Exception Handling

1 Introduction

Exception handling enables programs to handle some of the exceptional situations and continue with normal execution.

Exceptions are thrown from methods. The calling methods can

- catch and handle exceptions, or
- re-throw exceptions.

If a thrown exception is re-thrown all the way up to the Java virtual machine, or ignored (not all exception types can be ignored) then the program crashes.

Exception are not replacements for testing and data validation. They should only be used if the method cannot/should not decide how a given exceptional situation should be handled.
Example:

Scenario: We try to remove an item from a linked list. The remove method determines that the item is not on the list.

Question: Should the method throw an exception?

Example:

Scenario: We are evaluating postfix expressions. The method takes a String object containing a postfix expression as a parameter. It parses the input string and evaluates the expression. During evaluation, the method determines that there are too few operators in the input string.

Question: Should the method throw an exception?
2 Exception Types in Java

As everything else in Java, exceptions are classes. Java provides a very large set of different exception classes to indicate different possible problems. We can use any of these exception classes in your own program.
All Java exception classes are instances of Throwable (think things that can be thrown), [http://docs.oracle.com/javase/8/docs/api/java/lang/Throwable.html](http://docs.oracle.com/javase/8/docs/api/java/lang/Throwable.html). Throwable is further subdivided into Error ([http://docs.oracle.com/javase/8/docs/api/java/lang/Error.html](http://docs.oracle.com/javase/8/docs/api/java/lang/Error.html)) and Exception ([http://docs.oracle.com/javase/8/docs/api/java/lang/Exception.html](http://docs.oracle.com/javase/8/docs/api/java/lang/Exception.html)) classes.
2.1 Error Subclass of Throwable

Error class represents errors that normal applications should not even try to handle. They indicate serious problems from which an application cannot recover.

**Example:** LinkageError indicates that the program depends on some other class that has changed since the last compilation of the application and is no longer compatible with the current application.

**Example:** VirtualMachineError class indicates that the Java Virtual Machine is broken or has run out of resources necessary for it to continue operating (there is absolutely nothing that our program can do about it).

2.2 Exception Subclass of Throwable

Exception class represents errors that applications can try to handle (not all of those errors can be recovered from though). When we write our own exception classes they are subclasses of the Exception class. We see these exceptions whenever the program crashes (due to bugs or user errors).
2.2.1 Unchecked Exceptions - RuntimeExceptions

One of the big subclasses of Exception class is the class RuntimeException. RuntimeErrors are the only class of exceptions that our programs are allowed to ignore. They are, unfortunately, so abundant, that it would be almost impossible to try to catch and handle these. They are referred to as unchecked exceptions - the compiler does not make sure that you check for those (because the compiler does not know when they can occur either).

The RuntimeErrors that we see most often are

- ArithmeticException (try to divide by zero),
- NullPointerException (well, we all know how often that one comes out of nowhere),
- IndexOutOfBoundsException (try to access an array location that does not exist).

2.2.2 Checked Exceptions

Any other subclass of the Exception class has to be handled. Our code has to either catch it and handle it, or declare that our method might throw one of those exceptions.
3 Exception Handling - Syntax

3.1 Catching Exceptions

When a program is running the exception can be thrown either by other parts of the program or by some of the methods from Java provided classes. In either way, catching exceptions works the same.

In order to catch an exception, we need to surround the code that might potentially cause an exception by a `try ... catch ...` block.
try {
    // A - code before exception might occur
    // B - code in which an exception might be thrown
    // C - code after the exception might be thrown
}

catch (ExceptionType1 ex ) {
    // handle an exception of type Exception1
}

catch (ExceptionType2 ex ) {
    // handle an exception of type Exception2
}

... catch (Exception ex ) {
    // handle an exception of type Exception
}

finally {
    // code that is executed regardless of what exceptions were
    // thrown and caught
    // (usually used to handle finalizing of I/O operations
    // so that files are closed correctly and do not get corrupted)
}
The `try{ ... }` block is divided into three parts: A, B and C. Part A always executes. If an exception occurs during execution of part B, the rest of the `try{ ... }` block (i.e., part C) is skipped and the execution continues into the `catch{ ... }` blocks.

The first `catch{ ... }` block that matches the type of exception thrown is entered. Remember that any subclass exception type matches its superclass exception class. Only one of the `catch{ ... }` catch blocks is ever executed.

Each `try{ ... }` block has to be followed by at least one `catch{ ... }` block or `finally{ ... }` block.
3.2 Throwing Exceptions

In order to throw an exception, we need to create an exception object to be thrown. That object can carry information to the calling method about what problem occurred when the exception was thrown.

```java
returnType methodName ( ... ) throws ExceptionType {
    ...
    if (error occurs)
        throw new ExceptionType ( "message to the calling method" );
}
```

The keyword throws followed by the exception type is mandatory for checked exception types (it is not used if the method throws unchecked exception).

Notice that there are two different keywords: throws and throw. The first one is just declaration indicating the type of exception that the method might throw. The second one performs the actual action of throwing the error. If the method throws more than one type of exception the keyword throws should be followed by a comma-separated list of exception types.
4 Getting Information from Exceptions

The exception objects are like little messengers carrying the bad news from the method that throws them to the method that catches them. Those messages can be used to determine what went wrong. We will often make use of the following methods that are inherited by all exception classes from the Throwable class:

- `getMessage()` returns the message that was passed to the constructor when the exception object was created
- `toString()` returns the full name of the exception concatenated with the message returned by the `getMessage()` call
- `printStackTrace()` prints the stack trace on the console

WARNING: if our exception handler does only that, then effectively we are not handling the exception - that’s what JVM would have done
5 Ordering the `catch{ ... }` Blocks

As we suggested before, the order of the catch blocks (if more than one is specified) is important. A `catch{ ... }` block for a superclass of a given exception type should be listed after a `catch{ ... }` block for a subclass type. The most specific exception type should be listed first, with the superclass of all exception types, `Exception`, listed last (you do not need to go all the way to the top of the inheritance hierarchy of course).
6 Exercises to Try

What RuntimeException will the following programs throw, if any?

```java
public class Test {
    public static void main (String [] args ) {
        System.out.println( 1 / 0 ) ;
    }
}
```
What RuntimeException will the following programs throw, if any?

```java
public class Test {
    public static void main (String [] args ) {
        String s = "abc";
        System.out.println( s.charAt(3) ) ;
    }
}
```
What RuntimeException will the following programs throw, if any?

public class Test {
    
    public static void main (String [] args ) {
        Object o = null;
        System.out.println( o.toString() ) ;
    }

}
What is displayed when the following program is run?

```java
public class Test {
    public static void main(String[] args) {
        try {
            method();
            System.out.println("After the method call");
        }
        catch (ArithmeticException ex) {
            System.out.println("ArithmeticException");
        }
        catch (RuntimeException ex) {
            System.out.println("RuntimeException");
        }
        catch (Exception ex) {
            System.out.println("Exception");
        }
    }

    static void method() throws Exception {
        System.out.println(1 / 0);
    }
}
```
What is displayed when the following program is run?

```java
class Test {
    public static void main(String[] args) {
        try {
            method();
            System.out.println("After the method call");
        }
        catch (ArithmeticException ex) {
            System.out.println("ArithmeticException");
        }
        catch (RuntimeException ex) {
            System.out.println("RuntimeException");
        }
        catch (Exception ex) {
            System.out.println("Exception");
        }
    }
    static void method() throws Exception {
        try {
            String s = "abc";
            System.out.println(s.charAt(3));
        }
        catch (RuntimeException ex) {
            System.out.println("RuntimeException in method() ");
        }
        catch (Exception ex) {
            System.out.println("Exception in method() ");
        }
    }
}
```
What is displayed when the following program is run?

```java
public class Test {
    public static void main(String [] args) {
        try{
            openFile( args[0] );
        }
        catch (FileNotFoundException ex) {
            System.out.println( ex.getMessage() );
        }
        catch (Exception ex) {
            System.out.println( "Something else went wrong." );
        }
    }
    
    static void openFile( String fileName ) throws FileNotFoundException {
        File inFile;
        inFile = new File(fileName);
        if ( !inFile.exists() )
            throw new FileNotFoundException("The file " + fileName
            + " does not exist.");
    }
}
```