Chapter 15. Exceptional Handling

1) Introduction :

➔ When an exception condition arises in a method, an object representing that exception will be created and thrown in the method.

➔ That method may chose to handle the exception itself or pass it on. At some point it will be caught and processed.

➔ Exception could be generated by Java run time system or by programmers code.

➔ Exceptions thrown by Java relate to fundamental errors that violate the rules of the Java language or the constraints of the Java execution environment.

➔ Java exception handling is managed via five keywords: **try**, **catch**, **throw**, **throws**, and **finally**.

➔ Program statements that you want to monitor for exceptions are contained within a **try** block. If an exception occurs within the **try** block, it is thrown. Your code can catch this exception (using **catch**) and handle it in some rational manner. System-generated exceptions are automatically thrown by the Java run-time system. To manually throw an exception, use the keyword **throw**. Any exception that is thrown out of a method must be specified as such by a **throws** clause. Any code that absolutely must be executed after a **try** block completes is put in a **finally** block.

➔ All exception types are subclass of built in class **Throwable**. Hence Throwable is at the top of exception class hierarchy. Below throwable there are two subclasses which partition exceptions in two distinct branches. One is **Exception** which is made for exceptional condition which user programs should catch. One has to extend this class to create custom exceptions.

```
public class Exception extends Throwable
```

➔ There is important subclass of Exception which is **RunTimeException**. Exceptions of these types are automatically defined for the programs one write(like ArithmeticException).

➔ Other branch is topped by Error.

```
public class Error extends Throwable
```

➔ Exceptions of type Error are not expected to be caught under normal circumstances by one's program. They are used by Java run time system to indicate errors caused by Java run time environment like stack overflow.

➔ Example :

```
package ExceptionHandling;
public class UncatchedExceptionCls
{
	public static void main(String[] args)
	{
		int d = 0;
		int a = 42 / d;

	}
}
Output:
java.lang.ArithmeticException: / by zero
at ExceptionHandling.UncatchedExceptionCls.main(UncatchedExceptionCls.java:10)
```

➔ In the above example When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then *throws* this exception. This causes the execution of **UncatchedExceptionCls** to stop, because once an exception has been thrown, it must be *caught* by an exception handler and dealt with immediately. In this example, we haven't supplied any exception handlers of our own, so the exception is caught by the default handler provided by the Java

run-time system. (The above output is got after putting code in try/catch pair just for clarity).

➔ The stack trace will always show the sequence of method invocations that led up to the error. In the example below same exception is caught but in a method separate from **main( )**:

```
class Exc1 {
static void subroutine() {
int d = 0;
int a = 10 / d;
}
public static void main(String args[]) {
Exc1.subroutine();
}
}
```

The resulting stack trace from the default exception handler shows how the entire call stack is displayed:

```
java.lang.ArithmeticException: / by zero
at Exc1.subroutine(Exc1.java:4)
at Exc1.main(Exc1.java:7)
```

2) Try and Catch :

➔ Its better to handle Exceptions by programmer himself because of two reasons. One, its allows to fix the error and two, its prevents the programme from automatic termination.

Example:

```
package ExceptionHandling;

    public class TryCatchSimple
    {
      public static void main(String[] args)
      {

      try {
                  int d = 0;
                  int a = 42 / d;
                  System.out.println("this wont be printed");
            }
            catch (ArithmeticException e)
            {
                  System.out.println("Division by zero");
            }
            System.out.println("after catch statement");
      }
}
Output:
Division by zero
after catch statement
```

➔ Throwable class overrides toString() method so that it returns a string containing a description of the exception.

2.1 Multiple Catch clauses :

In some cases, a piece of code can generate more than one type of exceptions. In such cases one can use multiple case statements. When exception occurs catch statements are checked one by one and the one which matches gets executed and rest catch statements will be bypassed.

➔ Example :

```
package ExceptionHandling;

public class MultipleCatchCls
```

```
{
	public static void main(String[] args)
	{
		//String [] str = {"Jayant"};
		//args = str;
		try {
			int a = args.length;
			int div = 40/a;
			int [] intArr = {1};
			intArr[23] =55;
		} catch (ArithmeticException e) {
			System.out.println("divide by zero :"+e);
		}catch(ArrayIndexOutOfBoundsException e)
		{
			System.out.println("Array outofBounds :"+e);
		}
		System.out.println("After try/catch");
	}
}
Output:
divide by zero :java.lang.ArithmeticException: / by zero
After try/catch
```

➔ If we remove the comments (//) in the above program then output will be

```
Array outofBounds :java.lang.ArrayIndexOutOfBoundsException: 23
After try/catch
```

Note: We can give command line arguments in eclipse through 'Right' clicking on .class file → Run As → Run Configuration → Arguments(tab) → Program arguments.

2.2 Nested try statements :

➔ Try statements can be nested. Each time a **try** statement is entered, the context of that exception is pushed on the stack. If an inner **try** statement does not have a **catch** handler for a particular exception, the stack is unwound and the next **try** statement's **catch** handlers are inspected for a match. This continues until one of the **catch** statements succeeds, or until all of the nested **try** statements are exhausted. If no **catch** statement matches, then the Java run-time system will handle the exception.

➔ Example 1:

```
package ExceptionHandling;

public class NestedTryCls
{

	public static void main(String[] args)
	{

			try {
				int a = args.length;

				int b = 23/a;

				try
				{
				if(a==1)
				a = a/(a-a);


				if(a==2)
				{
```

```
                        int [] c = {1};
                        c[49] = 12;
                  }
                  } catch (ArrayIndexOutOfBoundsException e)
                        {
                        System.out.println("Array out of bounds :"+e);
                        }
                  } catch (ArithmeticException e)
                        {
                        System.out.println("Arithmetic excptn :"+e );
                  }
      }
}
```

Output:

```
C:>java NestedTryCls
Arithmetic excptn :java.lang.ArithmeticException: / by zero

C:>java NestedTryCls Hello
Arithmetic excptn :java.lang.ArithmeticException: / by zero

C:>java NestedTryCls Hello World
Array out of bounds :java.lang.ArrayIndexOutOfBoundsException: 49
```

➔ Above we can see when a = 1, the inner try block cannot catch divide by zero exception. Hence it will be passed to outer try block, where it is handled. When a=2 inner try block will catch ArrayIndexOutOfBoundsException.

➔ Example 2 : (How try nesting works between diff classes or methods)

```
package exceptionHandling;

class NestedTryCls12
{
      void method(int i)
      {
            try {
                  System.out.println("inside method()");
                  i = i/(i-i);
                  int [] intArr = {1};
                  intArr[22] = 45;
            } catch (ArrayIndexOutOfBoundsException e)
            {
                  System.out.println("Array IOB : "+e);
            }
      }
}

public class NestedTryCls1
{
      public static void main(String[] args)
      {
            NestedTryCls12 nt12 = new NestedTryCls12();
            try {

                  int a = args.length;
                  int b = 32/a;

                  nt12.method(a);

            } catch (ArithmeticException e) {
                  System.out.println("Arithmetic expt : "+e);
            }
      }
}
```

```
Output:

C:>java NestedTryCls1
Arithmetic expt : java.lang.ArithmeticException: / by zero

C:>java NestedTryCls1 Hello
inside method()
Arithmetic expt : java.lang.ArithmeticException: / by zero

C:>java NestedTryCls1 Hello World
Array IOB : java.lang.ArrayIndexOutOfBoundsException: 22
```

3. 'throw' :

➔ It is possible to throw an Exception explicitly using 'throw' statement as shown below

throw *throwableInstance*;

➔ Here throwableInstance is a object of type Throwable or subclass of Throwable. There are two ways one can get a Throwable object. Either by using a parameter in catch clause or creating one with new operator.

➔ No statement will be executed after throw statement. Control will be passed to the nearest try/catch block. If no matching exception found there then it will go to outer try block and so on. If there is nothing to catch this Exception then Java default exception handler will take control.

➔ Example :

```
package exceptionHandling;

class throwDemo1
{
	void method()
	{
		try {
			throw new NullPointerException("demo");
		} catch (NullPointerException e)
		{
			System.out.println("caught inside method");
			throw e;
		}
	}
}

public class throwDemo {

	public static void main(String[] args)
	{
		try {
			throwDemo1 td1 = new throwDemo1();
			td1.method();
		} catch (NullPointerException e) {
			System.out.println("Recaught "+e);
		}
	}
}
Output:
caught inside method
Recaught java.lang.NullPointerException: demo
```

➔ In the line

```
	new NullPointerException("demo");
```

We can see how Exception creation takes place. Many of Java's built-in exception type have two constructors. One is without String and one is with String. The String ('demo' in this case) is used to describe the exception and get displayed when

Exception object has been sent as argument to print() or println() methods.

4. Throws :

➔ If any method is capable of throwing certain Exception then it should specify this behavior so that caller of that method can guard itself against it. It is done by including **throws** clause in the method's declarations.

➔ throws lists all the exceptions that a method could throw except Error or RunTimeException or any of their subclasses. If it dont then compile time error will occur.

➔ General form of method declaration with throws clause is as shown below

type method-name(*parameter-list*) throws *exception-list*

{
// body of method
}

Here exception-list is comma separated list of Exceptions that method can throw.

➔ Example :

```
package exceptionHandling;

public class ThrowsCls {

	static void throwsMethod()
	{
		//throw new NullPointerException("nullpt");//This was fine
		System.out.println("Inside throwsMethod()");
		throw new IllegalAccessException("throw");//ERROR
	}

	public static void main(String[] args)
	{
		throwsMethod();
	}
}
```

The above code wont compile because of 'Unhandled exception type IllegalAccessException' needs to 'Add throws declaration'. To compile and run it properly one needs to add throws declaration to the method throwsMethod() and main() method must define try/catch block to catch this Exception as shown below.

```
package exceptionHandling;

public class ThrowsCls {

	static void throwsMethod() throws IllegalAccessException
	{
		System.out.println("Inside throwsMethod()");
		throw new IllegalAccessException("throwsDemo");
	}

	public static void main(String[] args)
	{
		try {
			throwsMethod();
		} catch (IllegalAccessException e) {
```

```
                System.out.println("caught "+e);
            }

        }

}
Output:
Inside throwsMethod()
caught java.lang.IllegalAccessException: throwsDemo
```

5. finally :

➔ When Exception occurs program execution takes abrupt non-linear path that alters the normal flow. Let in case if a method opens a file and reading through it and completes immaturely because of Exception and kept the file open. In such cases keyword finally comes handy.

➔ finally block comes after try/catch block and executes regardless of Exception occurred or not. It also executes even before method's return statement.

➔ Each try block should have either one catch block or finally block.

➔ Example 1:

```
package exceptionHandling;

public class FinallyDemoCls {
      static void finMeth1()
      {
            try {
                  System.out.println("Inside finMeth1()");
                  throw new RuntimeException("RunTime");
            } finally
            {
                  System.out.println("finMeth1's finally");
            }
      }

      static void finMeth2()
      {
            try {
                  System.out.println("Inside finMeth2()");
                  return;
            } finally{
                  System.out.println("finMeth2's finally");
            }
      }

      static void finMeth3()
      {
            try {
                  System.out.println("Inside finMeth3()");
                  } finally{
                  System.out.println("finMeth3's finally");
            }
      }

      public static void main(String[] args)
      {
            try {
                  finMeth1();
            } catch (Exception e) {
                  System.out.println("Exception is caught");
            }
            finMeth2();
```

```
            finMeth3();
        }
}
Output:
Inside finMeth1()
finMeth1's finally
Exception is caught
Inside finMeth2()
finMeth2's finally
Inside finMeth3()
finMeth3's finally
```

Note : We can see that try comes with many combination like try-catch, try with multiple catch, nested try-catch and try-finally combo.

➔ A programming language without garbarge collection and without automatic destructor needs finally to release the memory regardless of what is happening in the try block. But java has a garbage collection so releasing memory virtually never a problem then why need finally ? Finally needed for clean up like an open file or network connection. Some may suggest to place code in all the catch statement and not in finally but in that case there will be too much code duplication. Also its not guaranteed that an exception will occur or same exception will occur as provided in different catch statements.

➔ We know try-catch blocks could be nested. Each try-catch block could have their own finally block as shown in the example below.

➔ Example 2

```
package exceptionHandling;
//: exceptions/AlwaysFinally.java
// Finally is always executed.

class FourException extends Exception {}

public class ManyFinally {
  public static void main(String[] args) {
    System.out.println("Entering first try block");
    try {
      System.out.println("Entering second try block");
      try {
        throw new FourException();
      } finally {
        System.out.println("finally in 2nd try block");
      }
    } catch(FourException e) {
      System.out.println(
        "Caught FourException in 1st try block");
    } finally {
      System.out.println("finally in 1st try block");
    }
  }
}
Output:
Entering first try block
Entering second try block
finally in 2nd try block
Caught FourException in 1st try block
finally in 1st try block
```

➔ As we know each try block should be followed by either catch or finally. So if we have method and it has multiple returns and we wanted to do some clean up regardless of return then that method can be sorrounded by try which will be followed by only a finally to complete the clean up.

➔ Example 3

```
package exceptionHandling;
//: exceptions/MultipleReturns.java

public class FinallyAndReturns {
  public static void f(int i) {
    System.out.println("Initialization that requires cleanup");
    try {
      System.out.println("Point 1");
      if(i == 1) return;
      System.out.println("Point 2");
      if(i == 2) return;
      System.out.println("Point 3");
      if(i == 3) return;
      System.out.println("End");
      return;
    } finally {
      System.out.println("Performing cleanup");
    }
  }
  public static void main(String[] args) {
    for(int i = 1; i <= 4; i++)
      f(i);
  }
}
Output:
Initialization that requires cleanup
Point 1
Performing cleanup
Initialization that requires cleanup
Point 1
Point 2
Performing cleanup
Initialization that requires cleanup
Point 1
Point 2
Point 3
Performing cleanup
Initialization that requires cleanup
Point 1
Point 2
Point 3
End
Performing cleanup
```

➔ If finally has code that generate a new exception then previous exception will be completely lost.

➔ Example 4

```
package exceptionHandling;


class VeryImportantException extends Exception {
  public String toString() {
    return "A very important exception!";
  }
}

class HoHumException extends Exception {
  public String toString() {
    return "A trivial exception";
  }
}

public class LostException {
  void f() throws VeryImportantException {
```

```
    throw new VeryImportantException();
  }
  void dispose() throws HoHumException {
    throw new HoHumException();
  }
  public static void main(String[] args) {
    try {
      LostException lm = new LostException();
      try {
        lm.f();
      } finally {
        lm.dispose();
      }
    } catch(Exception e) {
      System.out.println(e);
    }
  }
}
Output:
A trivial exception
```

➔ We know that for unchecked exceptions(RunTimeException and its subclasses) compiler don't ask to handle it but if occurred then stack trace will be printed in runtime by java system. Hence only finally will suffice during compile but catch is a must to run the program. Catch can be avoided completely if finally will have 'return' statement which make exception lost completely. Check the example below.

➔ Example 5

```
package exceptionHandling;

public class OnlyFinally
{
	public static void main(String[] args)
	{
		try
		{
			throw new RuntimeException();
		}
		finally
		{
			System.out.println("inside finally");
			return;
		}
	}
}
Output:
inside finally
```

If in above program we comment out 'return' then output will be

```
inside finally
Exception in thread "main" java.lang.RuntimeException
	at exceptionHandling.OnlyFinally.main(OnlyFinally.java:11)
```

6. Java's Built-in Exceptions:

➔ Java has **java.lang** package which defines several exception classes. Apart from java.lang, exception classes are present in many different libraries like java.util, java.net, java.io etc. Java has ***unchecked*** Exceptions(RuntimeException and its subclasses) because compiler doesn't checks these Exceptions to see if method handles or throws this exceptions. Then there are ***checked*** exceptions, these are checked and enforced at compile time. If any method throws one or more of them then it

should be included in the methods throws list. Java also defines various other Exceptions related to its class and libraries.

➔ Example

```
package exceptionHandling;

class NewException extends Exception{}

public class CheckedUncheckedException {

	public static void main(String[] args)
	{
		throw new NullPointerException();//unchecked
		//throw new NewException();//checked
	}
}
```

Above we can see that for `NullPointerException` which is a unchecked exception, compiler don't give any error. But when we remove the comment and throw NewException, a checked exception then compiler don't give error and one should

Exception	Meaning
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
EnumConstantNotPresentException	An attempt is made to use an undefined enumeration value.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBounds	Attempt to index outside the bounds of a string.
TypeNotPresentException	Type not found.
UnsupportedOperationException	An unsupported operation was encountered.

TABLE 10-1 Java's Unchecked **RuntimeException** Subclasses Defined in **java.lang**

Exception	Meaning
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does not implement the **Cloneable** interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.

TABLE 10-2 Java's Checked Exceptions Defined in **java.lang**

7. Creating your own Exceptions :

➔ Creating your own Exception is very easy, for that one should extend class Exception. Exception class is subclass of Throwable and just inherit all its methods(don't override any). Exception class found in the java package has been shown below.

```
package java.lang;
public class Exception
  extends Throwable
{
  static final long serialVersionUID = -3387516993124229948L;

  public Exception() {}

  public Exception(String paramString)
  {
    super(paramString);
  }

  public Exception(String paramString, Throwable paramThrowable)
  {
    super(paramString, paramThrowable);
  }

  public Exception(Throwable paramThrowable)
  {
    super(paramThrowable);
  }

  protected Exception(String paramString, Throwable paramThrowable, boolean paramBoolean1,
boolean paramBoolean2)
  {
    super(paramString, paramThrowable, paramBoolean1, paramBoolean2);
  }
}
```

Above we can see Exception class don't define any new method neither override any. The methods defined in class Throwable are shown below.

➔ Example :

```
package exceptionHandling;

class CustomException extends Exception
{
```

```
	int a;

	CustomException(int i)
	{
		a = i;
	}

	/*public String toString()
	{
		return "CustomException["+a+"]";
	}*/
}

public class CustomException1
{
	static void compute(int j) throws CustomException
	{
		System.out.println("compute("+j+")");
		if(j>10)
			throw new CustomException(j);
		System.out.println("Normal exit");
	}

	public static void main(String[] args)
	{
		try {
			compute(1);
			compute(20);
		} catch (CustomException e) {
			System.out.println("Caught "+e);
		}
	}
}
Output:
compute(1)
Normal exit
compute(20)
Caught exceptionHandling.CustomException
```

➔ If we comment out toString() method in the above program then output will be

```
Output:
compute(1)
Normal exit
compute(20)
Caught CustomException[20]
```

8. (**)Throwable Class Methods

Modifier and Type	Method and Description
void	**addSuppressed**(**Throwable** exception) Appends the specified exception to the exceptions that were suppressed in order to deliver this exception.
Throwable	**fillInStackTrace**() Fills in the execution stack trace.
Throwable	**getCause**() Returns the cause of this throwable or `null` if the cause is nonexistent or unknown.
String	**getLocalizedMessage**() Creates a localized description of this throwable.

String	**getMessage**() Returns the detail message string of this throwable.
StackTraceElement[]	**getStackTrace**() Provides programmatic access to the stack trace information printed by **printStackTrace()**.
Throwable[]	**getSuppressed**() Returns an array containing all of the exceptions that were suppressed, typically by the try-with-resources statement, in order to deliver this exception.
Throwable	**initCause**(Throwable cause) Initializes the *cause* of this throwable to the specified value.
void	**printStackTrace**() Prints this throwable and its backtrace to the standard error stream.
void	**printStackTrace**(PrintStream s) Prints this throwable and its backtrace to the specified print stream.
void	**printStackTrace**(PrintWriter s) Prints this throwable and its backtrace to the specified print writer.
void	**setStackTrace**(StackTraceElement[] stackTrace) Sets the stack trace elements that will be returned by **getStackTrace()** and printed by **printStackTrace()** and related methods.
String	**toString**() Returns a short description of this throwable.

1. getMessgage(), toString(), printStackTrace(), printStackTrace(PrintStream s)

```
package exceptionHandling;

public class ThrowableMethods {

	public static void main(String[] args)
	{
		try {
			throw new Exception("My Exception");
		} catch (Exception e) {
			System.out.println("getMessage() : "+e.getMessage());
			System.out.println("toString() : "+e.toString());
			System.out.println("printStackTrace() : ");
			e.printStackTrace();
			e.printStackTrace(System.out);
		}
	}
}
Output
getMessage() : My Exception
toString() : java.lang.Exception: My Exception
printStackTrace() :
java.lang.Exception: My Exception
	at exceptionHandling.ThrowableMethods.main(ThrowableMethods.java:8)
java.lang.Exception: My Exception
	at exceptionHandling.ThrowableMethods.main(ThrowableMethods.java:8)
```

2. getStackTrace()

Information provided by printStackTrace() can also be accessed through method getStackTrace(). This method returns array of stack trace elements, each representing one stack frame. Element zero at the top of the stack is the last method invocation in

the sequence and bottom element is first method invocation in the sequence.

```
package exceptionHandling;

public class MethgetStackTrace {

	static void f()
	{
		try
		{
		throw new Exception();
		}
		catch(Exception e)
		{
			for(StackTraceElement ste : e.getStackTrace())
				System.out.println(ste.getMethodName());
		}
	}

	static void g() { f(); }
	static void h() { g(); }

	public static void main(String[] args)
	{
		f();
		System.out.println("----------------------------");
		g();
		System.out.println("----------------------------");
		h();
	}
}
Output
f
main
----------------------------
f
g
main
----------------------------
f
g
h
main
```

➔ If we replace `System.out.println(ste.getMethodName())` with `System.out.println(ste)` then output will be

```
exceptionHandling.MethgetStackTrace.f(MethgetStackTrace.java:9)
exceptionHandling.MethgetStackTrace.main(MethgetStackTrace.java:24)
----------------------------
exceptionHandling.MethgetStackTrace.f(MethgetStackTrace.java:9)
exceptionHandling.MethgetStackTrace.g(MethgetStackTrace.java:19)
exceptionHandling.MethgetStackTrace.main(MethgetStackTrace.java:26)
----------------------------
exceptionHandling.MethgetStackTrace.f(MethgetStackTrace.java:9)
exceptionHandling.MethgetStackTrace.g(MethgetStackTrace.java:19)
exceptionHandling.MethgetStackTrace.h(MethgetStackTrace.java:20)
exceptionHandling.MethgetStackTrace.main(MethgetStackTrace.java:28)
```

➔ '`StackTraceElement`' class is present in java.lang package. In the above example only `getMethod()` of '`StackTraceElement`' class has been used. Other methods are `getClassName()`, `getFileName()`, `getLineNumber()`, `isNativeMethod()`, `hashCode()`, `equals()`, `toString()`.

➔ Also we can note that stack trace through StackTraceElement class keeps on building StackTraceElement array.

3. fillInStackTrace()

➔ When a catched exception is rethrown, stack trace keeps on adding StackTraceElement to the StackTraceElement array.

When we print stack trace then it provides all the information pertaining to exception's origin.

➔ One can make a place, the new origin of that exception by using fillInStackTrace() method wherever it is called.

➔ Example

```
package exceptionHandling;

public class MethodfillInStackTrace {
  public static void f() throws Exception {
    System.out.println("originating the exception in f()");
    throw new Exception("thrown from f()");
  }
  public static void g() throws Exception {
    try {
      f();
    } catch(Exception e) {
      System.out.println("Inside g(),e.printStackTrace()");
      e.printStackTrace(System.out);
      throw e;
    }
  }
  public static void h() throws Exception {
    try {
      f();
    } catch(Exception e) {
      System.out.println("Inside h(),e.printStackTrace()");
      e.printStackTrace(System.out);
      throw (Exception)e.fillInStackTrace();
    }
  }
  public static void main(String[] args) {
    try {
      g();
    } catch(Exception e) {
      System.out.println("main: printStackTrace()");
      e.printStackTrace(System.out);
    }
    try {
      h();
    } catch(Exception e) {
      System.out.println("main: printStackTrace()");
      e.printStackTrace(System.out);
    }
  }
}
Output:
originating the exception in f()
Inside g(),e.printStackTrace()
java.lang.Exception: thrown from f()
      at exceptionHandling.MethodfillInStackTrace.f(MethodfillInStackTrace.java:8)
      at exceptionHandling.MethodfillInStackTrace.g(MethodfillInStackTrace.java:12)
      at exceptionHandling.MethodfillInStackTrace.main(MethodfillInStackTrace.java:30)
main: printStackTrace()
java.lang.Exception: thrown from f()
      at exceptionHandling.MethodfillInStackTrace.f(MethodfillInStackTrace.java:8)
      at exceptionHandling.MethodfillInStackTrace.g(MethodfillInStackTrace.java:12)
      at exceptionHandling.MethodfillInStackTrace.main(MethodfillInStackTrace.java:30)
originating the exception in f()
Inside h(),e.printStackTrace()
java.lang.Exception: thrown from f()
```

```
	at exceptionHandling.MethodfillInStackTrace.f(MethodfillInStackTrace.java:8)
	at exceptionHandling.MethodfillInStackTrace.h(MethodfillInStackTrace.java:21)
	at exceptionHandling.MethodfillInStackTrace.main(MethodfillInStackTrace.java:36)
main: printStackTrace()
java.lang.Exception: thrown from f()
	at exceptionHandling.MethodfillInStackTrace.h(MethodfillInStackTrace.java:25)
	at exceptionHandling.MethodfillInStackTrace.main(MethodfillInStackTrace.java:36)
```

➔ Above we can see for method h(), inside which fillInStackTrace() method is used, its not showing origin of exception as f() as seen for method g(). The origin is shown in method h().

4. Exception Chaining : initCause() method

➔ Often one want to catch the one exception and throw another, but still want to keep the information about the originating exception. This is called exception chaining. Here one should keep in mind that same exception is not thrown again and again in that case JVM automatically provide all info about exception origin in stack trace. But here we are catching one exception and throwing completely new one.

➔ Example

```
package exceptionHandling;

import java.util.ArrayList;

class NewException extends Exception {}

class MethinitCause1
{
	String str = null;
	void method() throws NewException
	{
		try {
			if(str.equals("Jayant"))
			{
				System.out.println("Jayant");
			}
		} catch (Exception e) {
			NewException ne = new NewException();
			//ne.initCause(e);
			throw ne;
		}
	}
}
public class MethinitCause
{
	public static void main(String[] args)
	{
		try {
			new MethinitCause1().method();
		} catch (NewException e) {

			e.printStackTrace();
		}
	}
}
output
exceptionHandling.NewException
	at exceptionHandling.MethinitCause1.method(MethinitCause.java:18)
	at exceptionHandling.MethinitCause.main(MethinitCause.java:29)
```

➔ Above we can see the stacktrace only prints NewException details. If one want to have full chain of exceptions then one

should comment out `ne.initCause(e);` in method method(). After that `e.printStackTrace()` will print the following.

```
exceptionHandling.NewException
      at exceptionHandling.MethinitCause1.method(MethinitCause.java:18)
      at exceptionHandling.MethinitCause.main(MethinitCause.java:29)
Caused by: java.lang.NullPointerException
      at exceptionHandling.MethinitCause1.method(MethinitCause.java:13)
      ... 1 more
```

Above we can see chain of exception with both `NullPointerException` and `NewException` is shown.

9. Chained Exceptions :

➔ Sometimes it happens that an ArithmeticException has occurred due to divide by zero but in reality that error has caused due to I/O error occurred. Hence its better that caller of the method should know that underlying cause is I/O error. This is achieved through chained Exceptions which has been added in Java 1.4.

➔ For chained exceptions two constructors have been added to the Throwable class. They are as shown below.

```
Throwable(Throwable causeExc)
Throwable(String msg, Throwable causeExc)
```

Here *causeExc* is a Exception which has causes the current exception. In the second constructor along with the causing exception one can also give description of the current exception. These two constructors also been added to **Error**, **Exception** and **RuntimeException** classes.

➔ The chained exception methods added to Throwable class are shown below.

```
Throwable getCause( )
Throwable initCause(Throwable causeExc)
```

The getCause() method returns the Exception which underlies the current exception. If there is no underlying exception then null will be returned. The initCause() method associates causeExc with the current exception. The method intiCause() could be only called once for each Exception Object.

➔ Example :

```
package exceptionHandling;

//Demonstrate exception chaining.
class ChainExcDemo
      {
            static void demoproc()
            {
                  //create an exception
                  NullPointerException e = new NullPointerException("top layer");
                  //add a cause
                  e.initCause(new ArithmeticException("cause"));
                  throw e;
      }

      public static void main(String args[])
      {
            try {
                  demoproc();
                  } catch(NullPointerException e) {
                        //display top level exception
                        System.out.println("Caught: " + e);
```

```
                        //display cause exception
                        System.out.println("Original cause: " + e.getCause());
                    }
        }
}
Output:
Caught: java.lang.NullPointerException: top layer
Original cause: java.lang.ArithmeticException: cause
```

10. (**)Restrictions with Exceptions

1. Exceptions of subclass and super class constructors

➔ We know that overidding rule says the overridden method cannot throw new or broader exceptions.

➔ But any method1() calling another method2() which is throwing exception then method1() must sorround it either with try/catch block or include throws clause in own declaration. Also method1() may include more exception in the throws list.

➔ Same is true in case of constructors. If super class constructors throws an exception then subclass constructor should handle the exception by including throws clause. However there are two main differences.

1. When super constructor is throwing an exception then subconstructor can only use throws and cannot use try-catch to sorround super constructor call(compiler error).

➔ Example 1

```
package testingPgms;

class ConstException extends Exception{}

class MethException extends Exception{}

class SupNSubConstExceptn1
{
      SupNSubConstExceptn1() throws ConstException
      {
            System.out.println("SupNSubConstExceptn1");
      }

      public void supMethod() throws MethException
      {
            System.out.println("supMethod()");
      }
}

class SupNSubConstExceptn2 extends SupNSubConstExceptn1
{
      SupNSubConstExceptn2() throws ConstException
      {
            super();
      }

      public void subMethod() throws MethException
      {
            super.supMethod();
      }
}
public class SupNSubConstExceptn
{
      public static void main(String[] args) throws ConstException, MethException
      {
            SupNSubConstExceptn2 sm2 = new SupNSubConstExceptn2();
            sm2.subMethod();
      }
}
```

```
Output:
SupNSubConstExceptn1
supMethod()
```

➔ Now this

2. Overridden methods cannot throw new or broader exceptions but this rule doesn't apply to constructors.

```
package exceptionHandling;


class RightAmountException extends Exception { }
class UndesiredException extends Exception{}
class OtherException extends Exception{}

class Customer
{
	double cash;
	double pdtAmount;

	Customer(double cash, double pdtAmt) throws NullPointerException, RightAmountException
	{
		this.cash = cash;
		this.pdtAmount = pdtAmt;
		if(cash==0)
			throw new RightAmountException();
	}

	public double finalPrice()
	{
		System.out.println("pdtAmount : "+pdtAmount);
		double tax = (pdtAmount*14)/100;
		System.out.println("tax : "+tax);
		double discount = ((pdtAmount+tax)*15)/100;
		System.out.println("discount : "+discount);
		System.out.println("final price : "+(pdtAmount+tax-discount));
		return pdtAmount+tax-discount;
	}

	public boolean avaibility() throws UndesiredException
	{
		if(finalPrice()>cash)
			throw new UndesiredException();
		else
			return true;

	}
}

class CCCustomer extends Customer
{
	String ccno;
	double cash;
	double pdtAmount;

	CCCustomer(String ccno, double cash, double pdtAmt)throws RightAmountException,
OtherException
	{
		super(cash, pdtAmt);
		this.ccno = ccno;
		if(ccno == null && cash==0)
			throw new RightAmountException();
	}

	public boolean cardRequired() throws OtherException
	{
		try {
```

```
                avaibility();
            } catch (UndesiredException e) {
                if(ccno==null)
                {
                    System.out.println("not eligible to buy");
                    return false;
                }
                System.out.println("needed credit card");
                e.printStackTrace();
                return true;
            }

            System.out.println("cash is enough");
            return false;
            }
}

public class ConstructorException
{
    public static void main(String[] args)
    {
                CCCustomer ccc;
                try {
                    ccc = new CCCustomer(null, 105, 110);
                    ccc.cardRequired();
                } catch (RightAmountException | OtherException e) {
                    System.out.println("not eligible to buy");
                    e.printStackTrace();
                }
    }
}
Output1 : ccc = new CCCustomer(null, 105, 110);//no credit card and no enough cash
pdtAmount : 110.0
tax : 15.4
discount : 18.81
final price : 106.59
not eligible to buy
exceptionHandling.UndesiredException
    at exceptionHandling.Customer.avaibility(ConstructorException.java:36)
    at exceptionHandling.CCCustomer.cardRequired(ConstructorException.java:60)
    at exceptionHandling.ConstructorException.main(ConstructorException.java:86)

Output2 : ccc = new CCCustomer(null, 107, 110);//no credit card but enough cash
pdtAmount : 110.0
tax : 15.4
discount : 18.81
final price : 106.59
cash is enough

Output3 : ccc = new CCCustomer("12345678", 105, 110);//less cash with credit card
pdtAmount : 110.0
tax : 15.4
discount : 18.81
final price : 106.59
needed credit card
exceptionHandling.UndesiredException
    at exceptionHandling.Customer.avaibility(ConstructorException.java:36)
    at exceptionHandling.CCCustomer.cardRequired(ConstructorException.java:60)
    at exceptionHandling.ConstructorException.main(ConstructorException.java:85)

Output4 : ccc = new CCCustomer("12345678", 0, 110);//credit card available
not eligible to buy
exceptionHandling.RightAmountException
    at exceptionHandling.Customer.<init>(ConstructorException.java:17)
    at exceptionHandling.CCCustomer.<init>(ConstructorException.java:51)
    at exceptionHandling.ConstructorException.main(ConstructorException.java:85)
```

➔ In the last output we can see that how problematic it could become if constructor is throwing exception based on a condition(cash==0) even when credit card is available with customer. As we cannot catch this exception in base class super constructor.

2. Constructor exception and clean-up

➔ If exception occurs inside a constructor then cleaning up becomes an issue as finally will no more will be an option.

➔ Example 1

```
package exceptionHandling;

import java.io.*;

public class ConstructorNfinally {
  private BufferedReader in;
  public ConstructorNfinally(String fname) throws Exception {
    try {
      in = new BufferedReader(new FileReader(fname));
      // Other code that might throw exceptions
    } catch(FileNotFoundException e) {
      System.out.println("Could not open " + fname);
      // Wasn't open, so don't close it
      throw e;
    } catch(Exception e) {
      // All other exceptions must close it
      try {
        in.close();
      } catch(IOException e2) {
        System.out.println("in.close() unsuccessful");
      }
      throw e; // Rethrow
    } finally {
        dispose();// Don't close it here!!!
    }
  }
  public String getLine() {
    String s;
    try {
      s = in.readLine();
    } catch(IOException e) {
      throw new RuntimeException("readLine() failed");
    }
    return s;
  }
  public void dispose() {
    try {
      in.close();
      System.out.println("dispose() successful");
    } catch(IOException e2) {
      throw new RuntimeException("in.close() failed");
    }
  }

  public static void main(String [] args)
  {
        String fileName = "F:/Java material/java pgm files/ConstructorException.txt";
        try {
            ConstructorNfinally cnf = new ConstructorNfinally(fileName);
            try{
                  String str;
                  while((str = cnf.getLine())!=null)
                  System.out.println(str);
```

```
                    } catch (Exception e) {
                    e.printStackTrace();
                    }
                    finally
                    {
                          // cnf.dispose();
                    }
              }
              catch(Exception e)
              {
                    System.out.println("Some Exception");
              }
  }
}
Output:
dispose() successful
java.lang.RuntimeException: readLine() failed
      at exceptionHandling.ConstructorNfinally.getLine(ConstructorNfinally.java:32)
      at exceptionHandling.ConstructorNfinally.main(ConstructorNfinally.java:52)
```

➔ Above we can see constructor for class `ConstructorNfinally` throws exceptions. But if it dont throw any exception then too finally will be executed and close the file. We want file to be remain open till life time of the object `cnf`. Hence constructor's finally is not the place to close the file.

➔ Now comment out constructor's finally

```
//dispose();// Don't close it here!!!
```

ind use rather finally provided in the main method as shown below

```
finally
      {
            cnf.dispose();
      }
```

The output will be

```
Hello world !!!
How's everyone ?
dispose() successful
```

➔ In the above code inside main method we can see that we have nested `try-catch` block hence if object creation take place successfully then only finally will execute else it wont. Hence the rule is right after creation of an object that requires the clean-up begin a `try-finally` to guarantee clean-up. Example below explains it clearly.

➔ Example 2

```
package exceptionHandling;

class ConstException extends Exception { }

class ConstExceptionHandling1
{
      static int count = 1;
      int id = count++;
      public void dispose()
      {
            System.out.println("cleanup "+id);
      }
}

class ConstExceptionHandling2 extends ConstExceptionHandling1
{
      ConstExceptionHandling2() throws ConstException {}
}
public class ConstExceptionHandling
```

```
{
	public static void main(String[] args)
	{
		ConstExceptionHandling1 ceh11 = new ConstExceptionHandling1();
		try
		{

		}
		finally
		{
			ceh11.dispose();
		}

		ConstExceptionHandling1 ceh12 = new ConstExceptionHandling1();
		ConstExceptionHandling1 ceh13 = new ConstExceptionHandling1();
		try
		{

		}
		finally//if it don't throw exception then two can be disposed
		{
			ceh13.dispose();
			ceh12.dispose();
		}
			try {
				ConstExceptionHandling2 ceh21 = new ConstExceptionHandling2();
				try
				{
					ConstExceptionHandling2 ceh22 = new ConstExceptionHandling2();
					try{
						}
						finally
						{
							ceh22.dispose();
						}
				} catch (ConstException e)
					{
						e.printStackTrace();
					}finally
					{
						ceh21.dispose();
					}
				}
				catch (ConstException e)
				{
					e.printStackTrace();
				}
	}
}
Output:
cleanup 1
cleanup 3
cleanup 2
cleanup 5
cleanup 4
```

11. (**)Exception matching

Multiple Exceptoin Rules

Only three types of exception classes are allowed in multiple catched scenerio.

1. Exception classes that are found above(superclasses) in a inheritance tree of “thrown exception” class should be placed in catched blocks present below the “thrown exception” catch block.

2. Exception classes that are found below(subclasses) in a inheritance tree of “thrown exception” class should be placed in

catched blocks present above the “thrown exception” catch block.

3. Checked exception classes are allowed to put anywhere in the multiple catched blocks applying the same two rules above for their order.

➔ Example

```
package exceptionHandling;

class OneException extends Exception{}
class TwoException extends OneException{}
class ThreeException extends TwoException{}
class FourException extends ThreeException{}
class FiveException extends OneException{}
class SixException extends Exception{}

public class ExceptionMatchSubclasses
{
	public static void main(String[] args)
	{
		try
		{
			throw new TwoException();
		}
		catch(ThreeException e)
		{
			System.out.println("FourException");
		}
		/*catch(FourException e)//Unreachable catch block for FourException. It is
					already handled by the catch block for ThreeException.
		{
			System.out.println("FourException");
		}*/
		catch(TwoException e)
		{
			System.out.println("TwoException");
		}
		catch(FiveException e)//Unreachable catch block for FiveException. This exception
					is never thrown from the try statement body.
		{
			System.out.println("FiveException");
		}
		catch(OneException e)
		{
			System.out.println("OneException");
		}
		catch(SixException e)//Unreachable catch block for SixException. This exception
					is never thrown from the try statement body.
		{
			System.out.println("SixException");
		}
		catch(ArrayIndexOutOfBoundsException e)
		{
			System.out.println("ArrayIndexOutOfBoundsException");
		}
		catch(ArithmeticException e)
		{
			System.out.println("ArithmeticException");
		}
		catch(RuntimeException e)
		{
			System.out.println("RuntimeException");
		}
		catch(Exception e)
		{
			System.out.println("Exception");
```

```
            }
        }
}
```

➔ Above we can see subclass FourException comes after super class ThreeException compiler throws an error "`Unreachable catch block for FourException. It is already handled by the catch block for ThreeException`".

➔ Also we can see there are many catch block for checked exceptions have been included.

➔ If we comment out `catch(FourException e)` then we get two more compiler errors as shown below

a. "`Unreachable catch block for FiveException. This exception is never thrown from the try statement body`".

b. "`Unreachable catch block for SixException. This exception is never thrown from the try statement body`".

Above exception occurs because both `FiveException` and `SixException` classes represent completely different branches.