Chapter 7: Exception Handling
# Table of Contents

Objectives.................................................................................................................................3  
Section 1: Introduction ..................................................................................................................4  
Exception Handling .......................................................................................................................5  
Exception Types ...........................................................................................................................6  
Handling Exceptions ....................................................................................................................7  
Exception Handling ......................................................................................................................8  
Catch Statement ............................................................................................................................9  
Exception Handling Example ......................................................................................................10  
Stack Trace ...................................................................................................................................11  
Exception Handling Nesting .........................................................................................................12  
Order of Catch Blocks ..................................................................................................................13  
Passing the Buck ...........................................................................................................................15  
Creating an Exception Class .........................................................................................................16  
Generating Exceptions ................................................................................................................17  
Summary .......................................................................................................................................18
Objectives

After completing this unit you will be able to:

- Handle exceptions in Java
- Learn when it’s appropriate to use them
- Use the try and catch blocks
- Create new exceptions
Section 1: Introduction
Exception Handling

A traditional approach to handling error messages is to return an error code from a procedure. The function will normally return a zero if the function executed without an error. If an error did occur then a non-zero value would be returned.

The problem with this approach is that the calling of the function may either:

- Be unaware that the function returns an error code (e.g. C’s printf function)
- Forget to check for an error, or
- Ignore the error all together

When the error is not catch, the continued execution of the program can lead to unpredictably and possibly disastrous consequences.

An alternative to this method is to use “catch” errors. Most modern block structured languages uses this approach. This technique requires less coding, is more readable and is very robust. When a routine detects an error, it “throws” an exception object. The exception object is then returned to the calling routine which then catches and handles the error.

The following code sequence illustrates the process. The method displayProjectory calls the method computePosition. Within computePosition, an error is detected. A PositionError object is created and returned to displayProjectory. Within displayProjectory the error is catch and handled.

```java
public computePosition() {
    …
    if(some condition) {
        // Create PositionError object
        // Return object;
    }
    …
}

public displayProjectory() {  
    computePosition();
    // Catch and handle the error.
}
```

The actual syntax used by Java is explained later.
Exception Types

An exception is an instance of a class derived directly or indirectly from the Throwable class. Two predefined Java classes are derived from Throwable: Error and Exception. From the Exception class is derived a RuntimeException class. As we will see shortly, programmer defined exceptions are derived from the Exception class.

There are numerous pre-defined errors that are derived from the Error and RuntimeException. There is little that a programmer will do with the Error object. These exception represent problems with the JVM and are normally can’t be recovered from. The Exception class is different. Two classes derive from Exception:

- Exceptions that must be caught
- RuntimeException

RuntimeException are exception such as division by zero and an array subscripting error. These do not have to be caught but like Error exceptions, if they are not caught, the program will terminate.

All exceptions derived from the Exception class that are not derived from the RuntimeException class must be handled in code or the code will not compile cleanly. There will be compile time errors.
Handling Exceptions

Any code sequence that uses a method that throws an exception of this type must either:

- Catch the exception or
- "Pass the buck"

The first technique uses a combination of a try and catch block and is elaborated on next.

The second technique is used when the current method is not the appropriate place to handle the exception. It allows the exception to be propagated higher into the sequence of method calls. This technique will also be discussed in more detail shortly.

Uncaught exceptions are propagated to the next higher context until they reach main where an error message and stack trace will be printed.
Exception Handling

Java implements exception handling using try and catch blocks. The try blocks are used to surround code that might fail and is followed by zero or more catch blocks.

The catch blocks are added after a try block to “catch” exceptions. The catch statements provide blocks of code to “handle” the error. A finally clause can optionally be used after the catch clauses.

```java
try {
    inString = is.readLine();
    value = Integer.parseInt(inString);
    ...
    potentialError = false;
}

catch (IOException e) {
    System.out.println("I/O Exception occurred");
}

catch (NumberFormatException e) {
    System.out.println("Bad format, try again...");
}

finally {
    // Perform any necessary clean-up code
    // Close files, sockets.
    // Unlink from any Databases
}
```
**Catch Statement**

The **catch** statement has exactly one argument. The **catch** statement will trap the exception if:

- It matches the exception type
- The parameter type is a base of the exception type
- The parameter is an interface that the exception type implements

Only the first catch statement that matches the exception will execute. If no matches are made, the method will terminate.

The finally block will always execute unless the `System.exit` method is invoked. In that case, the program immediately terminates.

The process of handling an error is up to the programmer. It may be as simple as displaying an error message or it can be quite complex. The programmer may use the error object to obtain more information about the error. Based on this and other information, the programmer may set flags that result in the method being invoked again.

In this code sequence, errors are assumed to be present when the try block is entered. If an error is generated it is caught and handled by the catch block. Since the errorsPresent is still set to true, the try block will be repeated. However, if no errors occur, at the end of the try block the errorsPresent flag is set to false which will allow the program to execute the while loop and continue executing.

```java
boolean errorsPresent;

...  
errorsPresent = true;  // Assume errors will be generated  
while (errorsPresent) {  
    try {  
        ...  
        errorsPresent = false;  // No errors exist so set to false  
    }
    catch (someException e) {  
        // Process error
    }
}
```
Exception Handling Example

This error handling approach uses a do while statement instead of a while statement.

```java
import java.io.*;

public class ExceptionDemo {

    public static void main(String args[]) {
        DataInputStream is = new DataInputStream(System.in);
        boolean potentialError = true;
        String inString;
        do {
            try {
                inString = is.readLine();
                value = Integer.parseInt(inString);
                ...
                potentialError = false;
            }
            catch (IOException e) {
                System.out.println("I/O Exception occurred");
            }
            catch (NumberFormatException e) {
                System.out.println("Bad format, try again...");
            }
            finally {
                // Perform any necessary clean-up code
                // Close files, sockets.
                // Unlink from any Databases
            }
        } while (potentialError);
    }
}
```
Stack Trace

The `printStackTrace` method is a method of the Throwable class which will display the program stack at that point in the program. The method can help you better determine the problems with your program by pinpointing the line and method that caused the program. You have seen this method in action before, whenever you had an unhandled runtime exception. The method is automatically called when an exception is not handled.

The ExceptionDemo program illustrates the explicit use of the method.

```java
import java.io.*;

public class ExceptionDemo {
    public void foo3() {
        try {
            //statements
            throw new Exception();
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
    public void foo2() { foo3(); }
    public void foo1() { foo2(); }
    public static void main(String args[]) {
        new ExceptionDemo().foo1();
    }
}
```

The output is shown below:

```
java.lang.Exception
    at ExceptionDemo.foo3(ExceptionDemo.java:8)
    at ExceptionDemo.foo2(ExceptionDemo.java:16)
    at ExceptionDemo.foo1(ExceptionDemo.java:20)
    at ExceptionDemo.main(ExceptionDemo.java:25)
Press any key to continue . . .
```
Exception Handling Nesting

Exception mechanism can be nested. This can become necessary when methods are used in a catch block that also throws exceptions.

try {
    // Code that may throw an exception
}

catch (someException e) {
    try {
        // Code to handle the exception
    }
    catch (anException e) {
        // Code to handle the nested exception
    }
}

catch (someOtherException e) {
    // Code to handle the exception
}
Order of Catch Blocks

The order in which catch blocks are listed after a try block can be significant. When an exception is thrown, the exception object is compared to the catch blocks in the order that they are listed. The comparison checks to see if the thrown exception is a type of the exception in the catch block.

For example, if a FileNotFoundException is thrown, it will match either a catch block that has a IOException or a FileNotFoundException exception. Since the comparison is stopped when the first match is found, if the IOException catch block is listed before the FileNotFoundException catch block. The FileNotFoundException block will never be executed.

Consider the following hierarchy of exception classes:

![Exception Class Hierarchy Diagram]

Given the following code sequence:

```java
try {
    ...
}
catch (AException e) {...}
catch (BException e) {...}
catch (CException e) {...}
catch (DException e) {...}
```

If an exception is thrown that is one of these types of exceptions the first catch block will always be executed. This is because An AException, BException, CException or a DException are all of type AException. The other catch blocks will never be executed.

The general rule is always to list the "most-derived" exceptions first. The following is the correct way of listing the exceptions.
try {
  ...
}
catch (DException e) {...}
catch (BException e) {...}
catch (CException e) {...}
catch (AException e) {...}

Notice that it doesn't make any difference with this hierarchy of exceptions whether the BException immediately precedes or follows the CException since they are at the same level.
Passing the Buck

A second technique for handling exceptions is to not actually handle them but to "pass the buck" up to the method that called the current method. To indicate that the current method does not try to catch certain exceptions, the `throws` keyword is used.

```java
public void someMethod() throws IOException, NumberFormatException {
    DataInputStream is = new DataInputStream (System.in);
    String inString;

    inString = is.readLine();
    value = Integer.parseInt (inString);
}
```

Exceptions should be handled where it makes the most sense, which is not necessarily in the method that generated it.
Creating an Exception Class

Within an application, there are usually some application specific exceptions that can occur. For example, if an Account class does not permit its balance to go negative, then it may want to throw an exception instead of displaying an error message. Displaying an error message within the withdraw method may seem like what you would want to do. However, this decision should be left up to the method that calls the withdraw method. The programmer may need to display the message using System.out, in a message box, or possibly not at all.

Exceptions can be created fairly easily. This involves extending the Exception class or other derived classes. By creating application specific exceptions, graceful handling of anomalies can be achieved.

To create a new exception class:

- Derive the class from the Exception class
- Add a default constructor
- Add a constructor that accepts a single String argument and calls its base class

```java
class NewException extends Exception {
    NewException () {
    }

    NewException (String message) {
        Super (message);
    }
}
```
Generating Exceptions

The `throw` statement is used to explicitly generate an exception.

```java
if (index > 100) {
    throw new NewException("index executed upper limit")
}
```

The statement is used in a method where an error of that type can occur. This method must declare that it can throw an error of that type by using the `throws` keyword.

```java
public void myFunc() throws NewException {
    ...
    if (index > 100) {
        // problem encountered !!!
        throw new NewException("index executed upper limit")
    }
    ...
}
```

The method will immediately return as the `throw` keyword acts as a return statement returning control to the method that called it. It is now up to the calling method to handle that exception. If the method does not throw an exception, then it continues doing its job.
Summary

- Exceptions can aid in your program design adding robustness and readability
- **try, catch, throw** and **finally** are the keywords used for implementing exception handling
- An application can define its own exceptions
- A method that throws an exception must either catch the exception or declare that it may throw an exception
- Exception handling may be nested where we find a try block inside of a catch block