Exceptions & File I/O

Lecture 12a

Topics

- Introduction to File Input and Output
- The PrintWriter Class
- The File Class
- The Scanner Class
- Handling Exceptions
- Throwing Exceptions

File Input and Output (1 of 3)

- The programs you have written so far require you to reenter data each time the program runs.

- Point to ponder #1:

  Why this happens?

Because RAM is volatile. The data stored in variables and objects in RAM disappears once the program stops running since volatile memory requires power to maintain the stored information.

File Input and Output (2 of 3)

- To retain data between the times it runs, a program must have a way of saving the data.
- The data can be saved to a file. Then, it will remain there after the program stops running.

- Point to ponder #2:

  Where do we usually store files?

  Computer's disk.

File Input and Output (3 of 3)

- Steps followed by a Java program to use a file:
  - The file has to be opened.
  - Data is then written to the file or read from the file.
  - The file must be closed prior to program termination.
- Files can be *input files* or *output files*.
  - Input files: a program reads input from
  - Output files: a program writes data to
- In general, there are two types of files:
  - Text (Data encoded as text. Readable in a text editor)
  - Binary (Data not encoded as text. Not readable in a text editor.)

Writing Text To a File (1 of 2)

- To write data to a file you create an instance of the `PrintWriter` class. This class allows you to open a file for writing.

```
PrintWriter outputFile = new PrintWriter("StudentData.txt");
```

Pass the name of the file that you wish to open as a string argument to the `PrintWriter` constructor.

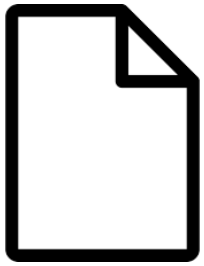
Warning: if the file already exists, it will be erased and replaced with a new file.

Writing Text To a File (2 of 2)

- Point to ponder #3:

  At this point, what should be the content of `StudentData.txt` ?

  Empty.



- Point to ponder #4:

  Where the file `StudentData.txt` will be created?

  Current directory (outside the src folder).

The `PrintWriter` Class (1 of 6)

- The `PrintWriter` class allows you to write data to a file using the `print` and `println` methods, as you have been using to display data on the screen.
- Just as with the `System.out` object, the `println` method of the `PrintWriter` class will place a newline character after the written data.
- The `print` method writes data without writing the newline character.

The `PrintWriter` Class (2 of 6)

Open the file.

```
PrintWriter outputFile = new PrintWriter("Names.txt");
outputFile.println("Chris");
outputFile.println("Kathryn");
outputFile.println("Jean");
outputFile.close();
```

Close the file.

Write data to the file.

Names.txt

Chris
Kathryn
Jean

The `PrintWriter` Class (3 of 6)

Open the file.

```
PrintWriter outputFile = new PrintWriter("Names.txt");
outputFile.print("Chris ");
outputFile.print("Kathryn ");
outputFile.print("Jean");
outputFile.close();
```

Close the file.

Write data to the file.

Names.txt

Chris Kathryn Jean

The `PrintWriter` Class (4 of 6)

- Point to ponder #5:

  Why is it required to close the files after we finish writing data to them?

  Because Java uses buffers in memory to temporarily store data that is being added. Only after we close the file, the information is written to it.

- Point to ponder #6:

  Why is a buffer desired for this writing operation?

  Because writing data to memory is faster than writing it to a disk.

The `PrintWriter` Class (5 of 6)

- To use the `PrintWriter` class, put the following `import` statement at the top of the source file:

```
import java.io.*;
```

The `PrintWriter` Class (6 of 6)

Example. `FileWriteDemo.java`

```
1 import java.util.Scanner; // Needed for Scanner class
2 import java.io.*; // Needed for File I/O classes
3
4 public class FileWriteDemo
5 {
6   public static void main(String[] args) throws IOException
7   {
8      String filename; // File name
9      String friendName; // Friend's name
10     int numFriends; // Number of friends
11
12     // Create a Scanner object for keyboard input.
13     Scanner keyboard = new Scanner(System.in);
14
15     // Get the number of friends.
16    System.out.print("How many friends do you have? ");
17    numFriends = keyboard.nextInt();
18
19     // Consume the remaining newline character.
20     keyboard.nextLine();
21     // Get the filename.
22     System.out.print("Enter the filename: ");
23     filename = keyboard.nextLine();
24     // Open the file.
25    PrintWriter outputFile = new PrintWriter(filename);
26
27     // Get data and write it to the file.
28     for (int i = 1; i <= numFriends; i++)
29     {
30        // Get the name of a friend.
31        System.out.print("Enter the name of friend " +
32                         "number " + i + ": ");
33        friendName = keyboard.nextLine();
34        // Write the name to the file.
35        outputFile.println(friendName);
36      }
37
38      outputFile.close(); // Close the file.
39      System.out.println("Data written to the file.");
40   }
41 }
```

File I/O Exceptions (1 of 3)

- When something unexpected happens in a Java program, an *exception* is thrown.
- Examples:
  - you want to create a file, but the disc is full
  - you want to overwrite a read-only file
  - you specify a file path that does not exist

- The program cannot continue until those situations have been dealt with. The program halts and an error message is displayed (Oh no, a bug in the code!).

File I/O Exceptions (2 of 3)

- The method that is executing when the exception is thrown must either handle the exception or throw it again.
- Handling the exception will be discussed soon.
- Because `PrintWriter` objects are capable of throwing exceptions, we need to write code that deals with possible exceptions or rethrow them
- To throw the exception again (rethrow), the method needs a `throws` clause in the method header with the appropriate exception type.

File I/O Exceptions (3 of 3)

- To insert a `throws` clause in a method header, simply add the word *throws* and the name of the expected exception.
- `PrintWriter` objects can throw an `IOException`, so we write the `throws` clause as so:

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

Appending Text to a File (1 of 3)

- Sometimes, you want to preserve an existing file and only append new data to its current contents (not recreate the file from scratch).
- Appending to a file means writing new data to the end of the data that already exists in the file.

Appending Text to a File (2 of 3)

- To avoid erasing a file that already exists, create a `FileWriter` object in this manner:

```
FileWriter fw = new FileWriter("names.txt", true);
```

Name of the file.

boolean indicating whether or not to append the data written.

- Then, create a `PrintWriter` object in this manner:

```
PrintWriter pw = new PrintWriter(fw);
```

Appending Text to a File (3 of 3)

- Example:

Names.txt

Chris
Kathryn
Jean

```
FileWriter fwriter = new FileWriter("Names.txt", true);
PrintWriter outputFile = new PrintWriter(fwriter);
outputFile.println("Bill");
outputFile.println("Steven");
outputFile.close();
```

Names.txt

Chris
Kathryn
Jean
Bill
Steven

Specifying a File Location (1 of 3)

- You an specify the location where the file will be created
- On a Windows computer, file paths contain backslash (\) characters.
- Remember, if the backslash is used in a string literal, it is the escape character. Thus, so you must use two of them:

```
PrintWriter outFile = new PrintWriter("A:\\PriceList.txt");
```

or

```
PrintWriter outFile = new PrintWriter("A:/PriceList.txt");
```

- Point to ponder #7:

  What is the output here?

```
System.out.println("\This is a test");
```

Compile-time error. The command \T does not exist.

Specifying a File Location (2 of 3)

- This is only necessary if the backslash is in a string literal.
- If the backslash is in a `String` object, then it will be handled properly.

For instance:

```
System.out.println("Enter the path");
String path = input.nextLine(); "C:\CPP\Friends.txt"
FileWriter fwriter = new FileWriter(path, true);
```

Specifying a File Location (3 of 3)

- To specify the location of the folder “files” (same level of the “src” folder) under your project on Eclipse use:

```
PrintWriter outFile = new PrintWriter("files/Friends.txt");
```

Reading Data From a File (1 of 4)

- You use the `File` class and the `Scanner` class to read data from a file:

Pass the name of the file as an argument to the `File` class constructor. It is used to represent a file

```
File myFile = new File("Customers.txt");
Scanner inputFile = new Scanner(myFile);
```

Pass the `File` object as an argument to the `Scanner` class constructor.

Reading Data From a File (2 of 4)

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

System.out.print("Enter the filename: ");
String filename = keyboard.nextLine();

File file = new File(filename);
Scanner inputFile = new Scanner(file);
```

- The lines above:
  - Creates an instance of the `Scanner` class to read from the keyboard
  - Prompt the user for a filename
  - Get the filename from the user
  - Create an instance of the `File` class to represent the file
  - Create an instance of the `Scanner` class that reads from the file

Reading Data From a File (3 of 4)

- Once an instance of `Scanner` is created, data can be read using the same methods that you have used to read keyboard input (`nextLine`, `nextInt`, `nextDouble`, etc).

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class ReadingFiles {

    public static void main(String[] args) throws IOException { //throws clause required here

        // Open the file.
        File file = new File("files/Friends.txt");
        Scanner inputFile = new Scanner(file);

        // Read a line from the file and advance to the next line.
        String str = inputFile.nextLine();
        System.out.println("The first line in the file is: " + str);

        // Read another line from the file and advance to the next line.
        str = inputFile.nextLine();
        System.out.println("The second line in the file is: " + str);

        // Close the file.
        inputFile.close();
    }
}
```

Reading Data From a File (4 of 4)

- Point to ponder #8:

  What if you try to read a file that does no exist?

  I/O Exception!!!

File I/O Exceptions

- The `Scanner` class can throw an `IOException` when a `File` object is passed to its constructor.
- So, we put a `throws IOException` clause in the header of the method that instantiates the `Scanner` class.

Detecting The End of a File (1 of 3)

- We usually need to read the contents of a file without knowing the number of items that are stored there.
- The Scanner class has a method named `hasNext` that can be used to determine whether the file has more data that can be read.

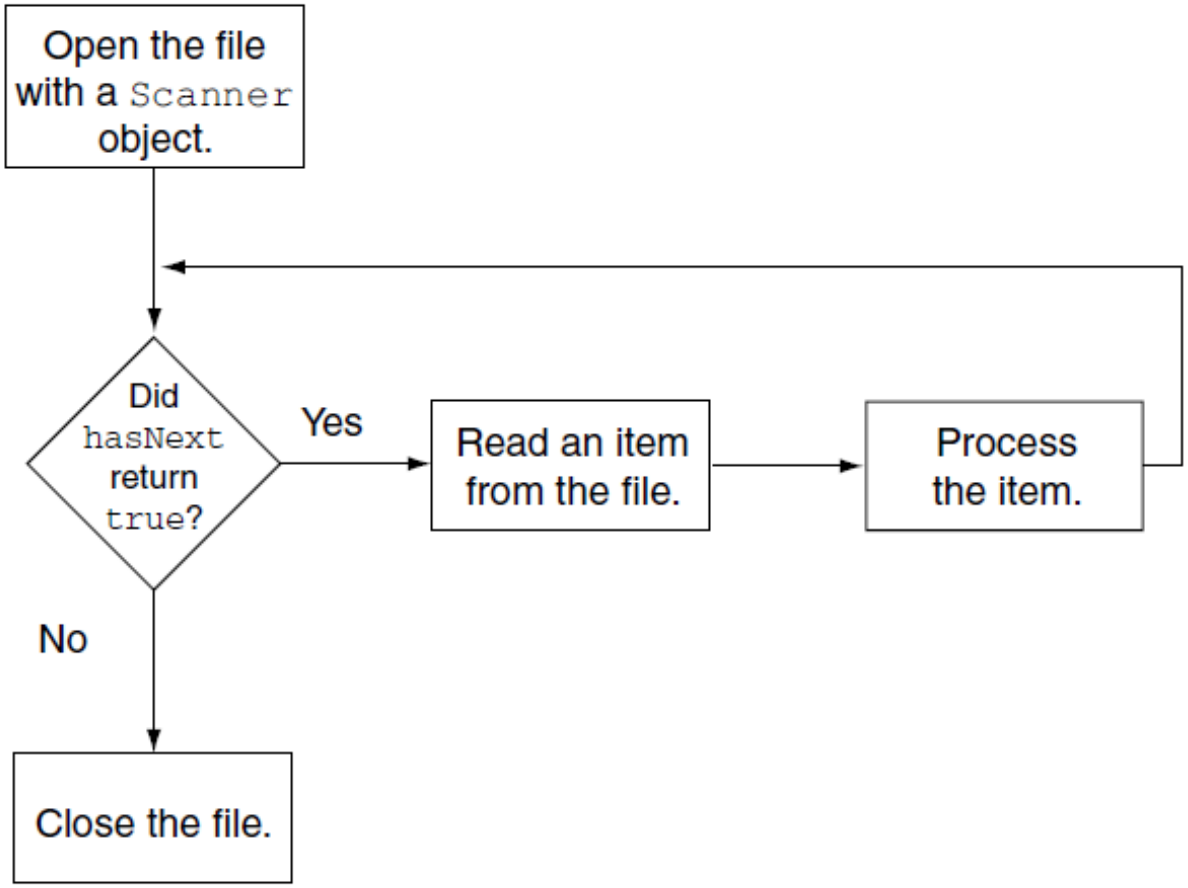
Detecting The End of a File (2 of 3)

- Scanner class's `hasNext()` will return true if another item can be read from the file.

```
   // Open the file.
   File file = new File(filename);
   Scanner inputFile = new Scanner(file);

   // Read until the end of the file.
   while (inputFile.hasNext())
   {
      String str = inputFile.nextLine();
      System.out.println(str);
   }
inputFile.close();// close the file when done.
```

Detecting The End of a File (3 of 3)



Reading Primitive Values from a File

- The Scanner class provides methods for reading primitive values
- Those methods can also be used to read primitive values from a file

Ex: FileSum.java

Checking for a File's Existence

- It's usually a good idea to make sure that a file exists before you try to open it for input
- If you attempt to open a file that does not exist, the program will throw an exception and halt
- File class's exists() method will return true if the file exists.

```
File file = new File("files/Frends.txt"); // Make sure the file exists
if (file.exists())
{
   Scanner inputFile = new Scanner(file); // Open the file for reading.
   ...
} else {
   System.out.println("The file Friends.txt is not found.");
}
```

Ex: FileSum2.java
FileWriteDemo2.java

Exceptions & File I/O

Lecture 12b

Topics

- Introduction to File Input and Output
- The PrintWriter Class
- The File Class
- The Scanner Class
- Handling Exceptions
- Throwing Exceptions

Handling Exceptions (1 of 3)

- An exception is an object that is generated as the result of an error or an unexpected event.
- Exception are said to have been “thrown.”
- It is the programmer responsibility to write code that detects and handles exceptions.
- Unhandled exceptions will crash a program (the application will halt).
- Java allows you to create exception handlers.
- Example: BadArray.java

Handling Exceptions (2 of 3)

```
1 /**
2 This program causes an error and crashes.
3 */
4
5 public class BadArray {
6   public static void main(String[] args) {
7     // Create an array with 3 elements.
8     int[] numbers = { 1, 2, 3 };
9
10    // Attempt to read beyond the bounds of the array.
11    for (int i = 0; i <= 3; i++)
12        System.out.println(numbers[i]);
13  }
14}
```

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 3 out of bounds for length 3 at BadArray.main(BadArray.java:12)

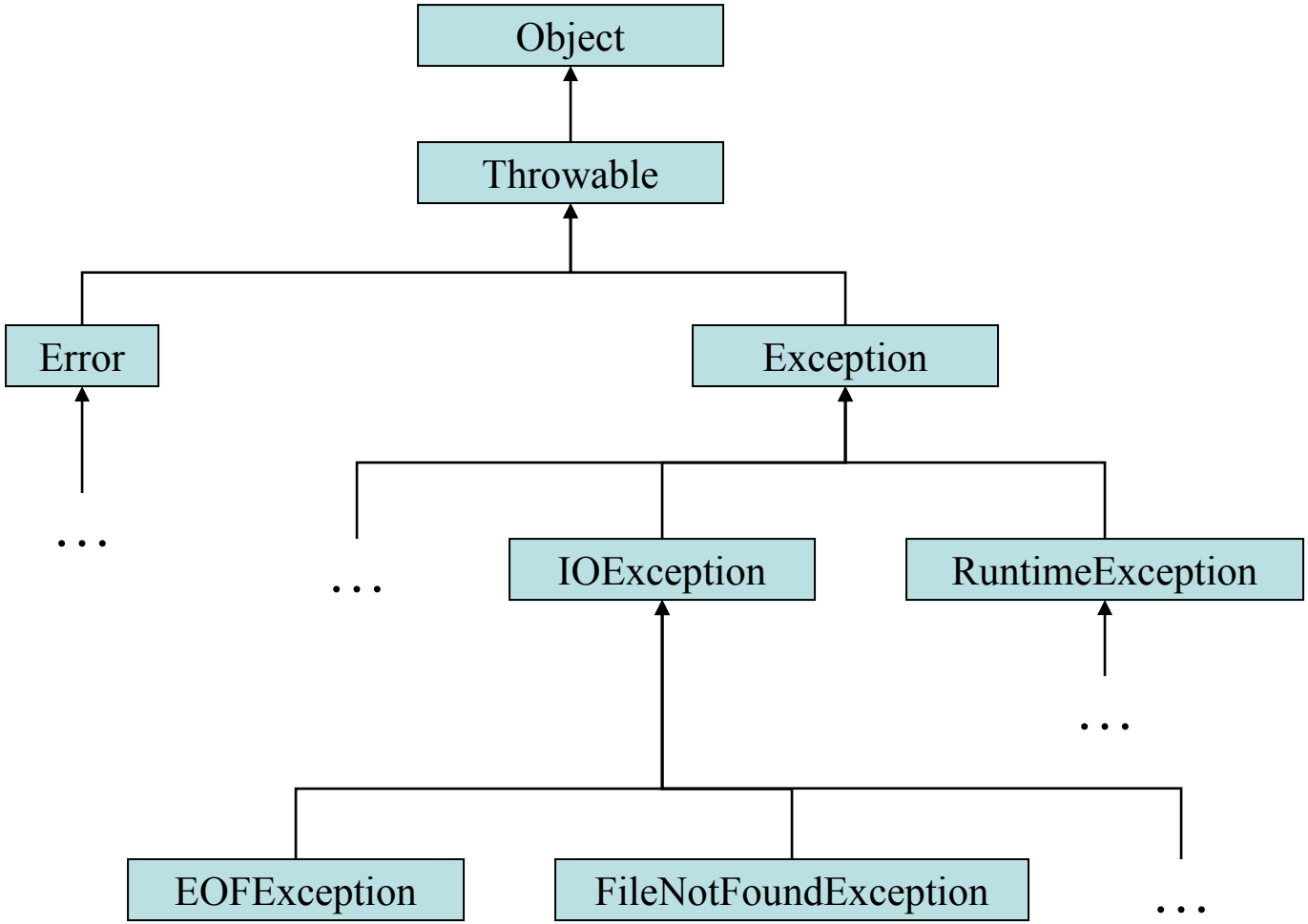
Handling Exceptions (3 of 3)

- An *exception handler* is a section of code that gracefully responds to exceptions.
- The process of intercepting and responding to exceptions is called *exception handling*.
- If your code does not handle an exception when it is thrown the *default exception* will do it.
- The default exception handler prints an error message and crashes the program.

Exception Classes (1 of 3)

- An exception is an object.
- Exception objects are created from classes in the Java API hierarchy of exception classes.
- All the exception classes in the hierarchy are derived from the `Throwable` class.
- `Error` and `Exception` are derived from the `Throwable` class.

Exception Classes (2 of 3)



Exception Classes (3 of 3)

- Classes that are derived from `Error`:
  - are for exceptions that are thrown when critical errors occur. (i.e.)
    - an internal error in the Java Virtual Machine, or
    - running out of memory.
- Applications should not try to handle these errors because they are the result of a serious condition.
- Programmers should handle the exceptions that are instances of classes that are derived from the `Exception` class.

Handling Exceptions (1 of 6)

- To handle an exception, you use a *try* statement.

```
try
{
   (try block statements...)
}
catch (ExceptionType ParameterName)
{
   (catch block statements...)
}
```

- First the keyword `try` indicates a block of code will be attempted (the curly braces are required).
- This block of code is known as a *try block*.

Handling Exceptions (2 of 6)

- A *try block* is:
  - one or more statements that are executed, and
  - can potentially throw an exception.
- The application will not halt if the try block throws an exception.
- After the try block, a `catch` clause appears.

Handling Exceptions (3 of 6)

- A catch clause begins with the key word `catch`:

  `catch (`*`ExceptionType ParameterName`*`)`

  - *`ExceptionType`* is the name of an exception class and
  - *`ParameterName`* is a variable name which will reference the exception object if the code in the try block throws an exception.

- The code that immediately follows the catch clause is known as a *catch block* (the curly braces are required).
- The code in the catch block is executed if the try block throws an exception of the *`ExceptionType`* class.

Handling Exceptions (4 of 6)

- This code is designed to handle a `FileNotFoundException` if it is thrown.

```
try
  {
    File file = new File ("MyFile.txt");
    Scanner inputFile = new Scanner(file);
    System.out.println("File was found.");
  }
  catch (FileNotFoundException e)
  {
    System.out.println("File not found.");
  }
  System.out.println("Done.");
```

- The Java Virtual Machine searches for a `catch` clause that can deal with the exception.

Handling Exceptions (5 of 6)

- The parameter must be of a type that is compatible with the thrown exception's type.
- After an exception, the program will continue execution at the point just past the catch block.

- Example: OpenFile.java

Handling Exceptions (6 of 6)

- Each exception object has a method named `getMessage` that can be used to retrieve the default error message for the exception.
- This is the same message that is displayed when the exception is not handled and the application halts.
- Example:
  - ExceptionMessage.java
  - ParseIntError.java

Polymorphic References To Exceptions (1 of 3)

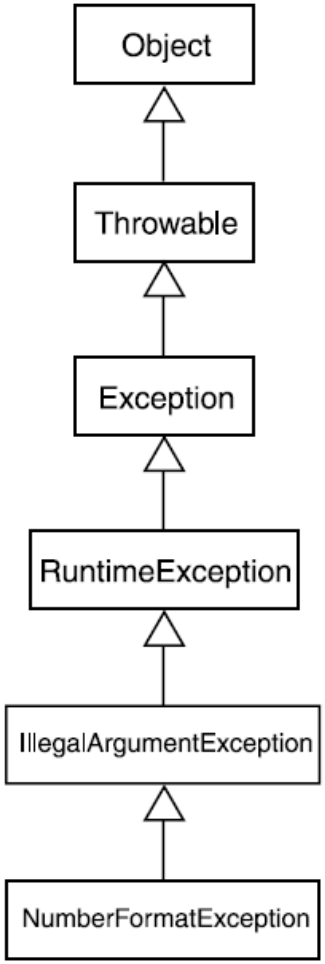
- When handling exceptions, you can use a polymorphic reference as a parameter in the `catch` clause.
- Most exceptions are derived from the `Exception` class.
- A `catch` clause that uses a parameter variable of the `Exception` type is capable of catching any exception that is derived from the `Exception` class.

Polymorphic References To Exceptions (2 of 3)

```
try
{
   number = Integer.parseInt(str);
}
catch (Exception e)
{
   System.out.println("The following error occurred: "
                      + e.getMessage());
}
```

- The `Integer` class's `parseInt` method throws a `NumberFormatException` object.
- The `NumberFormatException` class is derived from the `Exception` class.

Polymorphic References To Exceptions (3 of 3)



Handling Multiple Exceptions (1 of 4)

- The code in the try block may be capable of throwing more than one type of exception.
- A `catch` clause needs to be written for each type of exception that could potentially be thrown.
- The JVM will run the first compatible `catch` clause found (top-down).
- The `catch` clauses must be listed from most specific to most general.
- Example: SalesReport.java

Handling Multiple Exceptions (2 of 4)

- There can be many polymorphic catch clauses.
- A try statement may have only one `catch` clause for each specific type of exception.

```
try
{
  number = Integer.parseInt(str);
}
catch (NumberFormatException e)
{
  System.out.println("Bad number format.");
}
catch (NumberFormatException e) // ERROR!!!
{
  System.out.println(str + " is not a number.");
}
```

Handling Multiple Exceptions (3 of 4)

- The `NumberFormatException` class is derived from the `IllegalArgumentException` class.

```
try
{
  number = Integer.parseInt(str);
}
catch (IllegalArgumentException e)
{
  System.out.println("Bad number format.");
}
catch (NumberFormatException e) // ERROR!!!
{
  System.out.println(str + " is not a number.");
}
```

Handling Multiple Exceptions (4 of 4)

- The previous code could be rewritten to work, as follows, with no errors:

```
try
{
  number = Integer.parseInt(str);
}
catch (NumberFormatException e)
{
  System.out.println(str + " is not a number.");
}
catch (IllegalArgumentException e) //OK
{
  System.out.println("Bad number format.");
}
```

Exception Handlers to Recover from Errors (1 of 2)

- We saw how a try statement can have several catch clauses in order to handle different types of exceptions.
- However, those programs do not use the exception handlers to recover from any of the errors.
- Regardless of whether the file is not found, or a non-numeric item is encountered in the file, this program still halts.
- We can reach a better use of exception handling recovering from the exceptions.

SalesReport2.java (skipping an invalid input)

Exception Handlers to Recover from Errors (2 of 2)

```
double[] number = new double[5];
boolean exception;
Scanner input = new Scanner(System.in);
for (int i = 0; i < 5; i ++) {
    System.out.println("Enter the " + i + " grade:");
    do {
            try
            {
                number[i] = Double.parseDouble(input.nextLine());
                exception = false;
            }
             catch (NumberFormatException e)
             {
                exception = true;
                System.out.println("Invalid input. Enter a double for the grade.");
             }
        } while (exception == true); //reading an input until the user enters a double
}
```

The `finally` Clause (1 of 3)

- The try statement may have an optional `finally` clause.
- If present, the `finally` clause must appear after all the `catch` clauses.

```
try
{
   (try block statements...)
}
catch (ExceptionType ParameterName)
{
   (catch block statements...)
}
finally
{
   (finally block statements...)
}
```

The `finally` Clause (2 of 3)

- The *finally block* is one or more statements,
  - that are always executed after the try block has executed and
  - after any catch blocks have executed if an exception was thrown.
- The statements in the finally block execute whether an exception occurs or not.

The `finally` Clause (3 of 3)

```
// Open the file.
File file = new File(filename);
Scanner inputFile = new Scanner(file);
try
{
  // Read and display the file's contents.
  while (inputFile.hasNext())
  {
    System.out.println(inputFile.nextDouble());
  }
}
catch (InputMismatchException e)
{
  System.out.println("Invalid data found.");
}
finally
{
  //Close the file.
  inputFile.close();
}
```

```
// Open the file.
File file = new File(filename);
Scanner inputFile = new Scanner(file);
try
{
  // Read and display the file's contents.
  while (inputFile.hasNext())
  {
    System.out.println(inputFile.nextDouble());
  }
}
catch (InputMismatchException e)
{
  System.out.println("Invalid data found.");
}
//Close the file.
inputFile.close();
```

- Point to ponder #2: Is it the same thing?

No. Finally will always run. If another error is thrown in either the try or the catch, the finally will still execute. You won't get that functionality by not using the finally statement.

The Stack Trace

- The *call stack* is an internal list of all the methods that are currently executing.
- A *stack trace* is a list of all the methods in the call stack.
- It indicates:
  - the method that was executing when an exception occurred and
  - all the methods that were called in order to execute that method (chain of methods).
- Example: StackTrace.java

The Stack Trace

```
public static void main(String[] args){
        myMethod();
}
public static void myMethod() {
        produceError();
}
public static void produceError() {
       System.out.println("abc".charAt(3));
}
```

Program Output

```
Calling myMethod...
Calling produceError...
Exception in thread "main" java.lang.StringIndexOutOfBoundsException:
      String index out of range: 3
        at java.lang.String.charAt(Unknown Source)
        at StackTrace.produceError(StackTrace.java:35)
        at StackTrace.myMethod(StackTrace.java:22)
        at StackTrace.main(StackTrace.java:11)
```

Uncaught Exceptions (1 of 2)

- When an exception is thrown, it cannot be ignored.
- It must be handled by the program, or by the default exception handler.
- When the code in a method throws an exception:
  - normal execution of that method stops, and
  - the JVM searches for a compatible exception handler inside the method.

Uncaught Exceptions (2 of 2)

- If there is no exception handler inside the method:
  - control of the program is passed to the previous method in the call stack.
  - If that method has no exception handler, then control is passed again, up the call stack, to the previous method.
- If control reaches the `main` method:
  - the main method must either handle the exception, or
  - the program is halted, and the default exception handler handles the exception.

Checked and Unchecked Exceptions (1 of 5)

- There are two categories of exceptions:
  - Unchecked (runtime)
  - Checked (compile time).
- *Unchecked exceptions* are those that are derived from the `Error` class or the `RuntimeException` class.
- Exceptions derived from `Error` are thrown when a critical error occurs and should not be handled.
- `RuntimeException` serves as a superclass for exceptions that result from programming errors.

Checked and Unchecked Exceptions (2 of 5)

- These exceptions can be avoided with properly written code.
- Unchecked exceptions, in most cases, should not be handled.
- All exceptions that are *not* derived from `Error` or `RuntimeException` are *checked exceptions*.

Checked and Unchecked Exceptions (3 of 5)

- If the code in a method can throw a checked exception, the method:
  - must handle the exception, or
  - it must have a `throws` clause listed in the method header.
- The `throws` clause informs the compiler what exceptions can be thrown from a method.

Checked and Unchecked Exceptions (4 of 5)

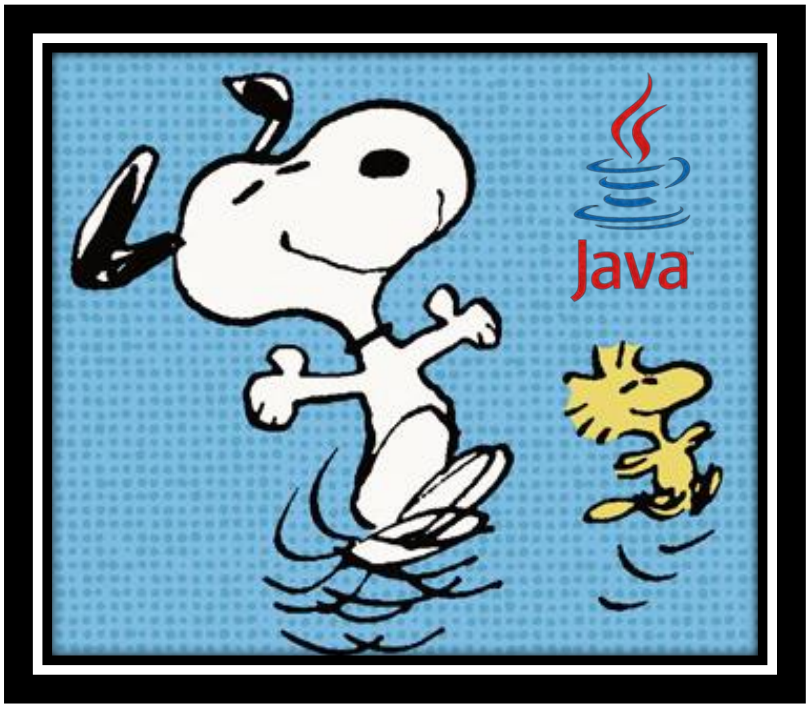
```
// This method will not compile!
public void displayFile(String name)
{
   // Open the file.
   File file = new File(name);
   Scanner inputFile = new Scanner(file);
   // Read and display the file's contents.
   while (inputFile.hasNext())
   {
     System.out.println(inputFile.nextLine());
   }
   // Close the file.
   inputFile.close();
}
```

Checked and Unchecked Exceptions (5 of 5)

- The code in this method is capable of throwing checked exceptions.
- The keyword `throws` can be written at the end of the method header, followed by a list of the types of exceptions that the method can throw.
- If you do not handle a checked exception that might occur, you must inform the compiler that your method might pass them up the call stack.

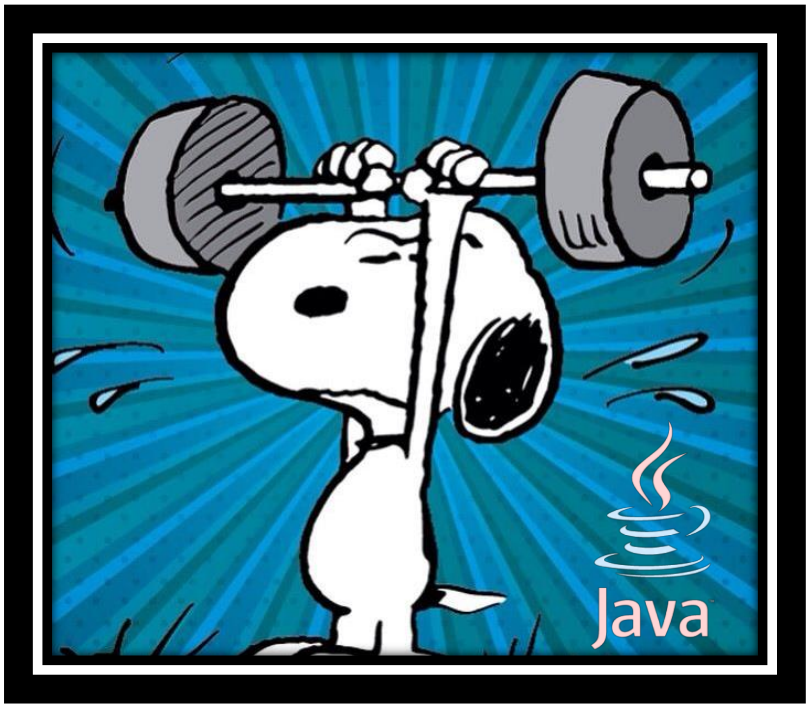
```
public void displayFile(String name)
    throws FileNotFoundException
```

The end …



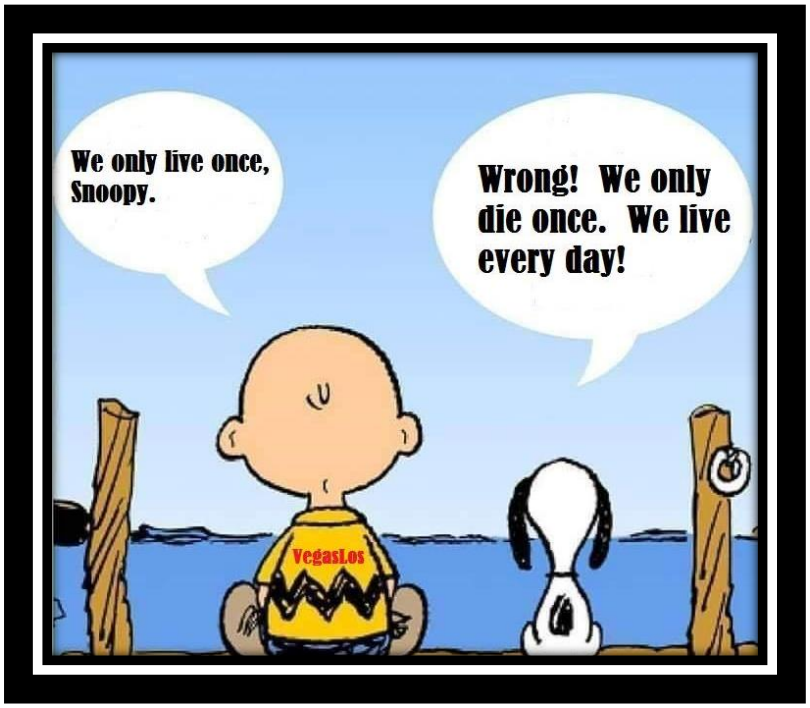

```
System.out.println("
 + "We made it!");
```

The end …




```
System.out.println("
 + "Keep up the great work!");
```

The end …




```
System.out.println("
  + "Enjoy your break! I'll be
around to help with anything.");
```