

# Exceptions

(CISS-111 William Jojo)

## Overview

When we develop a Java program there are many opportunities to fix problems of syntax and logic. Some opportunities are available through error messages from the compiler which refuses to allow us to move forward until the syntax errors are corrected. Other opportunities are from our own testing where logic errors are identified and corrected as a result of viewing results we know to be incorrect.

The problem of runtime errors, or *exceptions* is difficult to perceive at first if you have not given them much thought or dismissed out of hand the occasional `InputMismatchException` or `NumberFormatException`. If these have troubled you, there is a simple mechanism by which we can control the exceptions instead of the exceptions causing our program to abruptly halt with a stack trace.

## Handling Exceptions

The use of types in programming will eventually lead to an issue of compatibility. It is easy to visualize an integer becoming a floating point number – simply add a period and a zero on the end of the integer and you now have a floating point quantity. The compiler will allow an `int` to be assigned to a `float` or `double` type. Ask the compiler to look the other way when assigning a `double` to an `int` variable and your program will never get off the ground until you fix the “possible loss of precision” error that the compiler has identified. There are basically two choices:

1. Make the double an int permanently by changing variable types as needed and make them all the same type.
2. Use a cast to temporarily alter the type and acknowledge that we realize there is a potential for a calculation error and we accept the consequences.

This method of control over our program logic and data works very well. However, what

happens when we use the `nextInt()` method of the `Scanner` class and the data file or user provides a `double` quantity? We would expect a `InputMismatchException` to occur. The same would be true if the user enters "xyz" when our program was calling `nextDouble()`. What we need is a way to control the program and catch these exceptions before they terminate the program. This is necessary for a several reasons:

- The program may be able to recover from the error and retry the action.
- The program can detect the error and perform a controlled shutdown rather than an abrupt halt.
- The program may be easier to manage and will be written more professionally.

## The `try/catch/finally` Block

When an exception occurs, Java creates an object of the specific exception class. Some of these classes are the class `NumberFormatException` and the class `InputMismatchException`. All exception classes are subclasses of the superclass `Exception`.

We can surround a set of statements that we know could generate an exception with the `try` keyword, then `catch` exceptions and perform work based on the type of exception that has occurred. We can also use the `finally` block to cause work to be done whether an exception occurred or not.

Consider the following:

```
Scanner kb = new Scanner(System.in);
int num;

try
{
    System.out.print("Enter a number: ");
    num = kb.nextInt();
}
catch (InputMismatchException imeRef)
{
    System.out.println("Your input was not an integer: " +
        imeRef.toString());
}
```

```
}
```

This code block enables us to try reading an integer from the user and then catch an exception should it occur. Note that the `InputMismatchException` object is referenced with the `imeRef` parameter. Let us inspect another piece of code:

```
Scanner kb = new Scanner(System.in);
int line=0, val;
String s;

while (kb.hasNext())
{
    s = kb.next();
    line++;
    try
    {
        val = Integer.parseInt(s, 16);
    }
    catch (Exception eRef)
    {
        System.out.println("Possible data corruption at line " +
                           line);
        System.exit(1);
    }
    finally
    {
        System.out.println("The data was \"" + s + "\"");
    }
}
```

This example uses the superclass `Exception` object with the `eRef` reference parameter. The `finally` block is being used here as a debug tool. Recall that the `finally` block will be executed regardless of the presence of an exception.

It is important to realize that the `try` block ceases execution of statements as soon as an exception occurs and the the `catch` blocks are only executed when an exception occurs. Never place the `Exception` superclass catch above any other `Exception` subclasses, otherwise the superclass `Exception` will always be applied. In other words, the *order* of your `catch` blocks is important.

## The Exception Hierarchy

The `Exception` class is a subclass of the `Throwable` class which is a subclass of the `Object` class. *Table 1* shows some of the methods of the `Throwable` class and *Table 2* shows the constructors for the `Exception` class. Since the class `Exception` is a subclass of `Throwable`, it inherits the `Throwable` methods.

Throwable Class Constructors and Methods
<b>public Throwable()</b> Default constructor. Creates a <code>Throwable</code> object with no message string.
<b>public Throwable(String msg)</b> Creates a <code>Throwable</code> object with the message string <code>msg</code> .
<b>public String getMessage()</b> Returns the message stored in the <code>Throwable</code> object.
<b>public void printStackTrace()</b> Print the stack trace for viewing the sequence of method calls.
<b>public void printStackTrace(PrintWriter stream)</b> Print the stack trace for viewing the sequence of methods calls to the stream named <code>stream</code> .
<b>public String toString()</b> Return string representation of the <code>Throwable</code> object.

*Table 1: Constructors and Methods of the the Throwable class.*

Exception Class Constructors
<b>public Exception()</b> Default constructor. Creates an <code>Exception</code> object.
<b>public Exception(String msg)</b> Creates an <code>Exception</code> object with the message string <code>msg</code> .

*Table 2: Constructors of the Exception class.*

The exception hierarchy is rather simple to follow. Everything is ultimately a subclass of `Exception`. There are many subclasses from `ClassNotFoundException` to the `RuntimeException` and `IOException`.

The `RuntimeException` is a rather large subset of exceptions that can only occur while your program is running. As a result, you will see many subclasses listed under the `RuntimeException`.

The `Exception` class is defined in the `java.lang` package. Other subclasses of `Exception` are also defined in `java.lang` as well as `java.util`, `java.io` and `java.awt` to name a few.

Some exception classes defined in `java.lang`:

- `Exception`
  - `ClassNotFoundException`
  - `CloneNotSupportedException`
  - `IllegalAccessException`
  - `InstantiationException`
  - `InterruptedException`
  - `NoSuchFieldException`
  - `NoSuchMethodException`
  - `RuntimeException`
    - `ArithmeticException`
    - `ClassCastException`
    - `IllegalArgumentException`
      - `NumberFormatException`
    - `IndexOutOfBoundsException`
      - `ArrayIndexOutOfBoundsException`
      - `StringIndexOutOfBoundsException`
    - `NullPointerException`

Some exceptions classes defined in `java.util`:

- `Exception`
  - `RuntimeException`
    - `EmptyStackException`
    - `NoSuchElementException`
      - `InputMismatchException`

Some exceptions classes defined in `java.io`:

- `Exception`
  - `IOException`
    - `EOFException`
    - `FileNotFoundException`

The online Java documentation at the Sun site is the best place to determine what, if any, exceptions will be thrown by a method or constructor.

As final note about handling multiple exceptions, you could always catch the superclass `Exception` and then process an `if/else-if/else-if/else` against the superclass reference parameter like so:

```
try
{
    // sequence of input and
    // arithmetic statements that could
    // produce Scanner or division by zero
    // errors.
}
catch (Exception eRef)
{
    if ( eRef instanceof ArithmeticException)
        // statements
    else if (eRef instanceof InputMismatchException)
        // statements
    else
        // statements
}
```

## Checked and Unchecked Exceptions

Java has broken down the predefined exceptions into two categories: *checked* and *unchecked*. Checked exceptions are actually done by the compiler, that is, any exception that the compiler can successfully identify.

The `FileNotFoundException` is a prime example as is the `IOException`. Recall that this exception must be thrown often by the `main()` method of your program. This results in a

“throws `FileNotFoundException`” at the end of the `main()` declaration for any program that uses `FileReader` or `PrintWriter`. Of course, we are simply ignoring the exception since we have not actually set up a `try/catch` block to properly deal with the potential problem.

Unchecked exceptions are those that you may choose to catch or not. These are also exceptions that the compiler simply cannot accurately detect. These types of exceptions include `InputMismatchException`, `NumberFormatException` and `ArithmeticException`. Essentially, all of the `RuntimeException` subclasses are unchecked and should be checked by you, the programmer, if you wish to make your programs more reliable and to achieve reasonable recoverability.

When writing your methods keep in mind that you may ignore, throw or re-throw a caught exception. To throw or re-throw an exception, you simply need to throw an instantiated exception object. For example:

```
throw new Exception("Text associated with the exception");
```

or with:

```
throw new ArithmeticException("bad math skills!");
```

In the case of re-throwing:

```
try
{
    // statements
}
catch (InputMismatchException imeRef)
{
    throw imeRef;
}
```

## Creating an Exception Class

Now we will build a program that creates an exception class for improperly formatted data. This program requires that input contain at least two fields in a string a data separated by a colon.

We will start with the class that defines a `DataFormatException`. This exception extends the `Exception` class and defines two constructors in a similar fashion to `Exception`. The constructors simply call the superclass constructor `super()`. This class is shown in *Example 1*.

```
public class DataFormatException extends Exception
{
    public DataFormatException ()
    {
        super("Data must contain at least one semicolon");
    }

    public DataFormatException(String msg)
    {
        super(msg);
    }
}
```

*Example 1: Example defining the DataFormatException class.*

The class that will exercise our new `DataFormatException` class will also demonstrate how to catch and recover from the exception. In this particular case we are demonstrating that you do not need to simply stop execution just because something bad has occurred.

The program `excepttest` in *Example 2* uses an EOF-based `while` loop. You could, of course, use a `do-while`. Consider how your program needs to be constructed in order to facilitate moving forward in the event of an exception.

This program also throws the exception within the `try` block which is subsequently caught in the `catch` block. This demonstrates the tight level of control that the `try/catch` block offers a programmer.

Notice how the while loop contains the try/catch block and how there is no call to `System.exit()` as a result of the exception occurring. Remember that our goal is to keep the program going. Any entry of data without a colon will result in the exception being displayed.

```
import java.util.*;

public class exceptttest
{
    static Scanner kb = new Scanner(System.in);

    public static void main(String[] args)
    {
        String s;
        int pos;

        System.out.println("\nEnter data with at least one colon:");
        while (kb.hasNext())
        {
            try
            {
                s = kb.nextLine();
                pos = s.indexOf(":");
                if ( pos == -1 )
                    throw new DataFormatException();
                else
                    System.out.println("The value entered is " + s);
            }
            catch (DataFormatException dfeRef)
            {
                System.out.println(dfeRef);
            }
            catch (Exception eRef)
            {
                System.out.println(eRef);
            }

            System.out.println("\nEnter data with at least one colon:");
        }
    }
}
```

*Example 2: Program to exercise the DataFormatException class.*