**Sum 12 CS-401 LAB #8 Exceptions**

**Tim Hoffman**

**Exceptions: checked/unchecked and try catch blocks**

**Your Lab assignment is at the bottom of this page**

Exceptions are events triggered by the Java runtime when an error occurs in your program. The Java runtime is always monitoring the execution of your program and is able to detect error conditions such as:

* + trying to call ***Integer.parseInt( "fubar" );*** (parseInt can't convert "fubar" to an int)
  + trying to open a disk file that is not there: ***FileReader = new FileReader( "non\_existent\_file.txt" );***
  + trying to declare or index an array with a bad index: ***int arr[] = new int[ -1 ];***
  + trying to allocate more memory than is available: ***int arr[] = new int[ 999999999 ];***

**un-checked exceptions**

If the programmer does not use the try/catch mechanism to handle exceptions herself, the runtime will react as follows when it detects an **Exception** condition. Execution stops right at the exception expression and control immediately returns control to the method that called the method where the exception occurred. If that calling method is not handling the exception, then control keeps going back up the call chain until either some method handles the exception or until control finally returns to the command line to the environment that invoked the program. The following demo illustrates an un-checked exception. An un-checked exception need not be caught or thrown. We will see soon what it means for an exception to need to be caught or thrown

[**ExceptionsDemo1.java**](http://www.cs.pitt.edu/~hoffmant/401/lab-08/ExceptionsDemo1.java)

|  |
| --- |
| /\*  ExceptionsDemo1.java  - illustrates unchecked exception  \* Integer.parseInt()  \* works same with Double.parseDouble() and the other wrapper classes  \*/  import java.io.\*;  public class ExceptionsDemo1  {  public static void main( String args[] )  {  System.out.println("calling foo1()");  foo1();  System.out.println("back from foo1()");  System.out.println("returning from main()");  } //END main  public static void foo1()  {  System.out.println("In foo1()");  System.out.println("calling foo2()");  foo2();  System.out.println("back from foo2()");  System.out.println("returning from foo1()");  }  public static void foo2()  {  System.out.println("In foo2()");  int i;  double d;  String s;  s = "1234";  i = Integer.parseInt( s );  System.out.println("You entered int: " + i );  s = "fubar";  i = Integer.parseInt( s ); // this will throw a "NumberFormatException"  System.out.println("You entered int: " + i );  System.out.println("returning from foo2()");  }  } //EOF |

**Output of the above program**

Notice that neither main(), foo1() or foo2() completes. As soon as the exception triggers, control returns from foo2() to foo1(). As soon as control returned to foo1() at the call to foo2(), the runtime saw that foo1() was not handline the exception either. Control then returned to main() at the call to foo1(). Again, main() was not handling the exception either so control returned to main's caller - the command line.

|  |
| --- |
| calling foo1()  In foo1()  calling foo2()  In foo2()  You entered int: 1234  Exception in thread "main" java.lang.NumberFormatException: For input string: "fubar"  at java.lang.NumberFormatException.forInputString(Unknown Source)  at java.lang.Integer.parseInt(Unknown Source)  at java.lang.Integer.parseInt(Unknown Source)  at ExceptionsDemo1.foo2(ExceptionsDemo1.java:44)  at ExceptionsDemo1.foo1(ExceptionsDemo1.java:25)  at ExceptionsDemo1.main(ExceptionsDemo1.java:16) |

**checked exceptions**

[**ExceptionsDemo2.java**](http://www.cs.pitt.edu/~hoffmant/401/lab-08/ExceptionsDemo2.java.txt)

|  |
| --- |
| /\*  ExceptionsDemo2.java  - illustrates a checked exception  \* Scanner throws FileNotFoundException  \* which is checked exception  \* FileNotFoundException a sub-type of IOException which is a sub-type of Exception  \* IOException is a checked exception  \* checked exceptions must either be  - handled with try/catch  - or be advertised with a throws FileNotFoundException (or IoException or Exception)  at the end of the signature of the enclosing method  \*/  import java.io.\*;  import java.util.\*;  public class ExceptionsDemo2  {  public static void main( String args[] ) throws IOException  {  // the following statement could throw a checked exception  // since it is a checked exception we must "advertise"  // the fact that we are not handling this exception in our code.  // Instead we "throw" the Exception. Thus the clause  // "throws IOException" after the main method's signature.  File f = new File( args[0] );  Scanner infile = new Scanner (f );  String token = infile.next(); // read a string from infile  System.out.println("1st string of the files was: " + token );  } //END main  } //EOF |

If we remove the **throws IOException** from the above program and recomplie it, the compiler will give the following errors.

|  |
| --- |
| C:\Documents and Settings\Administrator\Desktop\exceptions>javac ExceptionsDemo2.java  ExceptionsDemo2.java:26: unreported exception java.io.FileNotFoundException; must be caught or declared to be thrown  Scanner infile = new Scanner (f);  ^  1 error  C:\Documents and Settings\Administrator\Desktop\exceptions> |

Note the compiler cites the Scanner's constructor as throwing an exception that must be handled or declared to be thrown

**We now show you how to catch your exceptions**

[**ExceptionsDemo3.java**](http://www.cs.pitt.edu/~hoffmant/401/lab-08/ExceptionsDemo3.java.txt)

|  |
| --- |
| /\*  ExceptionsDemo3.java  - handles the exception thrown by Scanner with a try catch block  \*/  import java.io.\*;  import java.util.\*;  public class ExceptionsDemo3  {  public static void main( String args[] )  {  // the following statement could throw a checked exception  // since it is a checked exception we must "advertise"  // the fact that we are not handling this exception in our code.  // Instead we "throw" the Exception. Thus the clause  // throws FileNotFoundException after the main method's signature.  File f = new File( args[0] ); // File C'tor thorw IO exception if file does not exist  Scanner infile = null;  try  {  infile = new Scanner (f );  }  catch (FileNotFoundException fnfe )  {  System.out.println("Exception caught: " + fnfe + "\n");  System.exit(0);  }  String token = infile.next(); // read a string from infile  System.out.println("1st string of the files was: " + token );  } //END main  } //EOF |

There is one important difference between how a catch behaves and how a normal method behaves. Once we reach the bottom of the catch block, we drop through and execute the next statement in main below the catch. Unlike a method call, we do NOT return to the call site to execute the next line of the try block.

**Stacking catch blocks**

Often there are multiple statements in a try blocks that could throw an Exception. We could handle it several ways. The most shotgun approach would be to have one catch block looking for Exception (the general case) that will catch any sub-type Exception that occurs. Usually we want to know which specific type of Exception occurred and handle it distinctly. To do so we stack multiple catch blocks under the single try block. Each block should test for its specific Exception type such as NumberFormat or IOException etc. As a catch all, it is customary to put a plain Exceptions catch at the bottom just in case we get an unexpected unchecked exception. It is important to put the catch all Exception at the bottom, other wise the compiler will see you are catching the general before the specific and will give you a compilation error.

[**ExceptionsDemo4.java**](http://www.cs.pitt.edu/~hoffmant/401/lab-08/ExceptionsDemo4.java)

|  |
| --- |
| /\*  ExceptionsDemo4.java  - illustrates Stacking catch blocks  - Stream objects  \*/  import java.io.\*;  import java.util.\*;  public class ExceptionsDemo4  {  public static void main( String args[] )  {  // we declare our stream outside the try block  // other wise they become local vars  // we also have to initialize since possible exception  // will abort assignment statement and compiler will know the  // vars might not get initialized, and refuse to compile code  Scanner infile = null;  File f = null;  try  {  infile = new Scanner( new File( "input.txt" ) );  }  catch (FileNotFoundException ioe)  {  System.out.println("Caught Exception: " + ioe );  }  // NOTE THE MULTIPLE CATCH BLOCKS : ONLY 1 CAN EXECUTE  // THE FIRST EXCPTION DETECTED WILL JUMP TO ITS MATCHING BLOCK.  // THE EXCEPTION CLASS IS THE GENERAL CASE AND WILL CATCH ANY EXCEPTION  // THAT DOES NOT HAVE A SPECIFIC MATCHING BLOCK.  // ALWAYS PUT THE PLAIN EXCEPTION CASE LAST AS A CATCH ALL.  // IF YOU PUT IT ANYWHERE ELSE THEN SPECIFIC EXCEPTIONS WILL GO INTO THE  // GENERAL EXCEPTION BLOCK BEFORE GETTING TO MATCH ON THEIR SPECIFIC CASES.  System.out.println( "calling foo1()" );  try  {  String token = infile.next();  System.out.println("1st word of infile was: " + token );  int num = Integer.parseInt( token );  System.out.println("the int conversion: " + num );  }  catch ( NumberFormatException nfe ) // a specific Exception  {  System.out.println("Caught: " + nfe );  }  catch (Exception e ) // catch-all Exception is the most general Exception type  {  System.out.println("Caught General Exception:" + e );  }  System.out.println("returning from main()");  } //END main  } //EOF |

**Output of the above program**

|  |
| --- |
| calling foo1()  1st line of infile was: Hello World  Caught NumberFormatException: java.lang.NumberFormatException: For input string: "Hello World"  returning from main() |

**Lab Assignment**

|  |  |
| --- | --- |
| **Problem** | **Starter File** |
| **Recover from a FileNotFoundException using try catch** | [Ex\_1.java](http://www.cs.pitt.edu/~hoffmant/401/lab-08/Ex_1.java) |
| **Recover from an InputMismatchException using try catch** | [Ex\_2.java](http://www.cs.pitt.edu/~hoffmant/401/lab-08/Ex_2.java) |