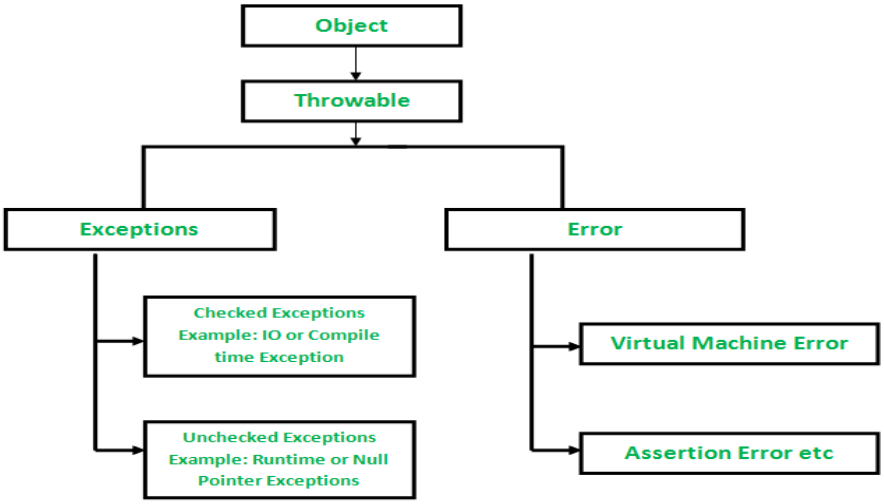
Exception Handling in Java

The **Exception Handling in Java** is one of the powerful mechanism to handle the runtime errors so that normal flow of the application can be maintained.

Advantage of Exception Handling :The core advantage of exception handling is **to maintain the normal flow of the application**. An exception normally disrupts the normal flow of the application that is why we use exception handling.

Hierarchy of Java Exception classes



Difference between Checked and Unchecked Exceptions

1) Checked Exception

The classes which directly inherit Throwable class except RuntimeException and Error are known as checked exceptions e.g. IOException, SQLException etc. Checked exceptions are checked at compile-time.

2) Unchecked Exception

The classes which inherit RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

3) Error

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

Java Exception Keywords

There are 5 keywords which are used in handling exceptions in Java.

Keyword	Description
try	The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. It means, we can't use try block alone.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the important code of the program. It is executed whether an exception is handled or not.
throw	The "throw" keyword is used to throw an exception.
throws	The "throws" keyword is used to declare exceptions. It doesn't throw an exception. It specifies that there may occur an exception in the method. It is always used with method signature.

Example to demonstrate try and catch statements

```
public class JavaExceptionExample

{
  public static void main(String args[])
  {
        try
        {
              //code that may raise exception
             int data=100/0;
        }
       catch(ArithmeticException e)
        {
            System.out.println(e);}
              //rest code of the program
           System.out.println("rest of the code...");
```

```
        }
}
```

Common Scenarios of Java Exceptions

There are given some scenarios where unchecked exceptions may occur. They are as follows:

1) A scenario where ArithmeticException occurs

If we divide any number by zero, there occurs an ArithmeticException.

```
int a=50/0; //ArithmeticException
```

---

2) A scenario where NullPointerException occurs

If we have a null value in any variable, performing any operation on the variable throws a NullPointerException.

```
String s=null;
System.out.println(s.length()); //NullPointerException
```

---

3) A scenario where NumberFormatException occurs

The wrong formatting of any value may occur NumberFormatException. Suppose I have a string variable that has characters, converting this variable into digit will occur NumberFormatException.

```
String s="abc";
int i=Integer.parseInt(s); //NumberFormatException
```

---

4) A scenario where ArrayIndexOutOfBoundsException occurs

If you are inserting any value in the wrong index, it would result in ArrayIndexOutOfBoundsException as shown below:

```
int a[]=new int[5];
a[10]=50; //ArrayIndexOutOfBoundsException
```

Java Multi-catch block

A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block.

Points to remember

- At a time only one exception occurs and at a time only one catch block is executed.
- All catch blocks must be ordered from most specific to most general, i.e. catch for ArithmeticException must come before catch for Exception.

Example 1

Let's see a simple example of java multi-catch block.

```
public class MultipleCatchBlock1
 {
   public static void main(String[] args)
   {

        try
       {
            int a[]=new int[5];
            a[5]=30/0;

       }
         catch(ArithmeticException e)
              {
             System.out.println("Arithmetic Exception occurs");
               }
        catch(ArrayIndexOutOfBoundsException e)
              {
               System.out.println("ArrayIndexOutOfBounds Exception occurs");
              }
        catch(Exception e)
            {
             System.out.println("Parent Exception occurs");
            }
           System.out.println("rest of the code");
   }
}
```

Nested try statements

- 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 of the nested try statements are exhausted
- If no catch statement matches, then the Java run-time system will handle the exception

Example

```
public class Nestedtry {

	public static void main(String[] args) {

		try
		{
			try
			{
				System.out.println("going to divide");
				int b =39/0;
			 }
			catch(ArithmeticException e)
			{
				System.out.println(e);
			}

			try {
				int a[]=new int[5];
				a[5]=4;
			}

			catch(ArrayIndexOutOfBoundsException e)
			{
				System.out.println(e);
			}

		   System.out.println("other statement");
	  }

	  catch(Exception e)

	  {
		System.out.println("handeled");
	  }

		System.out.println("normal flow..");
	}

}
```

Throw keyword

The throw keyword is used to throw an exception explicitly. Only object of Throwable class or its sub classes can be thrown. Program execution stops on encountering **throw** statement, and the closest catch statement is checked for matching type of exception.

```
class Test
{
 static void avg()
  {
    try
    {
      throw new ArithmeticException("demo");
    }
    catch(ArithmeticException e)
    {
      System.out.println("Exception caught");
    }
 }

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

Throws keyword

- 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
- A throws clause lists the types of exceptions that a method might throw

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

}

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

```
class Test
{
      static void check() throws ArithmeticException
    {
       System.out.println("Inside check function");
            throw new ArithmeticException("demo");
    }
    public static void main(String args[])
   {
        try
       {
        check();
       }
       catch(ArithmeticException e)
      {
           System.out.println("caught" + e);
      }
   }
}
```

Finally block

A finally block contains all the crucial statements that must be executed whether exception occurs or not. The statements present in this block will always execute regardless of whether exception occurs in try block or not such as closing a connection, stream etc.

Application of finally block: So basically the use of finally block is **resource deallocation**. Means all the resources such as Network Connections, Database Connections, which we opened in try block are needed to be closed so that we won't lose our resources as opened. So those resources are needed to be closed in finally block.

Example program to demonstrate the **three methods** that exit in various ways, none without executing their **finally** clauses.

In the below example **p1( )** method prematurely breaks out of the try by throwing an exception. The finally clause is executed on the way out. In method **p2( )** try statement is exited via a return statement. The finally clause is executed before method p2( ) returns. In method **p3( )** try statement executes normally, without error.

```
public class Idemo

{
        static void p1()
        {
                try
                {
                     System.out.println("inside method  p1");
                      throw new RuntimeException("demo");
                }

                finally
                {
                    System.out.println("method p1 finally");
                }
        }
        static void p2() // Return from within a try block.
       {
                try
                {
                    System.out.println("inside method p2");
                    return;
                }
                finally
                {
                    System.out.println("p2 finally");
                }
        }

        static void p3()
        {
                        try
                        {
                            System.out.println("inside method p3");
                        }
                        finally
                        {
                            System.out.println("p3 finally");
                        }
        }

        public static void main(String[] args)
        {
                try
                {
                      p1();

                }
                catch (Exception e)
                {
                        System.out.println("Exception caught");
                 }
          p2();
          p3();

    } }
```

Custom Exception in java

If you are creating your own Exception that is known as custom exception or user-defined exception. Java custom exceptions are used to customize the exception according to user need. **Create a new class whose name should end with Exception like ClassNameException**. This is a convention to differentiate an exception class from regular ones. **Make the class extends one of the exceptions which are subtypes of the java.lang.Exception** class. Generally, a custom exception class always extends directly from the Exception class. **Create a constructor with a String parameter** which is the detail message of the exception. In this constructor, simply call the super constructor and pass the message.

Example to demonstrate custom exception or user defined exception

```
class InvalidAgeException extends Exception
{
    InvalidAgeException(String s)
    {
       super(s);
    }
}

class TestCustomException1
{
         static void validate(int age) throws InvalidAgeException
        {
           if(age<18)
              throw new InvalidAgeException("not valid");
          else
            System.out.println("welcome to vote");
        }
          public static void main(String args[])
         {
             try
            {
            validate(13);
            }
            catch(Exception m)
          {
             System.out.println("Exception occured: "+m);
          }
         System.out.println("rest of the code...");
   }
}
```