Exception Handling

Introduction

While writing programs, it is common for a human being to commit errors. A software engineer may also commit several errors while designing the project or developing the code. These errors are known as 'bugs' and the process of removing them is known as 'debugging'. An error is defined as an abnormal condition or situation in the program, which stops the program from executing normally.

Errors in a Java Program

There are generally three types of errors in a Java program:

1. **Compile-time errors:** These are mostly syntax errors which prevent the program from compiling successfully. When the programmer violates the rules of the language, syntax errors are raised. For example, a programmer might forget to place a semi-colon, or he might misspell the name of a function etc. These are examples for syntax errors which are compile-time errors.
2. **Run-time errors:** The errors which are raised while executing a program are known as run-time errors. For example, dividing a number by zero, accessing an element in the array which not available etc can be treated as examples for run-time errors. **The run-time errors are also known as exceptions.**
3. **Logical errors:** These errors are raised due to mistakes done while writing the logic of the program. For example, writing wrong formulas can be treated as a logical error. These errors are not detected by either the java compiler or the JVM. The programmer himself is responsible for identifying and rectifying the logical errors.

Note: The compile-time errors are detected by the java compiler whereas run-time errors or exceptions are detected by the JVM (Java Virtual Machine).

Exceptions

The run-time errors are known as exceptions. Examples of exceptions are dividing a number by zero, trying to use an index which is not valid in a string etc. The exceptions are detected at run- time by the JVM. Based on how the java compiler treats the exceptions, the exceptions are categorized into two types:

1) Checked exceptions and

2) Unchecked exceptions.

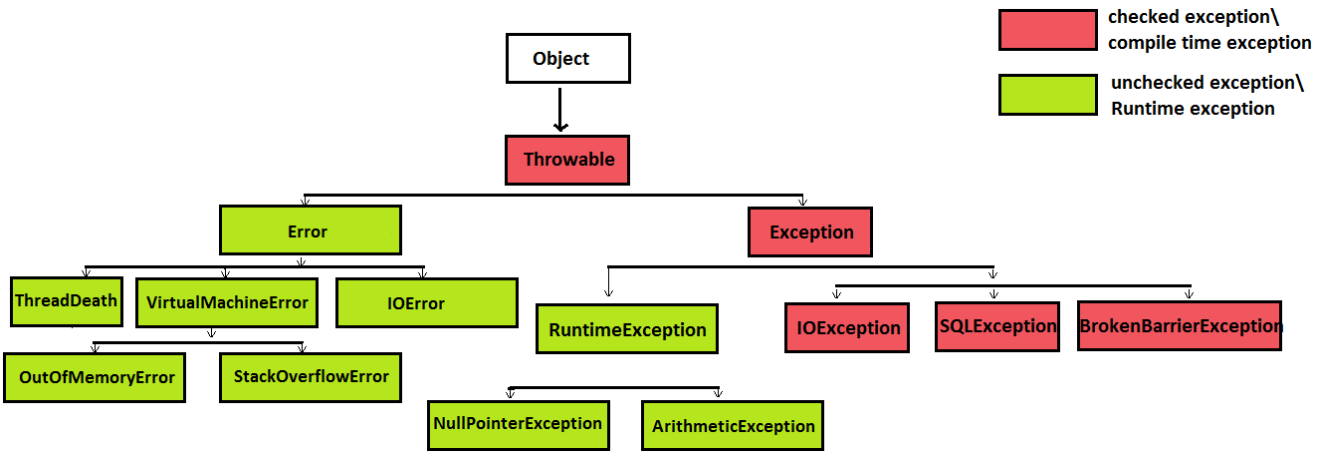
Checked exceptions are those exceptions for which the programmer must provide the exception handling by himself/herself. Unchecked exceptions are those exceptions for which it is not mandatory (compulsory) to provide exception handling code. So, the java compiler checks always whether any exception handling code is provided for checked exception or not. If not provided, then the compiler displays an error.

Examples of unchecked exceptions are: ArithmeticException, NullPointerException etc. Examples of checked exceptions are: IOException, ClassNotFoundException, IllegalAccessException etc.

Exception Hierarchy

For every exception in java, there is a class declared. The root or super class for all the exception classes in java is **Throwable**. The Throwable class has only two sub classes namely Exception and Error. Both these classes are having a number of sub classes. All the exceptions under the Exception class represent the exceptions that might rise due to errors in the code. But all the exceptions under the Error class represent the exceptions that arise due to errors with

respect to operating system or the JVM. The exception hierarchy is as shown in the below diagram:



Exception Handling

Exception is a run-time error. The action performed when an exception rises is known as exception handling or simply handling an exception is known as exception handling. For unchecked exceptions, even though the programmer does not provide any exception handling code, they are handled automatically by the JVM. But the checked exceptions are not handled automatically by the JVM. The programmer has to provide the handling code himself/herself. Java supports exception handling by providing five keywords namely:

1. try
2. catch
3. throw
4. throws
5. finally

The programmer should observe the statements in the program where there may be a possibility of exceptions. Such statements are enclosed in the **try block**. When the exception rises in one of the statements in the try block, the exception handling code must be provided in the **catch block**. After handling the exception, if there are any clean up operations like closing the connection to files or databases or any other resources, such code can be written in the **finally block.** The specialty of the finally block is that the code written in this block will be executed irrespective of whether the exception rises or not.

try block

The statements that are bound to raise an exception are enclosed within the try block. The try block can contain any number of statements. A try block must always follow with atleast one catch block or one finally block. The syntax for try block is as follows:

```
try
{
Statement(s);
}
```

catch block

The programmer provides the exception handling code in the catch block. The JVM detects the exception raised in the try block, loads the exception details on to the stack and returns a reference of the stack to the catch block. The syntax of a catch block is as shown below:

```
catch(ExceptionClass ref)
{
Statement(s);
}
```

The variable **ref** contains the reference to the exception stack on which the details of the exception are stored. We print those details by using the method **printStackTrace()** which available in the **Throwable** class or we can print the reference **ref** using the *print* or *println* methods.

finally block

When the exceptions are not handled by the programmer, sometimes it may lead to unexpected behavior of the system. When the program is using resources like files, database etc, it is a good practice to always close the connections to these resources. Such code is known as cleanup code. Such type of code is written inside the finally block as this block is executed irrespective of whether the exception raises or not. The syntax of finally block is as shown below:

```
finally
{
Statement(s);
}
```

Example:

Let's see a small example on exception handling. Consider the following program:

```
class Sample
{
      public static void main(String[] args)
      {
            int a = 10, b = 0,c;

            c = a/b;

     }

}
```

When we save the above file as "Sample.java" and compile the file by using the command **javac Sample.java** the program compiles successfully. But when we run the program by using the command **java Sample**, the JVM shows the user an error. The error is as shown below:

```
Exception in thread "main"
    java.lang.ArithmeticException: / by zero at
Sample.main(Sample.java:6)
```

In the above output **main** is the name of the thread in which the exception has raised, **java.lang.ArithmeticException** is the name of the exception that has raised, **/ by zero** is the exception message, **Sample.main** is the name of the method, **Sample.java** is the name of the file

and **6** is the line number at which the exception has raised.

In the above example we didn't use try and catch blocks for handling the arithmetic exception. So, how did JVM display the above error message? The above message is displayed because for every unchecked exception java provides a default handler. If the programmer doesn't handle the exception, then JVM automatically invokes the default handler of that exception. Now, let's see the use of try and catch blocks with the help of below example:

```
class Sample
{
public static void main(String[] args)
{
int a = 10, b = 0, c;
try
{
c = a/b;
}

catch(ArithmeticException e)
{
System.out.println("Divide by zero error!");
}
}
}
```

When the above program is executed, the statement **c = a/b** written in the try block raises the exception and a object for the **ArithmeticException** class will be generated and thrown automatically by the JVM. This object will be caught by the catch block and the output that will be displayed is: **Divide by zero error!.**

Multiple catch blocks (Handling multiple exceptions)

It is common that one program might raise multiple exceptions. To deal with multiple

```
class Sample
{
public static void main(String[] args)
{
try
{
                    int a[] = new int[10];
                    a[15] = 32;
String s = "Hello"; System.out.println(s.charAt(10));
}
catch(ArrayIndexOutOfBoundsException e)
{
System.out.println("Invalid array index!");
}
catch(StringIndexOutOfBoundsException e)
{
System.out.println("Invalid string index!");
}
catch(Exception e)
{
System.out.println("Error: "+e);
}
}

}
```

exceptions, we can write multiple catch blocks. When writing multiple catch blocks, the programmer has to follow the hierarchical order from subclass to super class. Let's consider the following example:

In the above program there are three catch blocks. Observe that the catch blocks are written in hierarchical order from subclass to superclass.

Nested try blocks

Java supports nested try blocks i.e. a try block within another try block. Using nested try blocks we can check for different types of exceptions. The syntax of nested try block is as shown below:

```
try
{
try
{
Statement(s);
}
}
```

While writing nested try blocks, every try block is followed by atleast one catch block. When an exception is raised in the inner try block, JVM searches for any matching catch blocks after the inner try block. If none of them matches, then JVM searches for any matching catch blocks after the outer try block. If a matching catch block is found, it will be executed. Otherwise, the default handler for that exception is invoked.

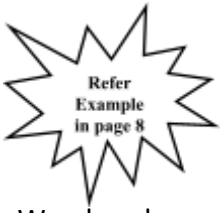
throw clause

The throw clause is used to create and throw the exception object explicitly. Until now, the exception object is created and thrown automatically or implicitly by the JVM. There are two uses of the throw keyword. They are:

1) To throw the exception objects explicitly. This is mainly used by software testers.
2) To create and throw objects of user defined exceptions.

The syntax of the throw clause is as shown below:

```
throw new
  ExceptionName();
```



throws clause

The throws clause is specifically used to throw away the checked exceptions. We already know that it is mandatory for the programmer to provide exception handling code for checked exceptions. In case if you do not wish to provide the exception handling code, you can throw away the exception by using the **throws** clause. The syntax of throws keyword is as shown below:

```
returntype methodname(Parameters...) throws
  ExceptionName
{
Statement(s);
}
```

User-defined Exceptions

Based on the factor how the exceptions are defined, they can be categorized into two types namely predefined exceptions and user defined exceptions. Now, let's see how to define our own

exceptions. The programmers go for user defined exceptions when they do not find any predefined exceptions which solve their problem. There are essentially three steps in creating a user defined exceptions. They are:

1) Name the exception
2) Create a class with the name and extend the Exception class
3) Provide the zero parameter constructor or a single parameter constructor

Let's consider the following code:

```
class MyException extends Exception
{
        String message;
        MyException() { }
        MyException(String
        msg)
{
Message = msg;
}
public void toString()
{
System.out.println(message);
}
}
```

In the above example, the name of the user defined exception is **MyException**. It extends the superclass Exception. We have also provided the toString() method to display the exception message. When using user defined exceptions we must use the **throw** keyword to create and throw the object of the user defined exception class.

Advantages of Exception Handling

1) To separate the exception handling code from the rest of the logic.
2) To propagate the errors up the call stack.
3) Grouping and differentiating error types.

Programs:

```
//Program demonstrating the use of try and
catch blocks class TryCatchDemo
{
        public static void main(String args[])
        {
                try
                {                                       int a,b;
                                                        a = 0; b = 20;
                                                        b = b/a;


                }

                catch(ArithmeticException e)
                {
```

```
                                        System.out.println("Divide by zero error");
                                }
                        }
                }
```

//Program demonstrating multiple catch blocks

```
class MultipleCatch
{
        public static void main(String args[])
        {
                try
                {                                               int a,b;
                                                                a = 0; b = 20;
                                                                b = b/a;
                                                                System.out.println("Statement after exception
                                                                ");
                }

                catch(ArithmeticException e)
                {
                        System.out.println("Divide by zero error");
                }
                catch(Exception e)
                {
                        System.out.println("There is an error. Please check the program again...");
                }
                System.out.println("Statement after all catch blocks     ");
        }
}
```

//Program demonstrating nested try blocks

```
class NestedTry
{
        public static void main(String args[])
        {
                try
                {                       try
                                        {
                                                                        i
                                                                        n
                                        }                               t
                                                                        c
                                                                        [
                                                                        ]
                                                                        =
                                                                        n
                                                                        e
                                                                        w
                                                                        i
```

```
n                    ]                    0
t                    ;                    ]
[                    c                    =
1                    [                    1
0                    2                    ;
                catch(ArithmeticException e)
                {
                        System.out.println("Divide by zero error");
                }
                System.out.println("Statement after inner try and catch block..."); int a,b;
                a = 0; b = 20;
                b = b/a;
        }
        catch(ArithmeticException e)
        {
                System.out.println("Divide by zero error");
        }
        catch(ArrayIndexOutOfBoundsException e)
        {
                System.out.println("Invalid index reference in the array...");
        }
        catch(Exception e)
        {
                System.out.println("There is an error. Please check the program again...");
        }
            System.out.println("Statement after outer try and catch blocks...");

    }
}
```

//Program demonstrating the throw keyword

```
class ThrowDemo
{
    public static void main(String args[])
    {
        try
        {                                   throw new ArithmeticException("Default message
                                            here.  ");
        }

        catch(ArithmeticException e)
        {
                System.out.println(e);
                System.out.println(e.getMessage());
        }
    }
}
```

//Program demonstrating the throws keyword

```
class ThrowsDemo
{
```

```
        public static void main(String args[])
        {
                try
                {                                               display();

                }

                catch(IllegalAccessException e)
                {
                        System.out.println(e);
                }
        }
        static void display() throws IllegalAccessException
        {
                throw new IllegalAccessException("Message...");
        }
}
```

//Program demonstrating the finally block

```
class FinallyDemo
{
        public static void main(String args[])
        {
                try
                {                                               int a,b;
                                                                a = 0; b = 20;
                                                                b = b/a;

                }

                catch(ArithmeticException e)
                {
                        System.out.println("Divide by zero error");
                }
                finally
                {
                        System.out.println("hi");

                }
        }
}
```

Question Bank:

1. With a suitable Java program explain user-defined exception handling
2. What is an exception? How are exceptions handled in Java programming? Explain with suitable program.
3. Write a program that includes a try block and a catch clause which processes the arithmetic exception generated by division-by-zero error.
4. Differentiate between checked and unchecked exceptions.

5. Differentiate between throw and throws with an example.
6. Write a program to illustrate the use of multiple catch blocks for a try block
7. What are the uses of ‘throw’ and ‘throws’ clauses for exception handling?
8. How to handle multiple catch blocks for a nested try block? Explain with an example