Exception Handling

1. Overview

The learning objectives of this laboratory session are:
- Understand the notion of an exception and learn the proper use of exceptions
- Acquire hands-on experience with existing an user-defined interfaces

Exception handling is a mechanism that allows Java programs to handle various exceptional conditions, such as semantic violations of the language and program-defined errors, in a robust way. When an exceptional condition occurs, an exception is thrown. If the Java virtual machine or run-time environment detects a semantic violation, the virtual machine or run-time environment implicitly throws an exception. Alternately, a program can throw an exception explicitly using the throw statement. After an exception is thrown, control is transferred from the current point of execution to an appropriate **`catch`** clause of an enclosing **`try`** statement. The **`catch`** clause is called an exception handler because it handles the exception by taking whatever actions are necessary to recover from it.

2. Handling Exceptions

A **`try`** statement contains a block of code to be executed. Putting a block in a **`try`** statement indicates that any exceptions or other abnormal exits in the block are going to be handled appropriately. A **`try`** statement can have any number of optional **`catch`** clauses that act as exception handlers for the **`try`** block. A **`try`** statement can also have a **`finally`** clause. The **`finally`** block is always executed before control leaves the **`try`** statement; it cleans up after the **`try`** block. Note that a **`try`** statement must have either a **`catch`** clause or a **`finally`** clause, or both.
Here is an example of a **`try`** statement that includes a **`catch`** clause and a **`finally`** clause:

```
try
{
    out.write(b);
}
catch (IOException e)
{
    System.out.println("Output Error");
}
finally
{
    out.close();
}
```

If `out.write()` throws an `IOException`, the exception is caught by the **`catch`** clause. Regardless of whether `out.write()` returns normally or throws an exception, the **`finally`** block is executed, which ensures that `out.close()` is always called.
A **`try`** statement executes the block that follows the keyword **`try`**. If an exception is thrown from within the **`try`** block and the **`try`** statement has any **`catch`** clauses, those clauses are searched, in order, for one that can handle the exception. If a **`catch`** clause handles an exception, that **`catch`** block is executed.
However, if the **`try`** statement does not have any **`catch`** clauses that can handle the exception (or does not have any **`catch`** clauses at all), the exception propagates up through enclosing statements in the current method. If the current method does not contain a **`try`** statement that can handle the exception, the exception propagates up to the invoking method. If this method does not contain an appropriate **`try`** statement, the exception propagates up again, and so on. Finally, if no **`try`** statement is found to handle the exception, the currently running thread terminates.
A **`catch`** clause is declared with a parameter that specifies the type of exception it can handle. The parameter in a **`catch`** clause must be of type **`Throwable`** or one of its subclasses. When an

exception occurs, the **catch** clauses are searched for the first one with a parameter that matches the type of the exception thrown or is a superclass of the thrown exception. When the appropriate **catch** block is executed, the actual exception object is passed as an argument to the **catch** block. The code within a **catch** block should do whatever is necessary to handle the exceptional condition.

The **finally** clause of a **try** statement is always executed, no matter how control leaves the **try** statement. Thus it is a good place to handle clean-up operations, such as closing files, freeing resources, and closing network connections.

3. Declaring Exceptions

If a method is expected to throw any exceptions, the method declaration must declare that fact in a **throws** clause. If a method implementation contains a **throw** statement, it is possible that an exception will be thrown from within the method. In addition, if a method calls another method declared with a **throws** clause, there is the possibility that an exception will be thrown from within the method. If the exception is not caught inside the method with a **try** statement, it will be thrown out of the method to its caller. Any exception that can be thrown out of a method in this way must be listed in a **throws** clause in the method declaration. The classes listed in a **throws** clause must be **Throwable** or any of its subclasses; the **Throwable** class is the superclass of all objects that can be thrown in Java.
However, there are certain types of **Throwable** that do not have to be listed in a **throws** clause. Specifically, if the exception is an instance of Error, RunTimeException, or a subclass of one of those classes, it does not have to be listed in a **throws** clause. Subclasses of the Error class correspond to situations that are not easily predicted, such as the system running out of memory. Subclasses of **RunTimeException** correspond to many common run-time problems, such as illegal casts and array index problems. The reason that these types of exceptions are treated specially is that they can be thrown from such a large number of places that essentially every method would have to declare them.
Consider the following example:

```
import java.io.IOException;
class throwsExample
{
    char[] a;
    int position;
    ...
    // Method explicitly throws an exception
    int read() throws IOException
   {
        if (position >= a.length)
            throw new IOException();
        return a[position++];
    }
    // Method implicitly throws an exception
    String readUpTo(char terminator) throws IOException
    {
        StringBuffer s = new StringBuffer();
        while (true)
       {
            int c = read(); // Can throw IOException
            if (c == -1 || c == terminator)
           {
                return s.toString();
           }
            s.append((char)c);
        }
        return s.toString();
    }
```

```
    // Method catches an exception internally
    int getLength()
   {
        String s;
        try
        {
            s = readUpTo(':');
        }
        catch (IOException e)
        {
            return 0;
        }
        return s.length();
    }
    // Method can throw a RunTimeException
    int getAvgLength()
    {
        int count = 0;
        int total = 0;
        int len;
        while (true)
        {
            len = getLength();
            if (len == 0)
                break;
            count++;
            total += len;
        }
        return total/count; // Can throw ArithmeticException
    }
}
```

The method `read()` can throw an IOException, so it declares that fact in its **`throws`** clause. Without that **`throws`** clause, the compiler would complain that the method must either declare IOException in its **`throws`** clause or **`catch`** it. Although the readUpTo() method does not explicitly throw any exceptions, it calls the read() method that does throw an IOException, so it declares that fact in its **`throws`** clause. Whether explicitly or implicitly thrown, the requirement to catch or declare an exception is the same. The getLength() method catches the IOException thrown by readUpTo(), so it does not have to declare the exception. The last method, getAvgLength(), can throw an ArithmeticException if count is zero. Because ArithmeticException is a subclass of RuntimeException, the fact that it can be thrown out of getAvgLength() does not need to be declared in a **`throws`** clause.

4. Generating Exceptions

A Java program can use the exception-handling mechanism to deal with program-specific errors in a clean manner. A program simply uses the throw statement to signal an exception. The throw statement must be followed by an object that is of type **`Throwable`** or one of its subclasses. For program-defined exceptions, you typically want an exception object to be an instance of a subclass of the **`Exception`** class. In most cases, it makes sense to define a new subclass of Exception that is specific to your program.
Consider the following example:

```
class WrongDayException extends Exception
{
    public WrongDayException () {}
    public WrongDayException(String msg)
   {
        super(msg);
    }
}
```

```
public class ThrowExample
{
    void doIt() throws WrongDayException
   {
        int dayOfWeek =(new java.util.Date()).getDay();
        if (dayOfWeek != 2  && dayOfWeek != 4)
            throw new WrongDayException("Tue. or Thur.");
        // The rest of doIt's logic goes here
        System.out.println("Did it");
    }
    public static void main (String [] argv)
   {
        try
       {
            (new ThrowExample()).doIt();
        }
        catch (WrongDayException e)
       {
            System.out.println("Sorry, can do it only on "
                               + e.getMessage());
        }
    }
}
```

The code in this example defines a class called `WrongDayException` to represent the specific type of exception thrown by the example. The **`Throwable`** class, and most subclasses of **`Throwable`**, have at least two constructors. One constructor takes a string argument that is used as a textual message that explains the exception, while the other constructor takes no arguments. Thus, the `WrongDayException` class defines two constructors.
In the class `ThrowExample`, if the current day of the week is neither Tuesday nor Thursday, the `doIt()` method throws a `WrongDayException`. Note that the `WrongDayException` object is created at the same time it is thrown. It is common practice to provide some information about an exception when it is thrown, so a string argument is used in the allocation statement for the `WrongDayException`. The method declaration for the `doIt()` method contains a **`throws`** clause, to indicate the fact that it can throw a `WrongDayException`.
The `main()` method in `ThrowExample` encloses its call to the `doIt()` method in a **`try`** statement, so that it can catch any `WrongDayException` thrown by `doIt()`. The **`catch`** block prints an error message, using the `getMessage()` method of the exception object. This method retrieves the string that was passed to the constructor when the exception object was created.

4.1. Printing Stack Traces

When an exception is caught, it can be useful to print a stack trace to figure out where the exception came from. A stack trace looks like the following:

```
java.lang.ArithmeticException: / by zero
        at t.cap(t.java:16)
        at t.doit(t.java:8)
        at t.main(t.java:3)
```

You can print a stack trace by calling the **`printStackTrace()`** method that all **`Throwable`** objects inherit from the **`Throwable`** class. For example:

```
int cap (x) {return 100/x}
try
{
    cap(0);
}
catch(ArithmeticException e)
```

```
{
    e.printStackTrace();
}
```

You can also print a stack trace anywhere in an application, without actually throwing an exception. For example:

```
new Throwable().printStackTrace();
```

4.2. Rethrowing Exceptions

After an exception is caught, it can be rethrown if is appropriate. The one choice that you have to make when rethrowing an exception concerns the location from where the stack trace says the object was thrown. You can make the rethrown exception appear to have been thrown from the location of the original exception throw, or from the location of the current rethrow.
To rethrow an exception and have the stack trace indicate the original location, all you have to do is rethrow the exception:

```
try
{
    cap(0);
}
catch(ArithmeticException e)
{
    throw e;
}
```

To arrange for the stack trace to show the actual location from which the exception is being rethrown, you have to call the exception's **`fillInStackTrace()`** method. This method sets the stack trace information in the exception based on the current execution context. Here's an example using the **`fillInStackTrace()`** method:

```
try
{
    cap(0);
}
catch(ArithmeticException e)
{
    throw (ArithmeticException)e.fillInStackTrace();
}
```

It is important to call **`fillInStackTrace()`** on the same line as the **`throw`** statement, so that the line number specified in the stack trace matches the line on which the throw statement appears. The **`fillInStackTrace()`** method returns a reference to the **`Throwable`** class, so you need to cast the reference to the actual type of the exception.

5. Exception guidelines

Use exceptions to:

1. Handle problems at the appropriate level. (Avoid catching exceptions unless you know what to do with them).
2. Fix the problem and call the method that caused the exception again.
3. Patch things up and continue without retrying the method.
4. Calculate some alternative result instead of what the method was supposed to produce.
5. Do whatever you can in the current context and rethrow the same exception to a higher context.

6. Do whatever you can in the current context and throw a different exception to a higher context.
7. Terminate the program.
8. Simplify. (If your exception scheme makes things more complicated, then it is painful and annoying to use.)
9. Make your library and program safer. (This is a short-term investment for debugging, and a long-term investment for application robustness.)

6. New in Java 7

6.1. Multi- catch : Handling Multiple Exceptions in One catch

Quite often, a try block is followed by several catch blocks to handle different types of exceptions. If the bodies of several catch blocks are identical, you can use the new Java SE 7 multi-catch feature to catch those exception types in a single catch handler and perform the same task. The syntax for a multi-catch is:

```
catch ( Type1 | Type2 | Type3 e )
```

Each exception type is separated from the next with a vertical bar (|). The preceding line of code indicates that one of the specified types (or any subclasses of those types) can be caught in the exception handler. Any number of Throwable types can be specified in a multi-catch.

6.2. Try -with-Resources: Automatic Resource Deallocation

The code used to release resources should tyically be placed in a finally block to ensure that a resource is released, regardless of whether there were exceptions when the resource was used in the corresponding try block. An alternative notation—the try -with-resources statement—simplifies writing code in which you obtain one or more resources, use them in a try block and release them in a corresponding finally block. For example, a file-processing application could process a file with a try -with-resources statement to ensure that the file is closed properly when it's no longer needed. Each resource must be an object of a class that implements the **`AutoCloseable`** interface—such a class has a **`close`** method. The general form of a try -with-resources statement is

```
try ( ClassName theObject = new ClassName () )
{
// use theObject here
}
catch ( Exception e )
{
// catch exceptions that occur while using the resource
}
```

where:

`ClassName` is a class that implements the **`AutoCloseable`** interface.

This code creates an object of type ClassName and uses it in the try block, then calls its close method to release any resources used by the object. The try -with-resources statement implicitly calls the theObject 's close method at the end of the try block. You can allocate multiple resources in the parentheses following try by separating them with a semicolon (;).

7. Lab Tasks

6.1. Create a class with a **main( )** that **`throws`** an object of class **Exception** inside a **try** block. Give the constructor for **Exception** a **String** argument. Catch the exception inside a **catch** clause and print the **String** argument. Add a **finally** clause and print a message to prove you were there.

6.2. Create your own exception class using the **extends** keyword. Write a constructor for this class that takes a **String** argument and stores it inside the object with a **String** reference. Write a method that prints out the stored **String**. Create a **try-catch** clause to exercise your new exception.

6.3. Define an object reference and initialize it to **null**. Try to call a method through this reference. Now wrap the code in a **try-catch** clause to catch the exception.

6.4. Create a class with two methods, **f( )** and **g( )**. In **g( )**, throw an exception of a new type that you define. In **f( )**, call **g( )**, catch its exception and, in the **catch** clause, throw a different exception (of a second type that you define). Test your code in **main( )**.

6.5. Create a three-level hierarchy of exceptions. Now create a base-class **A** with a method that **`throws`** an exception at the base of your hierarchy. Inherit **B** from **A** and override the method so it **`throws`** an exception at level two of your hierarchy. Repeat by inheriting class **C** from **B**. In **main( )**, create a **C** and upcast it to **A**, then call the method