Java Exceptions

Source	http://www.smartdataprocessing.com/lessons/l5.htm

Let's say a Java class reads a file with the customer's data. What's going to happen if someone deletes this file?

Will the program crash with that scary multi-line error message, or will it stay alive displaying a user friendly message like this one: "Dear friend, for some reason I could not read the file customer.txt. Please make sure that the file exists"? In many programming languages, error processing depends on the skills of a programmer.

Java forces software developers to include error processing code, otherwise the programs will not even compile.

Error processing in the Java is called exception handling.

You have to place code that may produce errors in a so-called try/catch block:

```
try{
  fileCustomer.read();
  process(fileCustomer);
}
catch (IOException e){
  System.out.println("Dear friend, I could not read the file
customer.txt...");
}
```

In case of an error, the method read() throws an exception. In this example the catch clause receives the instance of the class IOException that contains information about input/output errors that have occured. If the catch block exists for this type of error, the exception will be caught and the statements located in a catch block will be executed. The program will not terminate and this exception is considered to be taken care of.

The print statement from the code above will be executed only in case of the file read error. Please note that method process() will not even be called if the read fails.

Reading the Stack Trace

If an unexpected exception occurs that's not handled in the code, the program may print multiple error messages on the screen. Theses messages will help you to trace all method calls that lead to this error. This printout is called a stack trace. If a program performs a number of nested method calls to reach the problematic line, a stack trace can help you trace the workflow of the program, and localize the error.

Let's write a program that will intentionally divide by zero:

```
class TestStackTrace{
  TestStackTrace()
  {
    divideByZero();
  }

  int divideByZero()
  {
    return 25/0;
  }
```

```
  static void main(String[]args)
  {
    new TestStackTrace();
  }
}
```

Below is an output of the program - it traced what happened in the program stack before the error had occurred. Start reading it from the last line going up. It reads that the program was executing methods main(), init() (constructor), and divideByZero(). The line numbers 14, 4 and 9 (see below) indicate where in the program these methods were called. After that, the ArithmeticException had been thrown - the line number nine tried to divide by zero.

```
c:\temp>java TestStackTrace
  Exception in thread "main"
  java.lang.ArithmeticException: / by zero
     at TestStackTrace.divideByZero(TestStackTrace.java:9)
     at TestStackTrace.(TestStackTrace.java:4)
     at TestStackTrace.main(TestStackTrace.java:14)
```

Exception Hierarchy

All exceptions in Java are classes implicitely derived from the class Throwable which has immediate subclasses

Error and Exception.

Subclasses of the class Exception are called listed exceptions and have to be handled in your code.

Subclasses of the class Error are fatal JVM errors and your program can't fix them. Programmers also can define their own exceptions.

How you are supposed to know in advance if some Java method may throw a particular exception and the try/catch block should be used? Don't worry - if a method throws an exception and you did not put this method call in a try/catch block, the Java compiler will print an error message similar to this one:

"Tax.java": unreported exception: java.io.IOException; must be caught or declared to be thrown at line 57.

Try/Catch Block

There are five Java keywords that could be used for exceptions handling: try, catch, finally, throw, and throws.

By placing a code in a try/catch block, a program says to a JVM: "Try to execute this line of code, and if something goes wrong, and this method throws exceptions, please catch them, so that I could report this situation to a user." One try block can have multiple catch blocks, if more than one problem occurs. For example, when a program tries to read a file, the file may not be there - FileNotFoundException, or it's there, but the code has reached the end of the file - EOFException, etc.

```
public void getCustomers(){
  try{
    fileCustomers.read();
  }catch(FileNotFoundException e){
    System.out.println("Can not find file Customers");
  }catch(EOFException e1){
    System.out.println("Done with file read");
  }catch(IOException e2){
    System.out.println("Problem reading  file " +
                                e2.getMessage());
  }
```

```
}
```

If multiple catch blocks are handling exceptions that have a subclass-superclass relationship (i.e. EOFException is a subclass of the IOException), you have to put the catch block for the subclass first as shown in the previous example.

A lazy programmer would not bother with catching multiple exception, but will rather write the above code like this:

```
public void getCustomers(){
  try{
    fileCustomers.read();
  }catch(Exception e){
    System.out.println("Problem reading  file " +
                       e.getMessage());
  }
}
```

Catch blocks receive an instance of the Exception object that contains a short explanation of a problem, and its method getMessage() will return this info. If the description of an error returned by the getMessage() is not clear, try the method toString() instead.

If you need more detailed information about the exception, use the method printStackTrace(). It will print all internal method calls that lead to this exception (see the section "Reading Stack Trace" above).

Clause throws

In some cases, it makes more sense to handle an exception not in the method where it happened, but in the calling method. Let's use the same example that reads a file. Since the method read() may throw an IOException, you should either handle it or declare it:

```
class CustomerList{
  void getAllCustomers() throws IOException{
    file.read(); // Do not use try/catch  if you are not handling exceptions
here
  }

  public static void main(String[] args){
    System.out.println("Customer List");

    try{
      // Since the  getAllCustomers()declared exception,
      // we have to either  handle  it over here, or re-
      // throw it (see the throw clause explanation below)

     getAllCustomers();
   }catch(IOException e){
     System.out.println("Sorry, the  Customer List is not
                                                available");
  }
}
```

In this case, the IOException has been propagated from the getAllCustomers() to the main() method.

Clause finally

A try/catch block could be completed in different ways

1. the code inside the try block successfully ended and the program continues,
2. the code inside the try block ran into a return statement and the method is exited,
3. an exception has been thrown and code goes to the catch block, which handles it
4. an exception has been thrown and code goes to the catch block, which throws another exception to the calling method.

If there is a piece of code that must be executed regardless of the success or failure of the code, put it under the clause finally:

```
try{
  file.read();
}catch(Exception e){
  printStackTrace();
}
finally{
 file.close();
}
```

The code above will definitely close the file regardless of the success of the read operation. The finally clause is usually used for the cleanup/release of the system resources such as files, database connections, etc..

If you are not planning to handle exceptions in the current method, they will be propagated to the calling method. In this case, you can use the finally clause without the catch clause:

```
void myMethod () throws IOException{
  try{
    file.read();
  }
  finally{
    file.close();
  }
}
```

Clause throw

If an exception has occurred in a method, you may want to catch it and re-throw it to the calling method. Sometimes, you might want to catch one exception but re-throw another one that has a different description of the error (see the code snippet below).

The throw statement is used to throw Java objects. The object that a program throws must be Throwable (you can throw a ball, but you can't throw a grand piano). This technically means that you can only throw subclasses of the Throwable class, and all Java exceptions are its subclasses:

```
class CustomerList{

  void getAllCustomers() throws Exception{
    try{
      file.read(); // this line may throw an exception
    } catch (IOException e) {
      // Perform some internal processing of this error, and _
      throw new  Exception (
        "Dear Friend, the file has problems..."+
                                    e.getMessage());
      }
    }

    public static void main(String[] args){
      System.out.println("Customer List");
```

```
      try{
        // Since the  getAllCustomers() declares an
        // exception, wehave  to either  handle  it, or re-throw it
         getAllCustomers();
      }catch(Exception e){
        System.out.println(e.getMessage());
      }
   }
}
```

User-Defined Exceptions

Programmers could also create user-defined exceptions, specific to their business. Let's say you are in business selling bikes and need to validate a customers order. Create a new class TooManyBikesException that is derived from the class Exception or Throwable, and if someone tries to order more bikes than you can ship - just throw this exception:

```
class TooManyBikesException extends Exception{
  TooManyBikesException (String msgText){
     super(msgText);
  }
}

class BikeOrder{
  static  void validateOrder(String bikeModel,int quantity) throws
TooManyBikesException{

  // perform  some data validation, and if you do not like
  // the quantity for the specified model, do  the following:

  throw new TooManyBikesException("Can not ship" +
    quantity+" bikes of the model " + bikeModel +);
  }
}

class Order{
   try{
     bikeOrder.validateOrder("Model-123", 50);

     // the next line will be skipped in case of an exception
     System.out.println("The order is valid");
   } catch(TooManyBikes e){
      txtResult.setText(e.getMessage());
   }
}
```

In general, to make your programs robust and easy to debug, you should always use the exception mechanism to report and handle exceptional situations in your program. Be specific when writing your catch clauses - catch as many exceptional situations as you can predict. Just having one catch (Exception e) statements is not a good idea.

~~~ End of Article ~~~