Exception Handling in Java – Overview

We don't like exceptions but we always have to deal with them, great news is that Exception handling in Java is very robust and easy to understand and use. Exceptions in java can arise from different kind of situations such as wrong data entered by user, hardware failure, network connection failure, Database server down etc. In this section, we will learn how exceptions are handled in java.

Java being an object-oriented programming language, whenever an error occurs while executing a statement, creates an exception object and then the normal flow of the program halts and JRE tries to find someone that can handle the raised exception. The exception object contains a lot of debugging information such as method hierarchy, line number where the exception occurred, type of exception etc. When the exception occurs in a method, the process of creating the exception object and handing it over to runtime environment is called "throwing the exception".

Once runtime receives the exception object, it tries to find the handler for the exception. Exception Handler is the block of code that can process the exception object. The logic to find the exception handler is simple – starting the search in the method where error occurred, if no appropriate handler found, then move to the caller method and so on. So, if methods call stack is A->B->C and exception is raised in method C, then the search for appropriate handler will move from C->B->A. If appropriate exception handler is found, exception object is passed to the handler to process it. The handler is said to be "catching the exception". If there are no appropriate exception handler found then program terminates printing information about the exception.

Note that Java Exception handling is a framework that is used to handle runtime errors only, compile time errors are not handled by exception handling in java.

We use specific keywords in java program to create an exception handler block, we will look into these keywords next.

Java Exception Handling Keywords

Java provides specific keywords for exception handling purposes we will look after them first and then we will write a simple program showing how to use them for exception handling.

1. throw – We know that if any exception occurs, an exception object is getting created and then Java runtime starts processing to handle them. Sometime we might want to generate exception explicitly in our code, for example in a user authentication program we should throw exception to client if the password is null. throw keyword is used to throw exception to the runtime to handle it.
2. throws – When we are throwing any exception in a method and not handling it, then we need to use throws keyword in method signature to let caller program know the exceptions that might be thrown by the method. The caller method might handle these exceptions or propagate it to it's caller method using

throws keyword. We can provide multiple exceptions in the throws clause and it can be used with main() method also.

3. try-catch – We use try-catch block for exception handling in our code. try is the start of the block and catch is at the end of try block to handle the exceptions. We can have multiple catch blocks with a try and try-catch block can be nested also. catch block requires a parameter that should be of type Exception.
4. finally – finally block is optional and can be used only with try-catch block. Since exception halts the process of execution, we might have some resources open that will not get closed, so we can use finally block. finally block gets executed always, whether exception occurred or not.

Let's see a simple programing showing exception handling in java.

```
package com.journaldev.exceptions;


import java.io.FileNotFoundException;

import java.io.IOException;


public class ExceptionHandling {

	public static void main(String[] args) throws FileNotFoundException,
IOException {
		try{
			testException(-5);
			testException(-10);
		}catch(FileNotFoundException e){
			e.printStackTrace();
		}catch(IOException e){
			e.printStackTrace();
		}finally{
			System.out.println("Releasing resources");
		}
		testException(15);
	}

```

```
	public static void testException(int i) throws FileNotFoundException, IOException{
		if(i < 0){
			FileNotFoundException myException = new FileNotFoundException("Negative Integer "+i);
			throw myException;
		}else if(i > 10){
			throw new IOException("Only supported for index 0 to 10");
		}

	}

}
```

Output of above program is:

```
java.io.FileNotFoundException: Negative Integer -5
	at com.journaldev.exceptions.ExceptionHandling.testException(ExceptionHandling.java:24)
	at com.journaldev.exceptions.ExceptionHandling.main(ExceptionHandling.java:10)
Releasing resources
Exception in thread "main" java.io.IOException: Only supported for index 0 to 10
	at com.journaldev.exceptions.ExceptionHandling.testException(ExceptionHandling.java:27)
	at com.journaldev.exceptions.ExceptionHandling.main(ExceptionHandling.java:19)
```

Notice that testException() method is throwing exception using throw keyword and method signature uses throws keyword to let caller know the type of exceptions it might throw. In main() method, I am handling exception using try-catch block in main() method and when I am not handling it, I am propagating it to runtime with

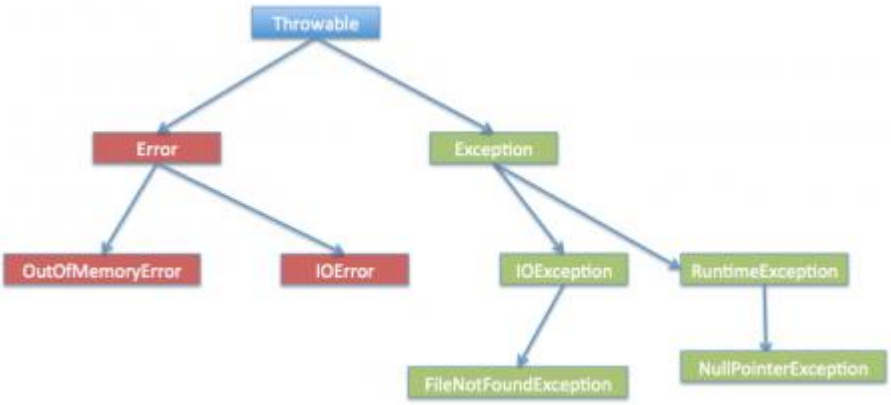
throws clause in main method. Notice that `testException(-10)` never gets executed because of exception and then execution of finally block after try-catch block is executed. The printStackTrace() is one of the useful method in Exception class and used for debugging purpose.

- We can't have catch or finally clause without a try statement.
- A try statement should have either catch block or finally block, it can have both blocks.
- We can't write any code between try-catch-finally block.
- We can have multiple catch blocks with a single try statement.
- try-catch blocks can be nested similar to if-else statements.
- We can have only one finally block with a try-catch statement.

Java Exception Hierarchy

As stated earlier, when any exception is raised an exception object is getting created. Java Exceptions are hierarchical and inheritance is used to categorize different types of exceptions. Throwable is the parent class of Java Exceptions Hierarchy and it has two child objects – Error and Exception. Exceptions are further divided into checked exceptions and runtime exception.

1. Errors: Errors are exceptional scenarios that are out of scope of application and it's not possible to anticipate and recover from them, for example hardware failure, JVM crash or out of memory error. That's why we have a separate hierarchy of errors and we should not try to handle these situations. Some of the common Errors are OutOfMemoryError and StackOverflowError.
2. Checked Exceptions: Checked Exceptions are exceptional scenarios that we can anticipate in a program and try to recover from it, for example FileNotFoundException. We should catch this exception and provide useful message to user and log it properly for debugging purpose. Exception is the parent class of all Checked Exceptions and if we are throwing a checked exception, we must catch it in the same method or we have to propagate it to the caller using throws keyword.
3. Runtime Exception: Runtime Exceptions are cause by bad programming, for example trying to retrieve an element from the Array. We should check the length of array first before trying to retrieve the element otherwise it might throw `ArrayIndexOutOfBoundException` at runtime. RuntimeException is the parent class of all runtime exceptions. If we are throwing any runtime exception in a method, it's not required to specify them in the method signature throws clause. Runtime exceptions can be avoided with better programming.



Exception Handling in Java – Useful Methods

Java Exception and all of it's subclasses doesn't provide any specific methods and all of the methods are defined in the base class Throwable. The exception classes are created to specify different kind of exception scenarios so that we can easily identify the root cause and handle the exception according to it's type. Throwable class implements Serializable interface for interoperability.

Some of the useful methods of Throwable class are;

- public String getMessage() – This method returns the message String of Throwable and the message can be provided while creating the exception through it's constructor.
- public String getLocalizedMessage() – This method is provided so that subclasses can override it to provide locale specific message to the calling program. Throwable class implementation of this method simply use `getMessage()` method to return the exception message.
- public synchronized Throwable getCause() – This method returns the cause of the exception or null id the cause is unknown.
- public String toString() – This method returns the information about Throwable in String format, the returned String contains the name of Throwable class and localized message.
- public void printStackTrace() – This method prints the stack trace information to the standard error stream, this method is overloaded and we can pass PrintStream or PrintWriter as argument to write the stack trace information to the file or stream.

Java 7 Automatic Resource Management and Catch block improvements

If you are catching a lot of exceptions in a single try block, you will notice that catch block code looks very ugly and mostly consists of redundant code to log the error, keeping this in mind Java 7 one of the feature was improved catch block where we can catch multiple exceptions in a single catch block. The catch block with this feature looks like below:

```
catch(IOException | SQLException ex){
    logger.error(ex);
    throw new MyException(ex.getMessage());
}
```

There are some constraints such as the exception object is final and we can't modify it inside the catch block, read full analysis at Java 7 Catch Block Improvements.

Most of the time, we use finally block just to close the resources and sometimes we forget to close them and get runtime exceptions when the resources are exhausted. These exceptions are hard to debug and we might need to look into each place where we are using that type of resource to make sure we are closing it. So java 7 one of the improvement was try-with-resources where we can create a resource in the try statement itself and use it inside the try-catch block. When the execution comes out of try-catch block, runtime environment automatically close these resources. Sample of try-catch block with this improvement is:

```
try (MyResource mr = new MyResource()) {
        System.out.println("MyResource created in try-with-resources");
    } catch (Exception e) {
        e.printStackTrace();
    }
```

Exception Handling in Java – Creating Custom Exception Classes

Java provides a lot of exception classes for us to use but sometimes we may need to create our own custom exception classes to notify the caller about specific type of exception with appropriate message and any custom fields we want to introduce for tracking, such as error codes. For example, let's say we write a method to process only text files, so we can provide caller with appropriate error code when some other type of file is sent as input.

Here is an example of custom exception class and showing it's usage.

```
package com.journaldev.exceptions;

public class MyException extends Exception {
```

```
	private static final long serialVersionUID = 4664456874499611218L;

	private String errorCode="Unknown_Exception";

	public MyException(String message, String errorCode){
		super(message);
		this.errorCode=errorCode;
	}

	public String getErrorCode(){
		return this.errorCode;
	}

}

package com.journaldev.exceptions;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStream;

public class CustomExceptionExample {

	public static void main(String[] args) throws MyException {
		try {
			processFile("file.txt");
		} catch (MyException e) {
			processErrorCodes(e);
```

```
            }

        }

        private static void processErrorCodes(MyException e) throws
MyException {
            switch(e.getErrorCode()){
            case "BAD_FILE_TYPE":
                System.out.println("Bad File Type, notify user");
                throw e;
            case "FILE_NOT_FOUND_EXCEPTION":
                System.out.println("File Not Found, notify user");
                throw e;
            case "FILE_CLOSE_EXCEPTION":
                System.out.println("File Close failed, just log it.");
                break;
            default:
                System.out.println("Unknown exception occured, lets log it
for further debugging."+e.getMessage());
                e.printStackTrace();
            }
        }

        private static void processFile(String file) throws MyException {
            InputStream fis = null;
            try {
                fis = new FileInputStream(file);
            } catch (FileNotFoundException e) {
                throw new
MyException(e.getMessage(),"FILE_NOT_FOUND_EXCEPTION");
```

```
            }finally{
                    try {
                            if(fis !=null)fis.close();
                    } catch (IOException e) {
                            throw new
MyException(e.getMessage(),"FILE_CLOSE_EXCEPTION");
                    }
            }
      }


}
```

Notice that we can have a separate method to process different types of error codes that we get from different methods, some of them gets consumed because we might not want to notify user for that or some of them we will throw back to notify user for the problem.

Here I am extending Exception so that whenever this exception is being produced, it has to be handled in the method or returned to the caller program, if we extends RuntimeException, there is no need to specify it in the throws clause. This is a design decision but I always like checked exceptions because I know what exceptions I can get when calling any method and take appropriate action to handle them.

Exception Handling in Java – Best Practices

Use Specific Exceptions – Base classes of Exception hierarchy doesn't provide any useful information, thats why Java has so many exception classes, such as IOException with further sub-classes as FileNotFoundException, EOFException etc. We should always throw and catch specific exception classes so that caller will know the root cause of exception easily and process them. This makes debugging easy and helps client application to handle exceptions appropriately.

Throw Early or Fail-Fast – We should try to throw exceptions as early as possible. Consider above processFile() method, if we pass null argument to this method we will get following exception.

```
Exception in thread "main" java.lang.NullPointerException
      at java.io.FileInputStream.<init>(FileInputStream.java:134)
      at java.io.FileInputStream.<init>(FileInputStream.java:97)
```

```
        at com.journaldev.exceptions.CustomExceptionExample.processFile(CustomExceptionExample.java:42)
        at com.journaldev.exceptions.CustomExceptionExample.main(CustomExceptionExample.java:12)
```

While debugging we will have to look out at the stack trace carefully to identify the actual location of exception. If we change our implementation logic to check for these exceptions early as below;

```
private static void processFile(String file) throws MyException {
                if(file == null) throw new MyException("File name can't be null", "NULL_FILE_NAME");
//further processing
}
```

Then the exception stack trace will be like below that clearly shows where the exception has occurred with clear message.

```
com.journaldev.exceptions.MyException: File name can't be null
        at com.journaldev.exceptions.CustomExceptionExample.processFile(CustomExceptionExample.java:37)
        at com.journaldev.exceptions.CustomExceptionExample.main(CustomExceptionExample.java:12)
```

Catch Late – Since java enforces to either handle the checked exception or to declare it in method signature, sometimes developers tend to catch the exception and log the error. But this practice is harmful because the caller program doesn't get any notification for the exception. We should catch exception only when we can handle it appropriately. For example, in above method I am throwing exception back to the caller method to handle it. The same method could be used by other applications that might want to process exception in a different manner. While implementing any feature, we should always throw exceptions back to the caller and let them decide how to handle it.

- Closing Resources – Since exceptions halt the processing of program, we should close all the resources in finally block or use Java 7 try-with-resources enhancement to let java runtime close it for you.

- Logging Exceptions – We should always log exception messages and while throwing exception provide clear message so that caller will know easily why the exception occurred. We should always avoid empty catch block that just consumes the exception and doesn't provide any meaningful details of exception for debugging.
- Single catch block for multiple exceptions – Most of the times we log exception details and provide message to the user, in this case we should use java 7 feature for handling multiple exceptions in a single catch block. This approach will reduce our code size and it will look cleaner too.
- Using Custom Exceptions – It's always better to define exception handling strategy at the design time and rather than throwing and catching multiple exceptions, we can create a custom exception with error code and caller program can handle these error codes. Its also a good idea to create a utility method to process different error codes and use it.
- Naming Conventions and Packaging – When you create your custom exception, make sure it ends with Exception so that it will be clear from name itself that it's an exception. Also make sure to package them like it's done in JDK, for example IOException is the base exception for all IO operations.
- Use Exceptions Judiciously – Exceptions are costly and sometimes it's not required to throw exception at all and we can return a boolean variable to the caller program to indicate whether an operation was successful or not. This is helpful where the operation is optional and you don't want your program to get stuck because it fails. For example, while updating the stock quotes in database from a third party webservice, we may want to avoid throwing exception if the connection fails.
- Document the Exceptions Thrown – Use javadoc `@throws` to clearly specify the exceptions thrown by the method, it's very helpful when you are providing an interface to other applications to use.