1. Exception Handling :: Introduction

Look at the *main*() method of the *ExamAvg* class and you will note this code:

```
public class ExamAvg {
  public static void main(String[] args) throws FileNotFoundException {
```

Why is that there (we will get to the real answer shortly)?

In the real world, users expect programs to not crash on them. Programs crash often times because of bugs but there can be other reasons as well. For example, suppose we are running the *ExamAvg* program and the contents of *scores-in.txt* is:

```
Fred
Wilma
Pebbles
```

This file would cause a bit of a problem because the program assumes that each line of the input file contains three integers and this file does not. So, what would happen when we run the program? Here is what I saw:

```
Exception in thread "main" java.util.InputMismatchException
  at java.util.Scanner.throwFor(Scanner.java:909)
  at java.util.Scanner.next(Scanner.java:1530)
  at java.util.Scanner.nextInt(Scanner.java:2160)
  at java.util.Scanner.nextInt(Scanner.java:2119)
  at ExamAvg.main(ExamAvg.java:32)
```

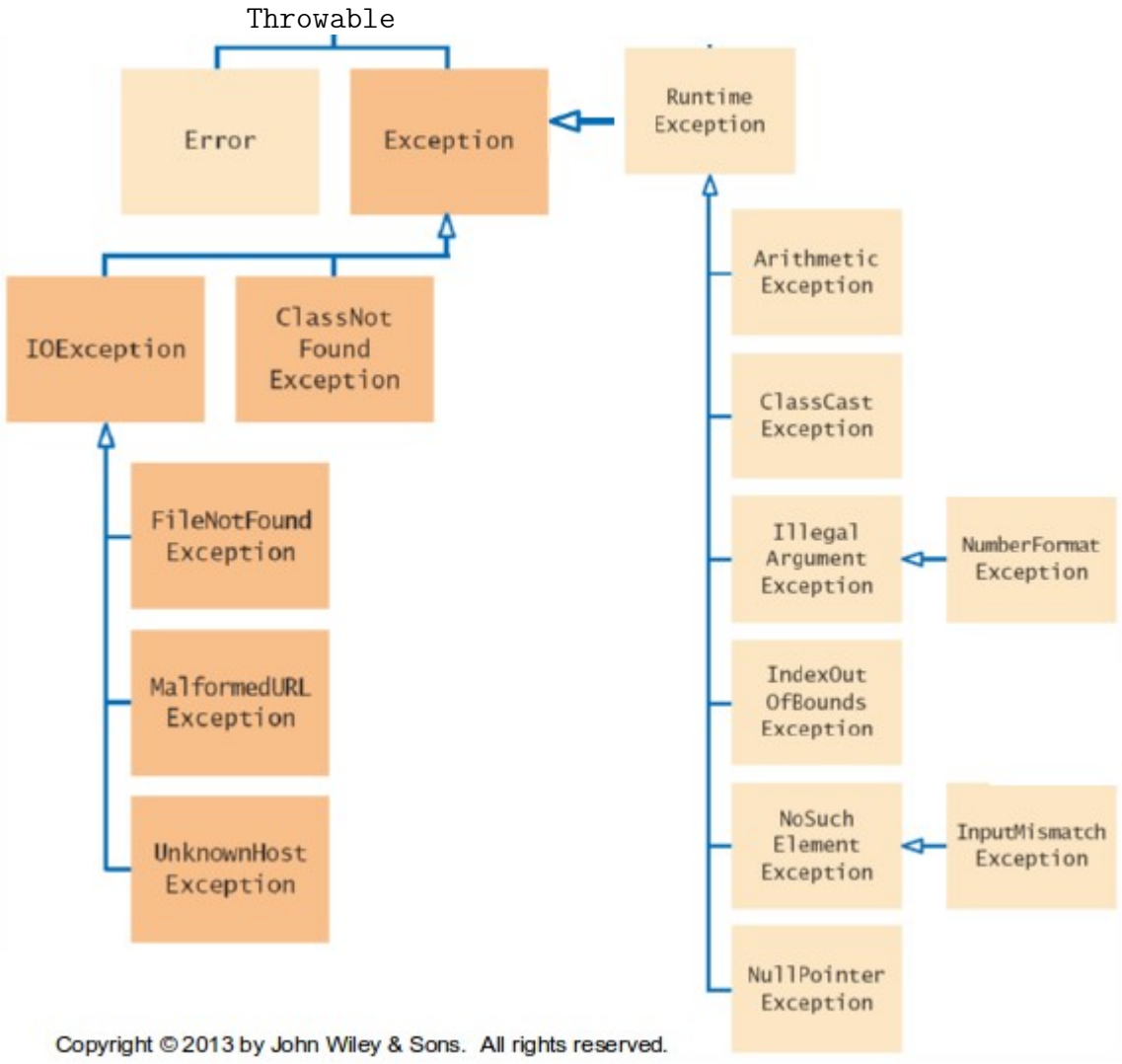
1. Exception Handling :: Introduction (continued)

An **exception** is an object of a class which gets **thrown** when something bad happens. The *InputMismatchException* is the one that got thrown when the characters in the input file ("Fred") did not match the expected data type that we were trying to read (**int**).

The **throws FileNotFoundException** clause of *ExamAvg.main*() told the Java compiler that *main*() would throw (i.e., **rethrow**) any exception objects of the *java.io.FileNotFoundException* class that were thrown while *main*() was executing. When *main*() throws an exception—of any type—the Java Virtual Machine (JVM) causes the program to terminate and generates an error message like the one you see above (this is called a **stack trace**).

In the real world, no Java program would ever have *main*() throw a *FileNotFoundException*. Rather, *main*() or some other method in the program, is supposed to **catch** and **handle** this exception. This is accomplished by writing an **exception handler**.

1. Exception Handling :: Exception Class Hierarchy



1. Exception Handling :: Exception Class Hierarchy (continued)

At the top of the hierarchy is a class named *java.lang.Throwable* which has two direct subclasses *java.lang.Error* and *java.lang.Exception*.

*Error*s indicate an abnormal condition outside of the application that should never have occurred (e.g., the JVM runs out of memory). Applications are neither required nor expected to catch *Error*s because the source of the *Error* is due to something that happened outside of the application. That is all we are will discuss about *Error*s.

*Exception*s and its subclasses are partitioned into **checked** and **unchecked exceptions**.

Unchecked exceptions are generally due to bugs in the code and indicate abnormal conditions that cannot be recovered from at runtime. They are generally not required to be handled and in fact, it is desirable to let them be thrown to the JVM so it will print a stack trace which will help you locate and correct the bug (they are called unchecked exceptions because the compiler will not check the code to see if an exception handler is in place).

Checked exceptions are abnormal conditions that arise in areas outside the immediate control of the program. They are not due to bugs and must be handled (they are called checked exceptions because the Java compiler will check to see if a proper handler is in place; if not, the code will not compile).