Exception Handling

Exception Handling

What is an Exception ?

An Exception is an unexpected event that interrupts the normal flow of the program. When an exception occurs program, execution gets terminated. In such cases we get a system generated error message. Ex. divide by zero, array index out of bound etc. Another example of exceptions are, if a requested class is not found, or a requested method is not found. These kinds of exceptions are due to the code in the program; the system is not responsible for these kinds of exceptions.

Exception Handling

What is an Error?

Errors are not exceptions at all, but problems that arise beyond the control of the user or the programmer. error of stack overflow, out of memory error, or a system crash error, this kind of error are due to the system

Exception Handling

Error	Exception
1. Impossible to recover from an error	1. Possible to recover from exceptions
2. Errors are of type 'unchecked'	2. Exceptions can be either 'checked' or 'unchecked'
3. Occur at runtime	3. Can occur at compile time or run time
4. Caused by the application running environment	4. Caused by the application itself

Exception Handling

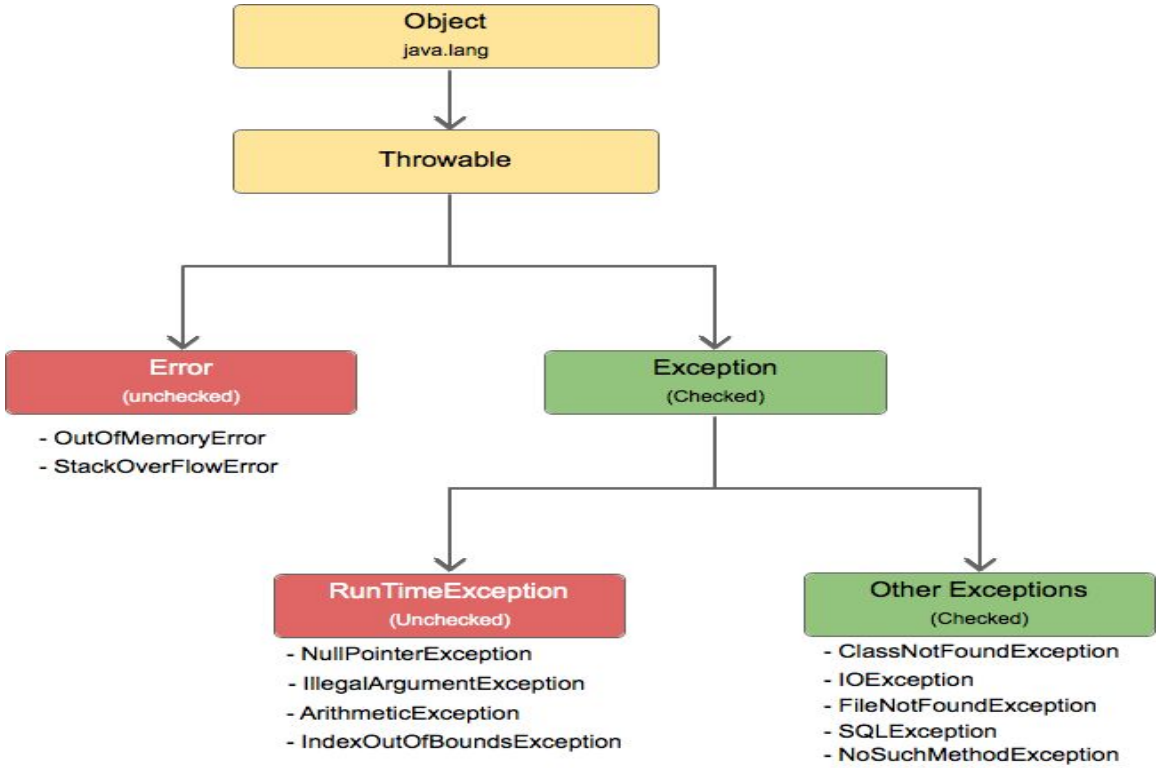
Exception handling is to design the program in such a way that even if there is an exception, all operations are performed then only the program should be terminated is called exception handling.

Following **five keywords** are used to handle the exceptions:

- **try** - Defines a block of code in which exceptions may occur
- **catch** - Catches this exception and handle it in some rational manner
- **throw** - Used to explicitly throw an exception object
- **throws** - Used in the signature of method to indicate that this method might throw one of the listed type exceptions.
- **finally** - A block that is used to execute important code such as closing connection, stream etc. This block is guaranteed to execute whether any exception is generated or not.

Exception Handling

Exception Hierarchy



Exception Handling

try block

- The try is used to define a block of code in which exceptions may occur, it must be used within the method.
- Java try block must be followed by either catch or finally block. No intermediate statements are allowed between try and catch block or between try and finally
- If any exception occurs in try block then control comes out to the try block and executes appropriate catch block and after executing the catch block control never goes back to the try block.

Exception Handling

catch block

- The catch block code catches the exception generated in try block and handles it.
- We can use multiple catch block with a single try.
- Catch block will execute only when exception occurs in try block.
- At a time only one catch block will execute out of multiple catch blocks.

Exception Handling

finally block

- Java finally block is a block that is used to execute important code such as closing connection, stream etc.
- A try block can only have finally block without any catch block. If catch block is also there then catch block will be executed first then finally block will execute.
- **The statements present in the finally block execute even if the try block contains control transfer statements like return, break or continue.**
- finally block will always execute whether an exception occurs or not.
- We can not have a try block without any catch or finally block. I.e. try block must have either catch block only or finally block only or both.

Exception Handling

try-with resources

Generally, when we use any resources like streams, connections, etc. we have to close them explicitly using finally block. try-with-resources, also referred as **automatic resource management,** is a new exception handling mechanism which automatically closes the resources used within the try catch block.

```
try(FileReader fr = new FileReader("file path")) {
      // use the resource
} catch () {
      // body of catch
}
```

Exception Handling

try-with resources

- We can take any number of resources in a try block but it should separated by semicolon.

PrintWriter output.txt") { ….. }

try (FileReader fr = new FileReader("file path") ; PrintWriter printWriter = new catch { ….. }

- The resourcses must be AutoClosable (a resource said to be AutoClosable if and only if the crosspending class implements AutoClosable Interface)
- All resources refrence variable are implicity final. We cannot reassign resource reference value in try block it wiil give compile time error.

Exception Handling

Multiple catch blocks

- try block can have multiple catch block (catching different type of exceptions) as a try block can generate more than one type of exception.
- While using multiple catch statements, it is important to remember that sub classes of class Exception inside catch must come before any of their super classes **otherwise it will lead to compile time error**.

Exception Handling

Multiple catch blocks

```
try{
        int x = 5/0;
}
catch(Exception e) { //This block handles all Exception
        System.out.println("Generic exception");
}
catch(ArithmeticException e) {
        //This catch block is unreachable as class Exception is the parent of the
        class ArithmeticException
        //error: exception ArithmeticException has already been caught
        System.out.println("arithmetic exception : division by zero");
}
```

Exception Handling

Single catch block to handle multiple exceptions

We can use single catch block to handle multiple exceptions. Each exception will be separated by a pipe or vertical bar. It's syntax is:

catch(Exception1 | Exception2 | Exception 3) { ….. }

Note: In such type of single catch statement, we can not take exceptions which are related to each other through inheritance i.e. we can not take child exception classes in the same catch block. Ex. we can not take ArithmeticException (child), if we are taking Exception (parent).

catch (ArithmeticException | Exception e) { …… }

Exception Handling

Single catch block to handle multiple exceptions

error: Alternatives in a multi-catch statement cannot be related by subclassing

```
catch (ArithmeticException | Exception e)
                           ^
```

Alternative ArithmeticException is a subclass of alternative Exception

Exception Handling

‘throws’ keyword

Any method that is capable of causing exceptions must list all the exceptions possible during its execution, so that anyone calling that method gets a prior knowledge about which exceptions are to be handled. A method can do so by using the throws keyword.

Syntax:

```
type method_name(parameter_list) throws exception_list
{
    // definition of method
}
```

Exception Handling

Throw vs Throws

Throw	Throws
Java throw keyword is used to explicitly throw an exception.	Java throws keyword is used to declare an exception.
Checked exception cannot be propagated using throw only.	Checked exception can be propagated with throws.
Throw is followed by an instance.	Throws is followed by class.
Throw is used within the method.	Throws is used with the method signature.
You cannot throw multiple exceptions.	You can declare multiple exceptions e.g. public void method()throws IOException,SQLException.

Exception Handling

```
import java.io.*;
class CheckedUncheckedDemo {
        public static void fun() throws IOException
    {
                System.out.println("fun() throws checked exception IOException.");
    }

    public static void anotherFun() throws ArithmeticException
    {
        System.out.println("anotherFun() throws unchecked exception ArithmeticException");
    }
    public static void main(String args[])
    {
        try{
            fun();
        }
        catch(IOException e)
        {
            System.out.println("Exception caught inside main");
            e.printStackTrace();
        }
        //fun();       //Compilation error: unreported exception IOException; must be caught or declared to be thrown
        anotherFun();   //Works. No compilation error

    }
}
```

‘throws’ Demonstration Program

```
import java.io.*;
class ThrowsDemo  {
        public static void fun() throws IOException,Exception
        {
                System.out.println("fun() throws IOException.");
                throw new IOException("Explicitly generated IOException by the application");
        }
        public static void anotherFun() throws IOException,Exception
        {
                fun();
        }
        public static void main(String args[])
        {
                try
                {
                        anotherFun();
                }
                catch(IOException e)
                {
                        System.out.println("IOException caught inside main");
                        e.printStackTrace();
                }
                catch(Exception e)
                {
                        System.out.println("Exception caught inside main");
                        e.printStackTrace();
                }
        }
}
```

Exception Handling

User Defined Exception : Checked Exception Demonstration

```
class MyException extends Exception
{
	String excpMsg;
	public MyException()
	{
		excpMsg = "This is MyException default message";
		super(excpMsg);
	}

	public MyException(String excpMsg)
	{
		super(excpMsg);
		this.excpMsg = excpMsg;
	}

	//Following is overriding getMessge(). We can uncomment and check it.
	/*public String getMessage()
	{
		return ("Overriden getMessge() : "+this.excpMsg);
	}*/

}
```

Exception Handling

User Defined Exception : Checked Exception Demonstration contd...

```
class UserDefinedExceptionDemo
{

	public static void fun() throws MyException
	{
		throw new MyException("exception is  thrown to test MyException");
	}

	public static void main(String args[])
	{

		try{
			fun();
		}
		catch(MyException me)
		{
			System.out.println(me.getMessage());
		}
		System.out.println("Terminating the application successfully");
	}
}
```

Exception Handling

User Defined Exception : Unchecked Exception Demonstration

```
class MyException extends RuntimeException
{
	String excpMsg;
	public MyException()
	{
		excpMsg = "This is MyException default message";
		super(excpMsg);
	}

	public MyException(String excpMsg)
	{
		super(excpMsg);
		this.excpMsg = excpMsg;
	}

	//Following is overriding getMessge(). We can uncomment and check it.
	/*public String getMessage()
	{
		return ("Overriden getMessge() : "+this.excpMsg);
	}*/

}
```

Exception Handling

User Defined Exception : Unchecked Exception Demonstration contd...

```
class UserDefinedExceptionDemo
{

	public static void fun() throws MyException
	{
		throw new MyException("exception is  thrown to test MyException");
	}

	public static void main(String args[])
	{

		try{
			fun();
		}
		catch(MyException me)
		{
			System.out.println(me.getMessage());
		}
		System.out.println("Terminating the application successfully");
	}
}
```

Exception Handling

Nested Exception

- We can use nested try catch block.
- Exception generated in the inner try block, if not caught by the inner catch block then conrol goes to check with outer catch block to find the match with the generated exception.
-

```
try
{
        try
        {
                //Exception generated here.
        }
        catch( ) // Will be first checked here for match with outer catch block
        { ……. }
}
catch( ) / / Then will be checked here for match with outer catch block
{ ……. }
```

Exception Handling

User defined exception

- We can define our own exceptions.
- These user defined exceptions can be Checked (by extending Exception class) or Unchecked (by extending RuntimeException class).
- Create parameterized constructor taking exception message as String. Inside constructor of our exception class, make a call to super by passing the string exception message.

That is done !!