Lecture 4 — Java III

Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

November 14, 2015

Part I

Exceptions, Annotations, & I/O

Exceptions

`Exceptions` are used to signal an error (something unusual) during the execution of a program.

This disrupts the normal flow of the program's instructions.

An `Exception` is "thrown" when it is generated.

The keyword is `throw`.

Exceptions

Exceptions are generated automatically or explicitly in a program.

Write: `throw new RuntimeException(...)`

The JRE may also generate an exception:

Call `toString()` on an object that is null. JRE throws a `NullPointerException`.

Handling the Exception

If there is no code to “handle” this exception, then it goes up one level to the caller of that method until it gets to `main`.

If `main` cannot deal with it either, then execution of the program will be stopped.

An error message will be printed on the screen.

How to handle the exception? With an exception handler.

Catching Exceptions

Given that an `Exception` is `thrown`, it should come as no surprise that the keyword to handle it is `catch`.

We demarcate code that a `catch` block is supposed to apply to in a `try` block.

```
try {
  // Some statements
} catch (Exception e) {
  // Handle the Exception
}
```

Try-Catch Block

Let's look at an example:

```
public void handleException() {
  Object example = null;
  try {
    System.out.println(example.toString());
  } catch (NullPointerException npe) {
    // Handle the NPE
  }
}
```

Dealing with the Exception

Handling an exception depends on the specific program.

Sometimes you might just display an error message.

In some cases you don't even want to handle the error.

Maybe the output contains information to find the problem.

Multiple Catch Blocks

```
try {
   // Some statements
} catch (NullPointerException e) {
   // Handle the Exception
} catch (IOException e2) {
   // Do Something Else
} catch (FileNotFoundException | SQLException e3) {
   // Some other things
}
```

The Finally Block

Optionally, after the catch block: the `finally` block.

The finally block always executes when the try block exits, whether it went to the catch block or not.

Use: avoid having cleanup code accidentally bypassed by a `return`, `continue`, or `break`.

Finally Block Format

```
try {
   // Some statements
} catch (Exception e) {
   // Handle the Exception
} finally {
   // Cleanup
}
```

Finally Blocks

What is the return value of this function?

```
public boolean tryFinally() {
  try {
    return false;
  } finally {
    return true;
  }
}
```

Exceptions: Putting it Together

```
public void writeList() {
    PrintWriter out = null;

    try {
        System.out.println("Entering" + " try statement");

        out = new PrintWriter(new FileWriter("OutFile.txt"));
        for (int i = 0; i < SIZE; i++)
            out.println("Value at: " + i + " = " + vector.elementAt(i));

    } catch (ArrayIndexOutOfBoundsException e) {
        System.err.println("Caught ArrayIndexOutOfBoundsException: "
                           +  e.getMessage());

    } catch (IOException e) {
        System.err.println("Caught IOException: " +  e.getMessage());

    } finally {
        if (out != null) {
            System.out.println("Closing PrintWriter");
            out.close();
        }
        else {
            System.out.println("PrintWriter not open");
        }
    }
}
```

Exception Example

Scenario 1: Exception Occurs

The `FileWriter` throws an `IOException`, because the file OutFile.txt is missing or inaccessible.

Execution of the lines in the try block stops.

The JRE then starts looking for a handler.

It finds the catch block that specifies the IOException.

The catch block is executed.

Then the finally block executes.

Exception Example

Scenario 2: Everything Goes Well

Suppose no exception occurs.

Then the lines of the try block are all executed.

Then the finally block executes.

Exception Handling

Sometimes we don't want to handle an exception ourselves; sometimes we want to "pass the buck".

To do that, declare that your method `throws` an exception:

```
public void writeList() throws IOException.
```

Then handling the `IOException` is the responsibility of whatever method calls `writeList()`.

Creating Exceptions

Want to create an exception in your code?

Use the `throw` keyword.

```
throw new RunTimeException()
```

Throw whatever type of exception is appropriate.

Final Thoughts on Exceptions

As a final note, exceptions are supposed to be exceptional.

Don't use them as part of the expected flow of your program.

Use them to handle error conditions & unexpected situations.

Eclipse: Stack Traces

Eclipse demo time: stack traces.

What happens when an exception occurs, what a stack trace looks like, and how to find the error using one.

Annotations

We've already seen some annotations.

When we override a method from a superclass, we put the annotation `@Override` above it.

Annotations start with the @-symbol and can be attached to methods, classes, and variables.

Annotations

Meta-data; have no direct effect on the operation of the code.

They can be used for:

- Information for the Compiler
- Compile-Time Processing
- Runtime Processing

Basic Annotation

```
@Override
public void doSomething() { ... }
```

Multiple annotations can appear on an element, and annotations can have properties, such as:

```
@Author(name = ''Alice'')
@EBook
public void doSomething() { ... }
```

Annotations

You can write your own annotations, but the only one you are likely to need is the `@Override` annotation.

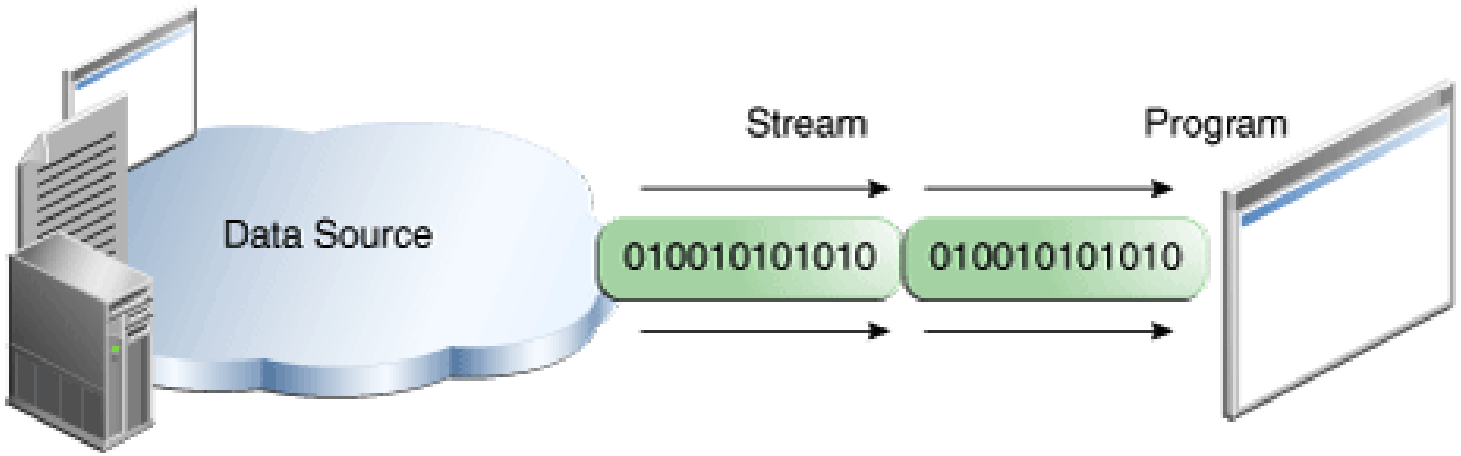
Java Input/Output

In Java, Input/Output (or I/O) is modelled as a *Stream*.

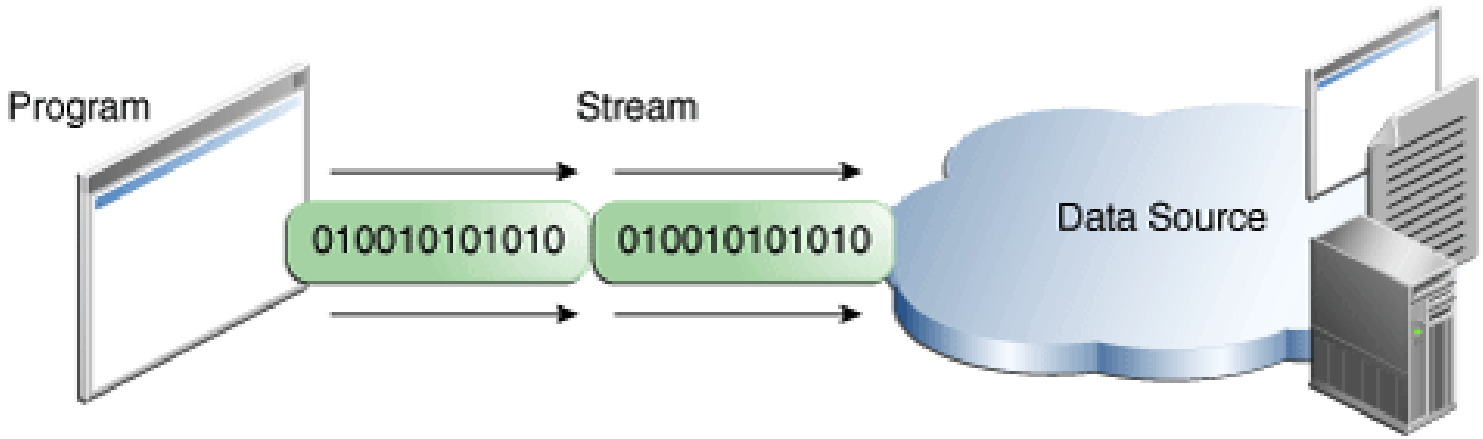
A Stream is a sequence of data.

A stream can be used to read data (input stream) or write data (output stream).

InputStream



OutputStream



Streams

The classes used to do I/O are descendants of the superclasses `InputStream` or `OutputStream`.

For example, to read from a file, use a `FileInputStream`.

Reading a File and Writing it Again

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

        FileInputStream in = null;
        FileOutputStream out = null;

        try {
            in = new FileInputStream("input.txt");
            out = new FileOutputStream("output.txt");
            int c;

            while ((c = in.read()) != -1) {
                out.write(c);
            }
        } finally {
            if (in != null) {
                in.close();
            }
            if (out != null) {
                out.close();
            }
        }
    }
}
```

Comments on File Copier

Note that the file streams are closed inside the `finally` block.

They will be closed even if something goes wrong.

This works, but it's really, really inefficient. Why?

We are reading from the disk one character (one byte) at a time and that's really slow.

What we'd often like to do is read a whole line at once.

Reading a Line at a Time

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

        BufferedReader inputStream = null;
        PrintWriter outputStream = null;

        try {
            inputStream = new BufferedReader(new FileReader("input.txt"));
            outputStream = new PrintWriter(new FileWriter("output.txt"));

            String l;
            while ((l = inputStream.readLine()) != null) {
                outputStream.println(l);
            }
        } finally {
            if (inputStream != null) {
                inputStream.close();
            }
            if (outputStream != null) {
                outputStream.close();
            }
        }
    }
}
```

Buffered Reader

We also use a `BufferedReader` to get buffered I/O.

When we read one byte at a time, each read or write request for an individual byte results in going to the disk or network.

When we work with buffered I/O, we read a buffer (some array of data) and store it.

Then when we try to read a byte we check if it's in the array we already have stored.

If so, se that value instead of going to the disk.

When we reach the end of the buffer, fill up the buffer again.

Buffered Writer

Buffered output streams mean that output is kept in a buffer and only written to disk when the buffer is full.

A crash might terminate execution before the buffer is written to disk and some of the expected output will not appear.

To force the buffer to output its data, you can call `flush()`.

Closing the stream also has the effect of flushing the buffer.

Other Kinds of Streams

Java also has data streams and object streams, which are used for reading/writing/storing/loading more complex things.

They are beyond the scope of this course.

Part II

On Problem Solving

The Key to Programming

The key to programming is simple.

1. Determine where we are (point A).
2. Determine where we're trying to go (point B).
3. Find a series of steps to get us from A to B.

Math Problem Example

Suppose you haven't memorized the answer to 12×13.

Other than whipping out the calculator, what can you do?

Break it down into a series of problems you know how to solve.

$(12x10) + (12x3)$

We now have three problems, but they're small and easy.

Decomposition

Note: we have to know the rules of math to understand how the problem can be decomposed.

In programming, the step you come up with might be a little bit too complicated for a computer or person to execute.

So you break the items of step 3 down by repeating this process.

Keep breaking it down until the steps are small enough for the computer to execute.

Decomposition: Step Size

How small those steps are is a matter of what language and tools you are using.

Imagine a Fourier Transform of a signal (fancy math).

If you're using an advanced language you might just write `FourierTransform(signal)`.

Or you might have to write the transform at a low level.

Key Lesson

Decomposing the problem is the key.

Learning the syntax of the language is another matter...