Exception Handling

An ***exception*** is an abnormal condition that arises in a code sequence at run time. In other words, an exception is a run-time error. In computer languages that do not support exception handling, errors must be checked and handled manually—typically through the use of error codes. This approach is as cumbersome as it is troublesome. Java's exception handling avoids these problems and, in the process, brings run-time error management into the object oriented world.

Exception Handling Fundamentals

A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code. When an exceptional condition arises, an object representing that exception is created and ***thrown*** in the method that caused the error. That method may choose to handle the exception itself, or pass it on. Either way, at some point, the exception is ***caught*** and processed.

Exceptions can be generated by the Java run-time system, or they can be manually generated by your 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. Manually generated exceptions are typically used to report some error condition to the caller of a method.

Java exception handling is managed using five keywords:

- **try**: A suspected code segment is kept inside ***try*** block.
- **catch:** The remedy is written within ***catch*** block.
- **throw:** Whenever run-time error occurs, the code must ***throw*** an exception.
- **throws:** If a method cannot handle any exception by its own and some subsequent methods needs to handle them, then a method can be specified with ***throws*** keyword with its declaration.
- **finally:** block should contain the code to be executed after finishing try-block.

The general form of exception handling is –

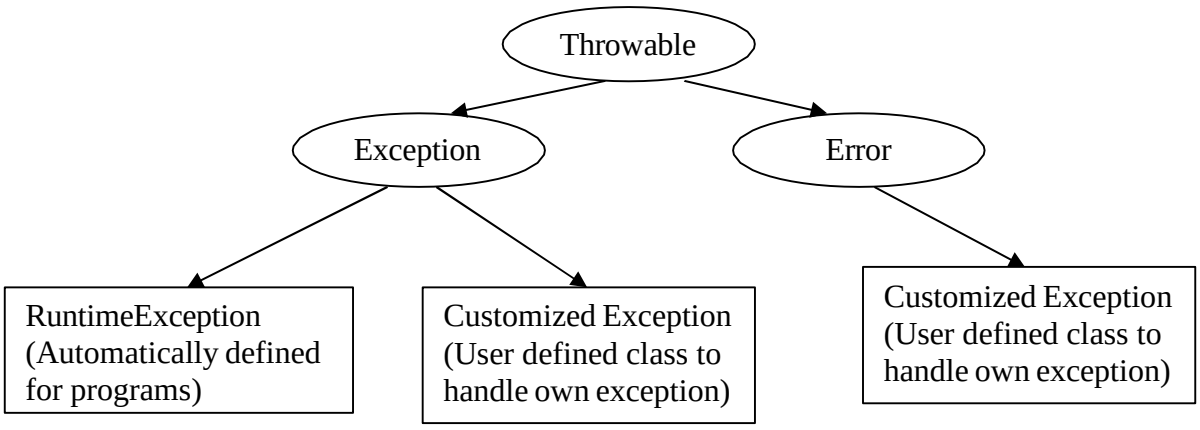
```
try
{
      // block of code to monitor errors
}


catch (ExceptionType1 exOb)
{
      // exception handler for ExceptionType1
}
catch (ExceptionType2 exOb)
{
```

```
        // exception handler for ExceptionType2
    }
    ...
    ….
    finally
    {
        // block of code to be executed after try block ends
    }
```

Exception Types

All the exceptions are the derived classes of built-in class viz. ***Throwable.*** It has two subclasses viz. ***Exception*** and ***Error.***



Exception class is used for exceptional conditions that user programs should catch. We can inherit from this class to create our own custom exception types. There is an important subclass of **Exception**, called **RuntimeException**. Exceptions of this type are automatically defined for the programs that you write and include things such as division by zero and invalid array indexing.

Error class defines exceptions that are not expected to be caught under normal circumstances by our program. Exceptions of type **Error** are used by the Java run-time system to indicate errors having to do with the run-time environment, itself. Stack overflow is an example of such an error.

Uncaught Exceptions

Let us see, what happens if we do not handle exceptions.

```
class Exc0
{
    public static void main(String args[])
    {
        int d = 0;
        int a = 42 / d;
    }
}
```

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 Exc0 to stop, because once an exception has been thrown, it must be *caught* by an exception handler and dealt with immediately. Since,

in the above program, we have not supplied any exception handlers of our own, so the exception is caught by the default handler provided by the Java run-time system.

Any un-caught exception is handled by default handler. The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program. Here is the exception generated when above example is executed:

```
java.lang.ArithmeticException: / by zero
at Exc0.main(Exc0.java:6)
```

The stack trace displays *class name, method name, file name* and *line number* causing the exception. Also, the type of exception thrown viz. *ArithmeticException* which is the subclass of *Exception* is displayed. The type of exception gives more information about what type of error has occurred. The stack trace will always show the sequence of method invocations that led up to the error.

```
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:6)
at Exc1.main(Exc1.java:10)
```

Using try and catch

Handling the exception by our own is very much essential as

- We can display appropriate error message instead of allowing Java run-time to display stack-trace.
- It prevents the program from automatic (or abnormal) termination.

To handle run-time error, we need to enclose the suspected code within *try* block.

```
class Exc2
{
      public static void main(String args[])
      {
            int d, a;

            try
            {
                  d = 0;
```

```
                a = 42 / d;
                System.out.println("This will not be printed.");
            } catch (ArithmeticException e)
            {
                System.out.println("Division by zero.");
            }
            System.out.println("After catch statement.");
        }
    }
```

Output:

Division by zero.
After catch statement.

The goal of most well-constructed catch clauses should be to resolve the exceptional condition and then continue on as if the error had never happened.

```
import java.util.Random;
class HandleError
{
    public static void main(String args[])
    {
        int a=0, b=0, c=0;
        Random r = new Random();

        for(int i=0; i<10; i++)
        {
            try
            {
                b = r.nextInt();
                c =  r.nextInt();
                a = 12345 / (b/c);
            } catch (ArithmeticException e)
            {
                System.out.println("Division by zero.");
                a = 0;
            }
            System.out.println("a: " + a);
        }
    }
}
```

The output of above program is not predictable exactly, as we are generating random numbers. But, the loop will execute 10 times. In each iteration, two random numbers (b and c) will be generated. When their division results in zero, then exception will be caught. Even after exception, loop will continue to execute.

Displaying a Description of an Exception: We can display this description in a println() statement by simply passing the exception as an argument. This is possible because Throwable overrides the toString() method (defined by Object) so that it returns a string containing a description of the exception.

```
catch (ArithmeticException e)
{
      System.out.println("Exception: " + e);
      a = 0;
}
```

Now, whenever exception occurs, the output will be –

```
      Exception: java.lang.ArithmeticException: / by zero
```

Multiple Catch Claues

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.

```
class MultiCatch
{
      public static void main(String args[])
      {
            try
            {
                  int a = args.length;
                  System.out.println("a = " + a);
                  int b = 42 / a;
                  int c[] = { 1 };
                  c[42] = 99;
            } catch(ArithmeticException e)
            {
                  System.out.println("Divide by 0: " + e);
            } catch(ArrayIndexOutOfBoundsException e)
            {
                  System.out.println("Array index oob: " + e);
            }
                  System.out.println("After try/catch blocks.");
      }
}
```

Here is the output generated by running it both ways:

```
C:\>java MultiCatch
a = 0
Divide by 0: java.lang.ArithmeticException: / by zero
After try/catch blocks.

C:\>java MultiCatch TestArg
a = 1
Array index oob: java.lang.ArrayIndexOutOfBoundsException:42
After try/catch blocks.
```

While using multiple *catch* blocks, we should give the exception types in a hierarchy of subclass to superclass. Because, *catch* statement that uses a superclass will catch all exceptions of its own type plus all that of its subclasses. Hence, the subclass exception given after superclass exception is never caught and is a *unreachable code,* that is an *error* in Java.

```
class SuperSubCatch
{
      public static void main(String args[])
      {
            try
            {
                  int a = 0;
                  int b = 42 / a;
            } catch(Exception e)
            {
                  System.out.println("Generic Exception catch.");
            }
            catch(ArithmeticException e)     // ERROR - unreachable
            {
                  System.out.println("This is never reached.");
            }
      }
}
```

The above program generates error “Unreachable Code”, because ArithmeticException is a subclass of Exception.

Nested try Statements

The try statement can be nested. That is, a try statement can be inside the block of another try. 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 the nested try statements are exhausted. If no catch statement matches, then the Java run-time system will handle the exception.

```
class NestTry
{
      public static void main(String args[])
      {
            try
            {
                int a = args.length;
                int b = 42 / a;

                System.out.println("a = " + a);

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

```
                    if(a==2)
                    {
                            int c[] = { 1 };
                            c[10] = 99;
                    }

                   }catch(ArrayIndexOutOfBoundsException e)
                   {
                    System.out.println("Array index out-of-bounds: " + e);
                   }
                 }catch(ArithmeticException e)
                  {
                      System.out.println("Divide by 0: " + e);
                  }
            }
      }
```

When a method is enclosed within a try block, and a method itself contains a try block, it is considered to be a nested try block.

```
class MethNestTry
{     static void nesttry(int a)
      {     try
            {
                  if(a==1)
                        a = a/(a-a);
                  if(a==2)
                  {
                        int c[] = { 1 };
                        c[42] = 99;
                  }
            }catch(ArrayIndexOutOfBoundsException e)
             {
                  System.out.println("Array index out-of-bounds: " + e);
             }
      }

      public static void main(String args[])
      {
            try
            {
                  int a = args.length;
                  int b = 42 / a;
                  System.out.println("a = " + a);
                  nesttry(a);
            } catch(ArithmeticException e)
              {
                  System.out.println("Divide by 0: " + e);
              }
      }
}
```

throw

Till now, we have seen catching the exceptions that are thrown by the Java run-time system. It is possible for your program to throw an exception explicitly, using the ***throw*** statement. The general form of **throw** is shown here:

```
throw ThrowableInstance;
```

Here, *ThrowableInstance* must be an object of type **Throwable** or a subclass of **Throwable**. Primitive types, such as **int** or **char**, as well as non-**Throwable** classes, such as **String** and **Object**, cannot be used as exceptions.

There are two ways you can obtain a **Throwable** object:

- using a parameter in a **catch** clause, or
- creating one with the **new** operator.

```
class ThrowDemo
{
      static void demoproc()
      {
            try
            {
                  throw new NullPointerException("demo");
            } catch(NullPointerException e)
            {
                  System.out.println("Caught inside demoproc: " + e);
                  throw e;

            }
      }

      public static void main(String args[])
      {    try{
            demoproc();
            }
            catch(NullpointerException e)
            {
            System.out.println("Recaught:" + e);
            }
      }
}
```

Here, **new** is used to construct an instance of **NullPointerException**. Many of Java's built-in run-time exceptions have at least two constructors:

- one with no parameter and
- one that takes a string parameter

When the second form is used, the argument specifies a string that describes the exception. This string is displayed when the object is used as an argument to **print()** or **println()**. It can also be obtained by a call to **getMessage()**, which is defined by **Throwable**.

throws

If a method is capable of causing an exception that it does not handle, it must specify this behavior so that

callers of the method can guard themselves against that exception. You do this by including a **throws** clause in the method's declaration. 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.

The general form of a method declaration that includes a **throws** clause:

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

Here, *exception-list* is a comma-separated list of the exceptions that a method can throw.

```
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);
            }
      }
}
```

finally

When exceptions are thrown, execution in a method takes a rather abrupt, nonlinear path that alters the normal flow through the method. Sometimes it is even possible for an exception to cause the method to return prematurely. This could be a problem in some methods. For example, if a method opens a file upon entry and closes it upon exit, then you will not want the code that closes the file to be bypassed by the exception-handling mechanism. The **finally** keyword is designed to address such situations.

The **finally** clause creates a block of code that will be executed after a **try**/**catch** block has completed and before the next code of **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. Any time a method is about to return to the caller from inside a **try/catch** block, via an uncaught exception or an explicit return statement, the **finally** clause is also executed just before the method returns. The **finally** clause is optional. However, each **try** statement requires at least one **catch** or a **finally** clause.

```
class FinallyDemo
{
      static void procA()
      {
            try
            {
                  System.out.println("inside procA");
                  throw new RuntimeException("demo");
```

```
            } finally
              {
                    System.out.println("procA's  finally");
               }
      }


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


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




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

Output:

```
      inside procA
      procA's finally
      Exception caught
```

inside procB
procB's finally
inside procC
procC's finally

Java's Built-in Exceptions

Inside the standard package **java.lang**, Java defines several exception classes. The most general of these exceptions are subclasses of the standard type **RuntimeException**. These exceptions need not be included in any method's **throws** list. Such exceptions are called as ***unchecked exceptions*** because the compiler does not check to see if a method handles or throws these exceptions. Java.lang defines few ***checked exceptions*** which needs to be listed out by a method using ***throws*** list if that method generate one of these exceptions and does not handle it itself. Java defines several other types of exceptions that relate to its various class libraries.

Table: Java's Unchecked Exceptions

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: Java's 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.

Creating your own Exception Subclasses

Although Java's built-in exceptions handle most common errors, sometimes we may want to create our own exception types to handle situations specific to our applications. This is achieved by defining a subclass of ***Exception*** class. Your subclasses don't need to actually implement anything—it is their existence in the type system that allows you to use them as exceptions. The **Exception** class does not define any methods of its own. It inherits those methods provided by **Throwable**. Thus, all exceptions, including those that you create, have the methods defined by **Throwable** available to them.

Method	Description
Throwable fillInStackTrace( )	Returns a Throwable object that contains a completed stack trace. This object can be re-thrown.
Throwable getCause( )	Returns the exception that underlies the current exception. If there is no underlying exception, null is returned.
String getLocalizedMessage( )	Returns a localized description of the exception.
String getMessage()	Returns a description of the exception.
StackTraceElement[] getStackTrace()	Returns an array that contains the stack trace, one element at a time, as an array of StackTraceElement. The method at the top of the stack is the last method called before the exception was thrown. This method is found in the first element of the array. The StackTraceElement class gives your program access to information about each element in the trace, such as its method name.
Throwable initCause(Throwable causeExc)	Associates causeExc with the invoking exception as a cause of the invoking exception. Returns a reference to the exception.
void printStackTrace( )	Displays the stack trace.

void printStackTrace(PrintStream stream)	Sends the stack trace to the specified stream.
void printStackTrace(PrintWriter stream)	Sends the stack trace to the specified stream.
void setStackTrace(StackTraceElement elements[ ])	Sets the stack trace to the elements passed in elements. This method is for specialized applications, not normal use.
String toString( )	Returns a String object containing a description of the exception. This method is called by println( ) when outputting a Throwable object.

We may wish to override one or more of these methods in exception classes that we create. Two of the constructors of **Exception** are:

Exception( )
Exception(String *msg*)

Though specifying a description when an exception is created is often useful, sometimes it is better to override **toString( )**. The version of **toString( )** defined by **Throwable** (and inherited by **Exception**) first displays the name of the exception followed by a colon, which is then followed by your description. By overriding **toString( )**, you can prevent the exception name and colon from being displayed. This makes for a cleaner output, which is desirable in some cases.

```
class MyException extends Exception
{
      int marks;

      MyException (int m)
      {
         marks=m;
      }

      public String toString()
      {
         return "MyException: Marks cannot be Negative";
      }
}
class CustExceptionDemo
{
      static void test(int m) throws MyException
      {
         System.out.println("Called test(): "+m);
         if(m<0)
            throw new MyException(m);

          System.out.println("Normal exit");
      }
```

```
        public static void main(String args[])
        {
           try{
              test(45);
              test(-2);
           }
           catch (MyException e)
           {
              System.out.println("Caught " + e);
            }
        }
}
```

Chained Exceptions

The concept of *chained exception* allows you to associate another exception with an exception. This second exception describes the cause of the first exception. For example, imagine a situation in which a method throws an ArithmeticException because of an attempt to divide by zero. However, the actual cause of the problem was that an I/O error occurred, which caused the divisor to be set improperly. Although the method must certainly throw an ArithmeticException, since that is the error that occurred, you might also want to let the calling code know that the underlying cause was an I/O error. Chained exceptions let you handle this, and any other situation in which layers of exceptions exist.

To allow chained exceptions, two constructors and two methods were added to **Throwable**. The constructors are shown here:

- Throwable(Throwable *causeExc*)
- Throwable(String *msg*, Throwable *causeExc*)

In the first form, *causeExc* is the exception that causes the current exception. That is, *causeExc* is the underlying reason that an exception occurred. The second form allows you to specify a description at the same time that you specify a cause exception. These two constructors have also been added to the **Error**, **Exception**, and **RuntimeException** classes.

Chained exceptions can be carried on to whatever depth is necessary. Thus, the cause exception can, itself, have a cause. Be aware that overly long chains of exceptions may indicate poor design. Chained exceptions are not something that every program will need. However, in cases in which knowledge of an underlying cause is useful, they offer an elegant solution.

Using Exceptions

Exception handling provides a powerful mechanism for controlling complex programs that have many dynamic run-time characteristics. It is important to think of try, throw, and catch as clean ways to handle errors and unusual boundary conditions in your program's logic. Unlike some other languages in which error return codes are used to indicate failure, Java uses exceptions. Thus, when a method can fail, have it throw an exception. This is a cleaner way to handle failure modes.

Note that Java's exception-handling statements should not be considered a general mechanism for nonlocal branching. If you do so, it will only confuse your code and make it hard to maintain.