. class objects) to a file on disk
- In a program, we **read** information from an **input** stream and **write** information to an **output** stream
- A program can manage multiple streams simultaneously

I/O Streams

- Since a stream is a noun, it is ideally described by a class
- A stream “knows” about data and “does” data flow between devices
- There are many different “types of” streams with different characteristics
- Different streams work with data in files, memory, or strings
- Different streams assume data is in the form of characters or raw bytes (binary information)
- Different streams allow the data to be stored temporarily in buffers to speed-up reading or writing

I/O Streams

- Each different “type of” stream can be described with a different class
- The Java API provides an extensive class hierarchy describing many different types of streams
- These classes are part of the `java.io` package

I/O Streams

- We've actually been using objects from some of the Java I/O stream classes in our programs without knowing it
- Consider, for example, the following statement:

```
System.out.println("Hello World");
```

- `System` is a Java class encapsulating aspects of the Java run-time environment
- `out` is a static object reference variable from the `PrintStream` class which is one of the classes in the Java I/O stream hierarchy
- `println` is a method from the `PrintStream` class to print and terminate with a new line

I/O Streams

- Since `out` is a static object reference variable in the `System` class, we call methods from the `PrintStream` class as we've been doing

```
System.out.println("Hello World");
```

- In the same way, we've used another static object reference variable, `in`, from the `System` class to create a `Scanner` object to read input from the keyboard:

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

- We refer to `out`, `in`, and another object reference variable, `err`, as **standard I/O streams**
- These I/O streams are associated with devices (e.g. console output window, keyboard)

I/O Streams

- There are three standard I/O streams:
  - *standard output* – defined by `System.out`
  - *standard input* – defined by `System.in`
  - *standard error* – defined by `System.err`
- We use `System.out` when we execute `println` statements
- `System.out` and `System.err` typically represent the console window
- `System.in` typically represents keyboard input, which we've used many times with `Scanner`

File I/O

- Oftentimes, we need to work with streams of characters flowing in and out of files on disk
- For these purposes, we can use the `FileReader` and `FileWriter` classes from the Java I/O class hierarchy
- We often use a combination of I/O classes to make our processing more efficient
- Some classes help by **buffering** data within a stream
- Buffering stores stream data in temporary memory areas (called buffers) instead of accessing data continually from I/O devices
- Other classes provide familiar methods (such as `println`) to aid in reading and writing stream data

File I/O

- Let's look at an example that uses a combination of I/O classes to write and read a string to and from a file
- Example `FileIO.java`

File I/O

```
public class FileIO {

    public static void main(String[] args) throws IOException {

        // Open file for writing
        FileWriter fw = new FileWriter("test.txt");
        BufferedWriter bw = new BufferedWriter(fw);
        PrintWriter pw = new PrintWriter(bw);

        // Write a string
        pw.println("Hello world");

        // Close file stream
        pw.close();

        // Open file for reading
        FileReader fr = new FileReader("test.txt");
        BufferedReader br = new BufferedReader(fr);

        // Read a string
        Scanner scan = new Scanner(br);
        String str = scan.next();
        System.out.println("Read string: " + str);

        // Close file stream
        br.close();
    }
}
```

-Note how we use I/O classes FileWriter, BufferedWriter, and PrintWriter to write a string to a file

-Note how use use I/O classes FileReader and BufferedRead to read a string from a file

-Note how we use the Scanner class with the file input stream instead of the keyboard (System.in)

-Note how we must close streams opened from File I/O classes

-Note the throws clause added in the main method header

-Recall that this is necessary when methods use classes that can throw **checked** exceptions

-The I/O classes we are using can potentially throw **checked** exceptions

-Instead of handling them in our method, we just include the throw clause

-This program will terminate abnormally if an I/O exception occurs because we are not handling them!

-Let's look specifically at handling I/O exceptions next

I/O Exceptions

- Many run-time errors (exceptions) can occur when working with I/O classes (e.g. file may not exist)
- I/O exceptions throw **checked** exceptions
- Recall that with checked exceptions, we must either include a throw clause or handle them directly in our program
- In our previous example, we simply include the throw clause
- Instead, however, programs should handle all the potential I/O exceptions when working with I/O classes

I/O Exceptions

- Let's return to our previous example and add code to our main method in order to handle I/O Exceptions
- Example `FileIOException1.java`

I/O Exceptions

```
public class FileIOException1 {

    public static void main(String[] args) {

        try
        {
            // Open file for writing
            FileWriter fw = new FileWriter("test.txt");
            BufferedWriter bw = new BufferedWriter(fw);
            PrintWriter pw = new PrintWriter(bw);

            // Write a string
            pw.println("Hello world");

            // Close file stream
            pw.close();
        }
        catch(IOException e)
        {
            System.err.println("I/O EXCEPTION DURING WRITING: " + e.getMessage());
        }
```

-Note how if we remove the throw clause, we must use try-catch blocks to handle I/O Exceptions

-We wrap file open and writing operations with a try-catch block

-Note we use the standard I/O stream, err, to print our exception message

-This prints text in a red color in the Eclipse Console window

-Note we use a method of the Exception class (getMessage) when printing the message

I/O Exceptions

```
        try
        {
            // Open file for reading
            FileReader fr = new FileReader("test.txt");
            BufferedReader br = new BufferedReader(fr);

            // Read a string
            Scanner scan = new Scanner(br);
            String str = scan.next();
            System.out.println("Read string: " + str);

            // Close file stream
            br.close();
        }
        catch(IOException e)
        {
            System.err.println("I/O EXCEPTION DURING READING: " + e.getMessage());
        }
    }
}
```

-In a similar way, we wrap file open and reading operations with a try-catch block

I/O Exceptions

- We can also wrap both reading and writing operations with a single try-catch block
- Example `FileIOException2.java`

I/O Exceptions

```
public class FileIOException2 {

    public static void main(String[] args) {

        try
        {
            // Open file for writing
            FileWriter fw = new FileWriter("test.txt");
            BufferedWriter bw = new BufferedWriter(fw);
            PrintWriter pw = new PrintWriter(bw);

            // Write a string
            pw.println("Hello world");

            // Close file stream
            pw.close();

            // Open file for reading
            fr = new FileReader("test.txt");
            BufferedReader br = new BufferedReader(fr);
```

I/O Exceptions

```
            // Read a string
            Scanner scan = new Scanner(br);
            String str = scan.next();
            System.out.println("Read string: " + str);

            // Close file stream
            br.close();
        }
        catch(IOException e)
        {
            System.err.println("I/O EXCEPTION: " + e.getMessage());
        }
    }
}
```

I/O Exceptions

- Oftentimes, an object reference variable created within a `try` block needs to be accessed outside the `try` block
- In this scenario, simply declare and initialize it to `null` before the `try` block

```
SomeClass obj = null;

try
{
  obj = new SomeClass();
}
catch(…)
{
  …
}

obj.method1();
```