Exception Handling

An exception is an abnormal condition that arises in a code sequence at run time. In other words, an exception is a runtime error. A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code. In java, **exception is an event that disrupts the normal flow of the program**. When an exceptional condition arises, an object representing that exception is created and thrown in the method that caused the error.

Exception Handling is a mechanism to handle runtime errors. The core advantage of exception handling is **to maintain the normal flow of the application**. Exception normally disrupts the normal flow of the application that is why we use exception handling.

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 runtime 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.

Problem without exception handling/ uncaught exception:

Let's try to understand the problem if we don't use try-catch block.

```
public class Testtrycatch1
{
        public static void main (String args[])
        {
                int data=50/0;      //may throw exception
                System.out.println ("rest of the code...");
        }
}
```

Output:

```
Exception in thread main java.lang.ArithmeticException:/ by zero

        at Testtrycatch1.main(Testtrycatch1.java:5)
```

Exception Types:

- All exception types are subclasses of the built-in class **Throwable**. Thus, **Throwable** is at the top of the exception class hierarchy.
- Immediately below **Throwable** are two subclasses that partition exceptions into two distinct branches:

1. **Exception:** exceptional conditions that user programs should catch

   a. **Checked Exception:**

   - Checked exceptions are checked at compile-time.
   - e.g. IOException, SQLException etc.

   b. **Unchecked Exception (Runtime Exception):**

   - Unchecked exceptions are not checked at compile-time rather they are checked at runtime.
   - e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, etc.

2. **Error:** which defines exceptions that are not expected to be caught under normal circumstances by your program

   - Error is irrecoverable
   - e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.

Advantage of Exception Handling:

to maintain the normal flow of the application.

- Exception normally disrupts the normal flow of the Let's take a scenario:

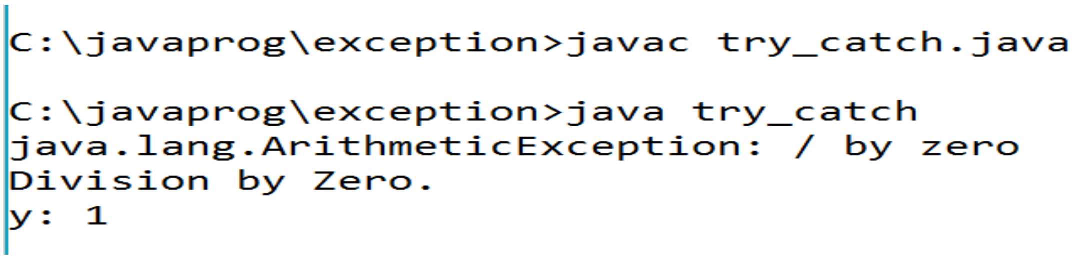
```
statement1;
statement2;
statement3;   //exception occurs
statement4;
statement5;
```

- Suppose there is 5 statements in your program and there occurs an exception at statement3, rest of the code will not be executed i.e. statement4 and statement5 will not run. If we perform exception handling, rest of the statement will be executed. That is why we use exception handling in java.

Using try and catch:

```
class try_catch
{
        public static void main(String [] args)
        {
                int a=10,b=5,c=5;
                int x,y;
                try
                {
                        x= a / (b-c);
                        System.out.println ("This is Not Printed");
                }catch (ArithmeticException e)
                {
                        System.out.println (e);
                        System.out.println ("Division by Zero.");
                }
                y = a / (b+c);
                System.out.println ("y: "+y);
        }
}
```

Output:



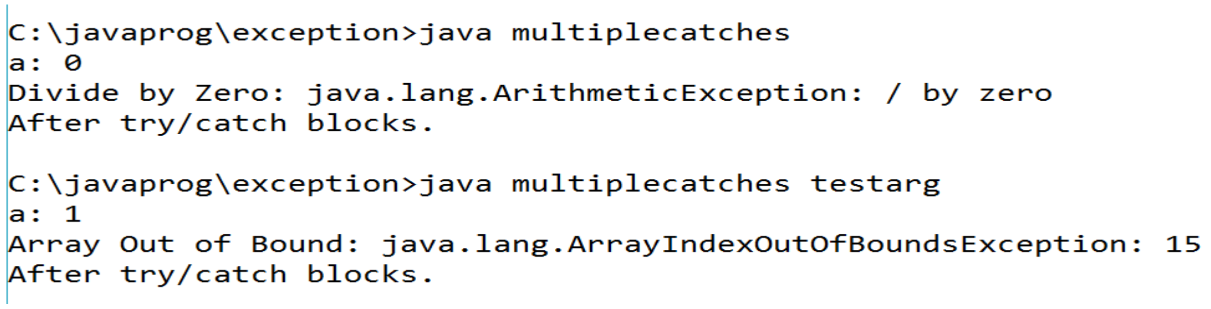
```
C:\javaprog\exception>javac try_catch.java

C:\javaprog\exception>java try_catch
java.lang.ArithmeticException: / by zero
Division by Zero.
y: 1
```

Multiple catch Clauses:

In some cases, more than one exception could be raised by a single piece of code. To handle this type of situation, you can specify two or more catch clauses, each catching a different type of exception. When an exception is thrown, each catch statement is inspected in order, and the first one whose type matches that of the exception is executed. After one catch statement executes, the others are bypassed, and execution continues after the try / catch block. The following example traps two different exception types:

```
class multiplecatches
{
        public static void main(String [] args)
        {
                try
                {
                        int a = args.length;
                        System.out.println("a: "+a);
                        int b = 15/a;                   // Divide by Zero
                        int c[] = {1};
                        c[15]= 99;                      // Array Out of Bound
                }
                catch(ArithmeticException e)
                {
                        System.out.println ("Divide by Zero: "+ e);
                }
                catch (ArrayIndexOutOfBoundsException d)
                {
                        System.out.println("Array Out of Bound: "+ d);
                }
                System.out.println("After try/catch blocks.");
        }
}
```

Output:



```
C:\javaprog\exception>java multiplecatches
a: 0
Divide by Zero: java.lang.ArithmeticException: / by zero
After try/catch blocks.

C:\javaprog\exception>java multiplecatches testarg
a: 1
Array Out of Bound: java.lang.ArrayIndexOutOfBoundsException: 15
After try/catch blocks.
```

Nested try Statements:

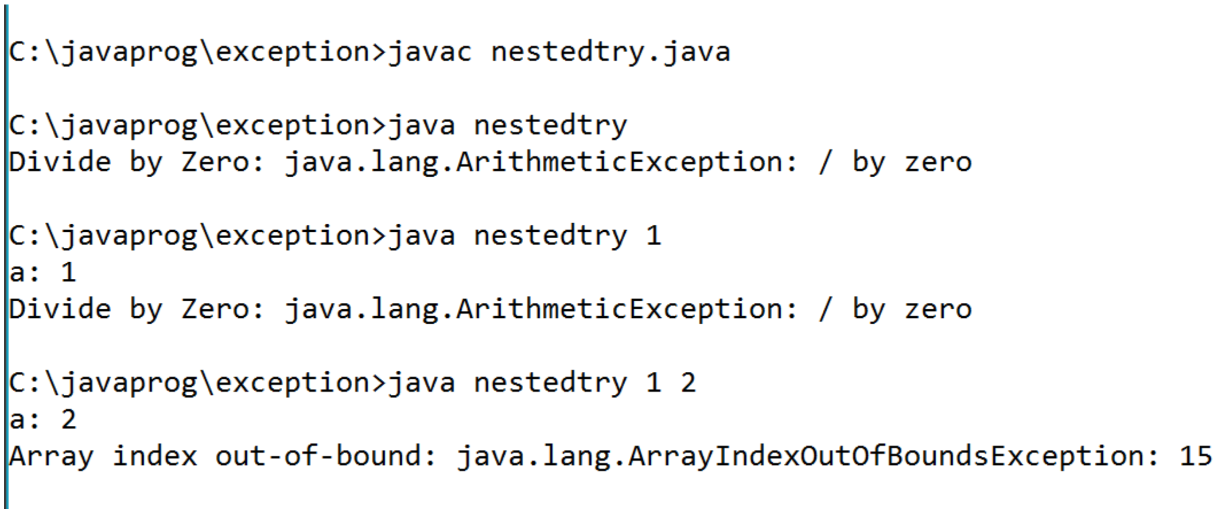
Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested.

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.

```
class nestedtry

{

        public static void main(String [] args)

        {

                try{

                        int a = args.length;

                        int b = 15/a;

                        System.out.println("a: "+a);
```

```
            try{
                if(a== 1)
                    a = a/(a-a);
                if(a == 2)
                {
                    int c[] = {1};
                    c[15]=99;
                }
            } catch(ArrayIndexOutOfBoundsException e)
            {
                System.out.println("Array index out-of-bound: "+ e);
            }
        }catch(ArithmeticException e)
        {
            System.out.println("Divide by Zero: "+ e);
        }
    }
}
```

Output:



```
C:\javaprog\exception>javac nestedtry.java

C:\javaprog\exception>java nestedtry
Divide by Zero: java.lang.ArithmeticException: / by zero

C:\javaprog\exception>java nestedtry 1
a: 1
Divide by Zero: java.lang.ArithmeticException: / by zero

C:\javaprog\exception>java nestedtry 1 2
a: 2
Array index out-of-bound: java.lang.ArrayIndexOutOfBoundsException: 15
```

throw:

The Java throw keyword is used to explicitly throw an exception.

The general form of throw is shown here:

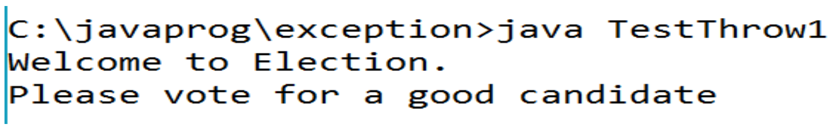
throw *ThrowableInstance*;

Here, *ThrowableInstance* must be an object of type Throwable or a subclass of Throwable.

There are two ways you can obtain a Throwable object: using a parameter in a catch clause or creating one with the new operator.

The flow of execution stops immediately after the throw statement; any subsequent statements are not executed. The nearest enclosing try block is inspected to see if it has a catch statement that matches the type of exception. If it does find a match, control is transferred to that statement. If not, then the next enclosing try statement is inspected, and so on. If no matching catch is found, then the default exception handler halts the program and prints the stack trace.

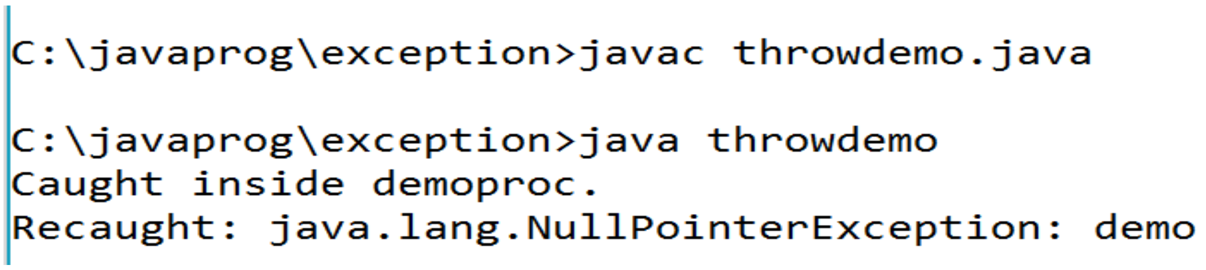
```
public class TestThrow1
{
        static void validate(int age)
        {
                if(age<18)
                        throw new ArithmeticException("Not a valid Voter.");
                else
                        System.out.println("Welcome to Election.");
        }
        public static void main(String args[])
        {
           validate(19);
           System.out.println("Please vote for a good candidate");
        }
}
```



```
C:\javaprog\exception>java TestThrow1
Welcome to Election.
Please vote for a good candidate
```

```
class throwdemo
{
        static void demoproc()
        {
                try{
                        throw new NullPointerException("demo");
                } catch(NullPointerException e)
                {
                        System.out.println("Caught inside demoproc.");
                        throw e;
                }
        }
        public static void main(String [] args)
        {
                try{
                        demoproc();
                } catch(NullPointerException e)
                {
                        System.out.println("Recaught: "+ e);
                }
        }
}
```

Output:



```
C:\javaprog\exception>javac throwdemo.java

C:\javaprog\exception>java throwdemo
Caught inside demoproc.
Recaught: java.lang.NullPointerException: demo
```

```
}
```

throws:

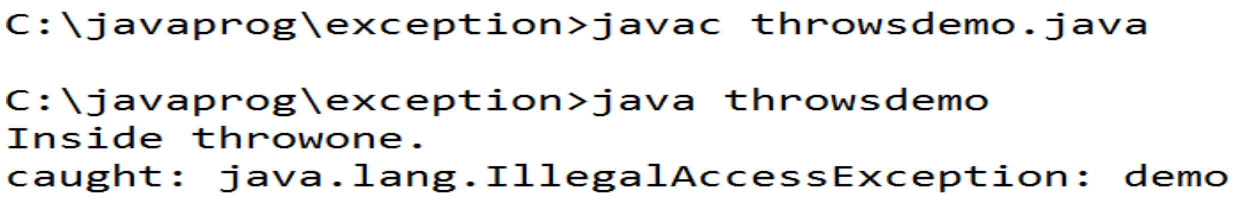
A throws clause lists the types of exceptions that a method might throw. This is necessary for all exceptions, except those of type Error or RuntimeException, or any of their subclasses. All other exceptions that a method can throw must be declared in the throws clause. If they are not, a compile-time error will result. This is the general form of a method declaration that includes a throws clause:

```
type method-name(parameter-list) throws exception-list
{
        // body of method
}
```

It provides information to the caller of the method about the exception.

```
class throwsdemo

{

        static void throwone() throws IllegalAccessException

        {

                System.out.println("Inside throwone.");

                throw new IllegalAccessException("demo");

        }

        public static void main(String [] args)

        {

                try

                {

                        throwone();

                }
                catch(IllegalAccessException e)

                {

                        System.out.println("caught: "+ e);

                }

        }

}
```

Output:

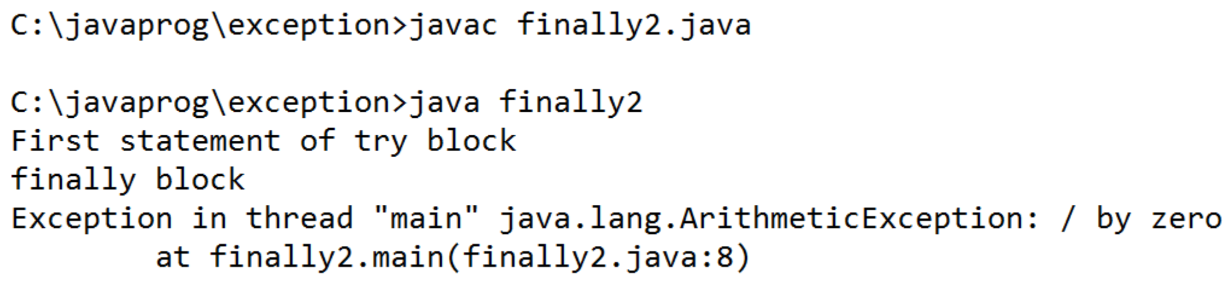


```
C:\javaprog\exception>javac throwsdemo.java

C:\javaprog\exception>java throwsdemo
Inside throwone.
caught: java.lang.IllegalAccessException: demo
```

finally:

finally creates a block of code that will be executed after a try /catch block has completed and before the code following the try/catch block. The finally block will execute whether or not an exception is thrown. If an exception is thrown, the finally block will execute even if no catch statement matches the exception. The finally clause is optional. However, each try statement requires at least one catch or a finally clause.

```
class finally2

{

        public static void main(String args[])

        {

                try

                {
                        System.out.println("First statement of try block");

                        int num=45/0;

                        System.out.println(num);

                }

                catch(ArrayIndexOutOfBoundsException e)

                {

                        System.out.println("ArrayIndexOutOfBoundsException");

                }

                finally

                {

                        System.out.println("finally block");

                }

                System.out.println("Out of try-catch-finally block");

        }

}
```



```
C:\javaprog\exception>javac finally2.java

C:\javaprog\exception>java finally2
First statement of try block
finally block
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at finally2.main(finally2.java:8)
```

Java's Built-in Exceptions:

Unchecked RuntimeException:

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.

Checked Exceptions:

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.
ReflectiveOperationException	Superclass of reflection-related exceptions.

Creating Your Own Exception Subclasses:

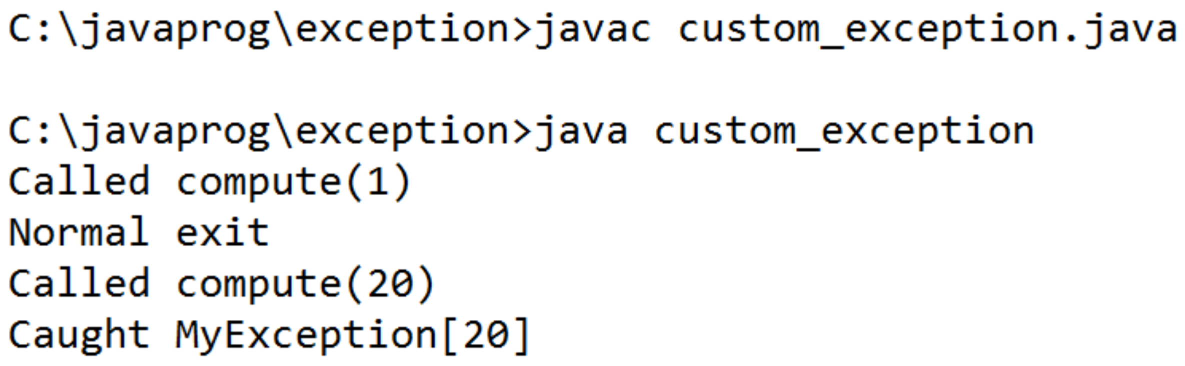
You create your own exception types to handle situations specific to your applications. This is quite easy to do: just define a subclass of Exception.

The Exception class does not define any methods of its own. It does, of course, inherit those methods provided by Throwable. Thus, all exceptions, including those that you create, have the methods defined by Throwable available to them. You may also wish to override one or more of these methods in exception classes that you create.

```
class MyException extends Exception
{
        private int detail;
        MyException(int a)
        {
                detail = a;
        }
        public String toString()
        {
                return "MyException[" + detail + "]";
        }
}
```

```
class custom_exception
{
        static void compute(int a) throws MyException
        {
                System.out.println("Called compute(" + a + ")");
                if (a > 10)
                        throw new MyException(a);
                System.out.println("Normal exit");
        }
        public static void main(String args[])
        {
                try {
                        compute(1);   compute(20);
                 }catch (MyException e)
                {
                        System.out.println("Caught " + e);
                }
        }
}
```

<u>Output:</u>



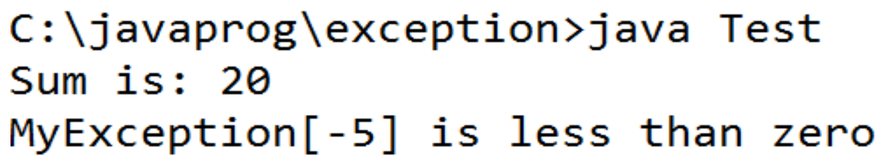
```
C:\javaprog\exception>javac custom_exception.java

C:\javaprog\exception>java custom_exception
Called compute(1)
Normal exit
Called compute(20)
Caught MyException[20]
```

Example 2:

```
class MyException extends Exception
{
        private int number;
        MyException(int a)
        {
                number = a ;
        }
        public String toString()
        {
                return "MyException[" + number +"] is less than zero.";
        }
}

class Test
{
        static void sum(int a,int b) throws MyException
        {
                if(a<0)
                        throw new MyException(a);
                else
                        System.out.println("Sum is: "+(a+b));
        }
```

```
    public static void main(String[] args)
    {
        try
        {
            sum(10, 10);
            sum(-5,8);
        }
        catch(MyException me)
        {
            System.out.println(me);
        }
    }
}
```

Output:



```
C:\javaprog\exception>java Test
Sum is: 20
MyException[-5] is less than zero
```