Intro to Java (Page 3)

JUMP June 2019

Draft: 07/08/2019

Table of Contents

Exception Handling

1. The Exception Chain and Hierarchy
2. And if you don’t handle them..
3. Using Multiple “catch” statement
4. Creating Subclass Exceptions and Exception Superclasses
5. Nested Try Blocks
6. Throwing an exception on purpose
7. The “Throwable” Object
8. “finally”
9. “throws”
10. Java’s Built-In Exceptions

Exception Handling – The Exception Chain and Hierarchy

1. In Java all exceptions are represented by classes
2. All are derived from the “Throwable” class
3. Two direct subclass of “Throwable” are:
    1. Exception: errors that occur in your app
        1. Example: divide-by-zero, array out-of-bound, file errors are these types of errors
        2. RuntimeException are errors that occur directly at runtime
    2. Error: errors that occur in the JVM itself, not your app

Exception Handling – The Exception Chain and Hierarchy

1. Exception handling is managed via 5 keywords
    1. try
    2. catch
    3. throw
    4. throws
    5. finally
2. They form a subsystem in which the use of one implies the use of another.

```
try {
 // block of code to monitor for errors
}
catch (ExcepType1 exOb) {
 // handler for ExcepType1
}
catch (ExcepType2 exOb) {
 // handler for ExcepType2
}
```

Exception Handling – The Exception Chain and Hierarchy

Example of code where an error is generated in a method, but caught in a try catch block.

See live code in STS “SimpleException2”

Exception Handling – And if you don’t handle them..

1. The reason for catching exceptions is to prevent abnormal program termination.
2. If your app doesn’t catch the exception, the JVM will.
3. When the JVM catches an error, it will terminate the app and display a stack trace and error message.

```
// Let JVM handle the error.
class NotHandled {
  public static void main(String args[]) {
    int nums[] = new int[4];
    System.out.println("Before exception is generated.");
    // generate an index out-of-bounds exception
    nums[7] = 10;
  }
}
```

Exception Handling – And if you don’t handle them..

In this example we are not using the correct exception handler.

```
// This won't work!
class ExcTypeMismatch {
  public static void main(String args[]) {
    int nums[] = new int[4];
    try {
      System.out.println("Before exception is generated.");
      //generate an index out-of-bounds exception
      nums[7] = 10;
      System.out.println("this won't be displayed");
    }
    /* Can't catch an array boundary error with an ArithmeticException. */
    catch (ArithmeticException exc) {
      // catch the exception
      System.out.println("Index out-of-bounds!");
    }
    System.out.println("After catch statement.");
  }
}
```

Exception Handling – Using Multiple “catch” statement

Exception handling can be set to deal with multiple types of errors by chaining catch blocks.

See live code in STS “SimpleException4”

See live code in STS “SimpleExceptionMultiCatch”

Exception Handling – Creating Subclass Exceptions and Exception Superclasses

- It is possible to catch all errors using “Throwable”
- It is also possible to create our own.

See live code in STS “SimpleException5”

See live code in STS “SimpleExceptionSubclass”

Exception Handling – Nested Try Blocks

It is possible to nest Try blocks

See live code in STS “SimpleException6”

Exception Handling – Throwing an exception on purpose

While the idea of throwing an exception on purpose may seem crazy, it is actually beneficial to be able to do so, if only for testing purposes to ensure your code is running correctly.

See live code in STS “SimpleException7”

We can also “rethrow” an Exception, the reason is that this way allows multiple handlers to access the exception. At some point, you will want to create your own custom exceptions as part of your overall code management and design strategy and it will be important to handle errors for multiple handlers under a single exception.

See live code in STS “SimpleException8”

Exception Handling – The “Throwable” Object

- All exceptions support the methods defined by “Throwable”
- **printStackTrace()** : Display a standard error message plus a record of the method call(s) that lead up to the exception.
- **toString()**: to retrieve the standard error message.

Method	Description
Throwable fillInStackTrace( )	Returns a **Throwable** object that contains a completed stack trace. This object can be rethrown.
String getLocalizedMessage( )	Returns a localized description of the exception.
String getMessage( )	Returns a description of the exception.
void printStackTrace( )	Displays the stack trace.
void printStackTrace(PrintStream *stream*)	Sends the stack trace to the specified stream.
void printStackTrace(PrintWriter *stream*)	Sends the stack trace to the specified stream.
String toString( )	Returns a **String** object containing a complete description of the exception. This method is called by **println( )** when outputting a **Throwable** object.

See live code in STS “SimpleException9”

Exception Handling – “finally”

- Code which must execute at all time after the handling of the exception.
- Basically, after the try/catch has executed, “finally” will execute as well.

See live code in STS “SimpleExceptionFinally”

Exception Handling – “throws”

- In some cases, if a method generates an exception that it does not handle, it must declare that exception in a **throws** clause. Here is the general form of a method that includes a **throws** clause:

```
ret-type methName(param-list) throws except-list {
   // body
}
```

- The exceptions can be a listed separated by commas.

See live code in STS “SimpleIOException”

Exception Handling – Java’s Built-In Exceptions

- Inside the java.lang standard package, Java defines several exception classes.
- The most general of these exceptions are subclasses of the standard type “RuntimeException”.
- “java.lang” is implicitly imported in all Java programs
- Most exceptions from “RuntineException” are automatically available and need not be included in any method’s “throws” list.
- These are called “unchecked exceptions” because the compiler does not check to see if a method handles or throws these exceptions.
- For more information on exceptions, please follow the link: https://www.geeksforgeeks.org/types-of-exception-in-java-with-examples/

Input/Output (I/O)

1. Java I/O is all about Streams
2. Byte Streams and Character Streams
3. The Byte Stream Classes
4. The Character Stream Classes
5. The Predefined Streams
6. Using the Byte Streams
7. Reading and Writing Files using Byte Streams
8. Automatically Closing a File
9. Reading and Writing Binary Data
10. Random-Access Files
11. Java’s Character-Based Streams
12. File I/O Using Character Streams
13. Using Java’s Type Wrappers to Convert Numeric Strings

Input/Output (I/O) - Java I/O is all about Streams

- Java programs perform input/output (I/O) through streams
- An I/O Stream is an abstraction that either produces or consumes information.
- A stream is linked to a physical I/O device by Java
- All streams behave in the same manner even if the physical devices differ
- Same I/O classes and methods can be applied to different types of devices

Input/Output (I/O) - Byte Streams and Character Streams

- There are 2 types of I/O streams
  - Byte
  - Character
- Byte streams are great for handling input and output of bytes, for example the reading and writing of binary data
- Character streams use Unicode and can be internationalized.
- At the lowest of levels, all I/O streams are in fact at the byte level, but we have many classes which abstract the harsh coding necessary to implement such features

Input/Output (I/O) - The Byte Stream Classes

1. Byte Streams are defined using two class hierarchies
    1. InputStream
    2. OutputStream
2. This is all about reading and writing to various devices including hard disk drives

http://ecomputernotes.com/java/stream/byte-stream-classes

Input/Output (I/O) - The Character Stream Classes

1. Character streams are defined by using two class hierarchies
    1. Reader
    2. Writer

http://ecomputernotes.com/java/stream/java-character-stream-classes

Input/Output (I/O) - The Predefined Streams

1. With the “java.lang” package comes a class defined as “System”.
2. “System” encapsulates several aspect of run-time environment
3. It contains 3 predefined stream variables
   1. “in”
   2. “out”
   3. “err”
4. These fields are declared as public final and static within System. This means they can be used by any other part of your program without reference to a specific System object (no instantiation)

- “System.out”: your standard output stream, by default the console.
- “System.in”: your standard input stream, by default the keyboard.
- “System.err”: your standard error stream which is also by default the console.

Note: All streams can be redirected to any compatible I/O device.

Input/Output (I/O) - Using the Byte Streams

- InputStream and OutputStream can throw an IOException error
- Can be used to read console input, although it is preferable to use a character stream.

https://docs.oracle.com/javase/7/docs/api/java/io/InputStream.html

https://docs.oracle.com/javase/7/docs/api/java/io/OutputStream.html

See live code in STS "IOReadBytes"

Input/Output (I/O) - Reading and Writing Files using Byte Streams

Use FileInputStream and FileOutputStream for I/O disk based operations.

See live code in STS “IOReadTextFile”

See live code in STS “IOWriteFile”

Input/Output (I/O) - Automatically Closing a File

1. Using a Try with Resources technique is basically to ensure that the resource will be closed regardless of any issues arising
2. This prevents memory leaks.

See live code in STS “IOTryWithResources”

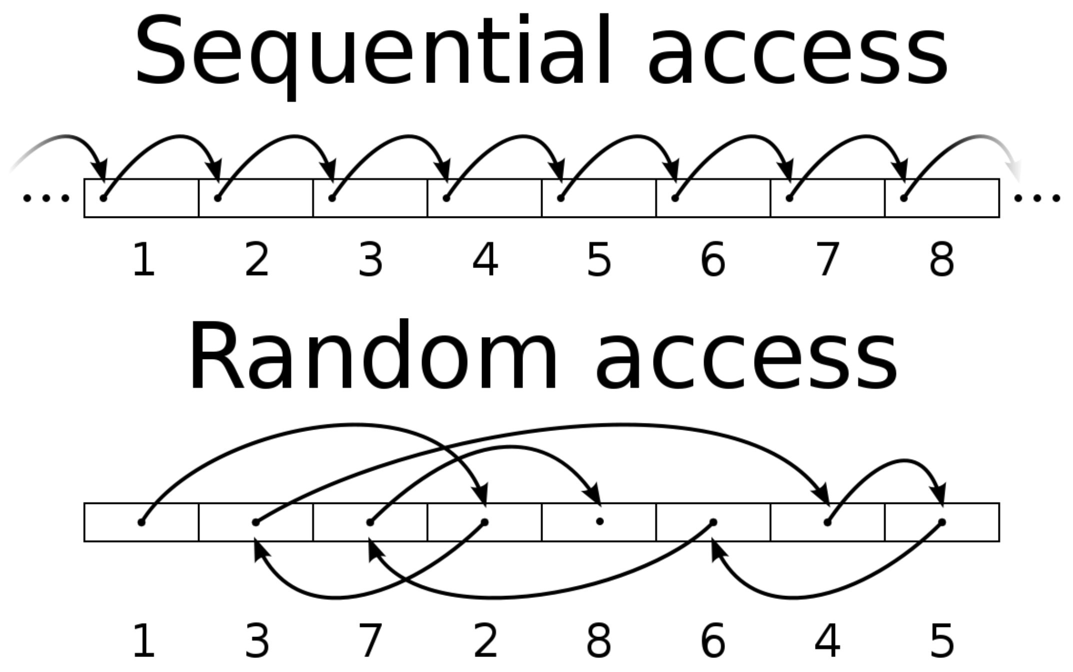
Input/Output (I/O) - Reading and Writing Binary Data

- DataInputStream
  https://docs.oracle.com/javase/7/docs/api/java/io/DataInputStream.html
- DataOutputStream
  https://docs.oracle.com/javase/7/docs/api/java/io/DataOutputStream.html

See live code in STS “IODataOutputStream”

Input/Output (I/O) - Random-Access Files

- All the files we've read so far have been read 'sequentially'.
- Random-Access Files read and write is about reading or writing data by setting a file pointer. With Random Access capability, you are using file positioning to determine the location of your operation within the file.



See live code in STS "IORandomAccess"

Input/Output (I/O) - Java’s Character-Based Streams

- Reader https://docs.oracle.com/javase/7/docs/api/java/io/Reader.html
- Writer https://docs.oracle.com/javase/7/docs/api/java/io/Writer.html

- Note: the best class for reading from the console is the BufferedReader…

See live code in STS “IOBufferedReader”

Input/Output (I/O) - File I/O Using Character Streams

1. FileReader
2. FileWriter

See live code in STS “IOFileReader”

See live code in STS “IOFileWriter”

See live code in STS “IOReadLines”

Input/Output (I/O) - Using Java's Type Wrappers to Convert Numeric Strings

- Easy and convenient way to read numerical strings
- **Support for primitives: Float**, **Long**, **Integer**, **Short**, **Byte**, **Character**, and **Boolean**.

See live code in STS "IOTypeWrappers"