Exceptions

An exception **is an unexpected event**, which may occur during the execution of a program (at run time), to disrupt the normal flow of the program's instructions. This leads to the abnormal termination of the program.

Therefore, these exceptions are needed to be handled. The exception handling in java is one of the powerful mechanisms to handle the runtime errors so that normal flow of the application can be maintained.

An exception may occur due to the following reasons. They are.

- Invalid data as input.
- Network connection may be disturbed in the middle of communications
- JVM may run out of memory.
- File cannot be found/opened.

These exceptions are caused by user error, programmer error, and physical resources. Based on these, the exceptions can be classified into three categories.

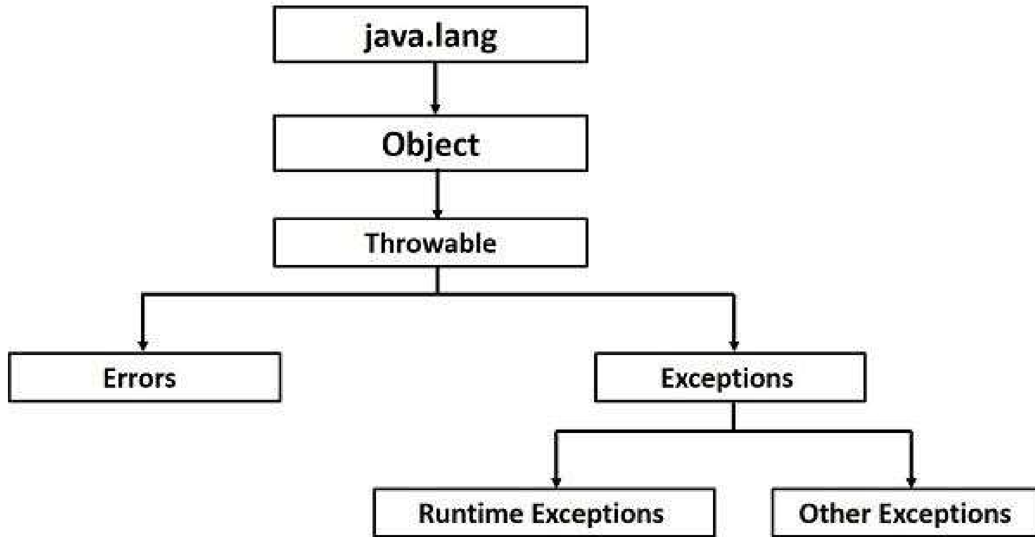
- ***Checked exceptions*** − A checked exception is an exception that occurs at the compile time, also called as compile time exceptions. These exceptions cannot be ignored at the time of compilation. So, the programmer should handle these exceptions.
- ***Unchecked exceptions*** − An unchecked exception is an exception that occurs at run time, also called as **Runtime Exceptions**. These include programming bugs, such as logic errors or improper use of an API. Runtime exceptions are ignored at the time of compilation.
- ***Errors*** − Errors are not exceptions, but problems may arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything about an error. For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.
- ***Error:*** An Error indicates serious problem that a reasonable application should not try to catch.
- ***Exception:*** Exception indicates conditions that a reasonable application might try to catch.

Exception Hierarchy

The java.lang.Exception class is the base class for all exception classes. All exception and errors types are sub classes of class Throwable, which is base class of hierarchy. One branch is headed by Exception. This class is used for exceptional conditions that user programs should catch. NullPointerException is an example of such an exception. Another branch, Er- ror are used by the Java run-time system(JVM) to indicate errors having to do with the run- time environment itself(JRE). StackOverflowError is an example of such an error.

Errors are abnormal conditions that happen in case of severe failures, these are not han- dled by the Java programs. Errors are generated to indicate errors generated by the runtime environment. Example: JVM is out of memory. Normally, programs cannot recover from errors.

The Exception class has two main subclasses: IOException class and RuntimeException Class.



Exceptions Methods

Method	Description
public String getMessage()	Returns a detailed message about the exception that has occurred. This message is initialized in the Throwable constructor.
public Throwable getCause()	Returns the cause of the exception as represented by a Throwable object.
public String toString()	Returns the name of the class concatenated with the result of getMessage().
public void printStackTrace()	Prints the result of toString() along with the stack trace to System.err, the error output stream.
public StackTraceElement [] getStackTrace()	Returns an array containing each element on the stack trace. The element at index 0 represents the top of the call stack, and the last element in the array represents the method at the bottom of the call stack.
public Throwable fillInStackTrace()	Fills the stack trace of this Throwable object with the current stack trace, adding to any previous information in the stack trace.

Exception handling in java uses the following Keywords

1. try
2. catch
3. finally
4. throw
5. throws

The try/catch block is used as follows:

```
try
 {
// block of code to monitor for errors
// the code you think can raise an exception
}
catch (ExceptionType1 exOb)
 {
// exception handler for ExceptionType1
}
catch (ExceptionType2 exOb)
{
// exception handler for ExceptionType
}
// optional
finally {
// block of code to be executed after try block ends
}
```

throwing and catching exceptions

Catching Exceptions

A method catches an exception using a combination of the **try** and **catch** keywords. The program code that may generate an exception should be placed inside the try/catch block. The syntax for try/catch is depicted as below−

Syntax

```
try {
// Protected code
} catch (ExceptionName e1) {
// Catch block
}
```

The code which is prone to exceptions is placed in the try block. When an exception occurs, that exception is handled by catch block associated with it. Every try block should be immediately followed either by a catch block or finally block.

A catch statement involves declaring the type of exception that might be tried to catch. If an exception occurs, then the catch block (or blocks) which follow the try block is checked. If the type of exception that occurred is listed in a catch block, the exception is passed to the catch block similar to an argument that is passed into a method parameter.

To illustrate the try-catch blocks the following program is developed.

```
class Exception_example {
public static void main(String args[])
{
 int a,b;
try { // monitor a block of code. a
 = 0;
 b = 10 / a; //raises the arithmetic exception
System.out.println("Try block.");
}
 catch (ArithmeticException e)
{ // catch divide-by-zero error
System.out.println("Division by zero.");
 }
 System.out.println("After try/catch block.");
 }
 }
```

Output:

Division by zero.

After try/catch block.

Multiple catch clauses

In some cases, more than one exception could be raised by a single piece of code. To handle this multiple exceptions, two or more catch clauses can be specified. Here, each catch block catches 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 MultiCatch_Example {
 public static void main(String args[]) {
try {
int a,b;
a = args.length;
System.out.println("a = " + a);
b = 10 / a; //may cause division-by-zero error
int arr[] = { 10,20 };
 c[5] =100;
```

```
}
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 the execution of the program in both ways:

```
C:\>java MultiCatch_
Example a = 0
Divide by 0: java.lang.ArithmeticException: / by zero
After try/catch blocks.
C:\>java MultiCatch_Example arg1
a = 1
Array index oob: java.lang.ArrayIndexOutOfBoundsException:5
After try/catch blocks.
```

While the multiple catch statements is used, it is important to remember that exception subclasses must come before their superclasses. A catch statement which uses a superclass will catch exceptions of that type plus any of its subclasses. Thus, a subclass would never be reached if it came after its superclass. And also, in Java, unreachable code is an error. For example, consider the following program:

```
class MultiCatch_Example {
public static void main(String args[]) {
try {
int a,b;
a = args.length;
System.out.println("a = " + a);
b = 10 / a; //may cause division-by-zero error
int arr[] = { 10,20 };
c[5] =100;
}
```

```
catch(Exception e) { System.out.println("Generic
Exception catch.");
}
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.");
 }
 }
```

The exceptions such as ArithmeticException, and ArrayIndexOutOfBoundsException are the subclasses of Exception class. The catch statement after the base class catch statement is raising the unreachable code exception.

nested try block

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.

```
try
{
   statement 1;
   statement 2;
   try
   {
      statement 1;
      statement 2;
   }
   catch(Exception e)
   {
   }
}
catch(Exception e)
{
}
```

....

The following program is an example for Nested try statements.

```
class Nestedtry_Example{
 public static void main(String args[]){
  try{
    try{
     System.out.println("division");
     int a,b;
     a=0;
     b =10/a;
    }
    catch(ArithmeticException e)
   {
   System.out.println(e);
  }
  try
  {
   int a[]=new int[5];
   a[6]=3;
   }
   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 Java throw keyword is used to explicitly throw an exception. The general form of throw is shown below:

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 to obtain a Throwable object:

1. using a parameter in a catch clause
2. creating one with the new operator.

The following program explains the use of throw keyword.

```
public class TestThrow1{
static void validate(int age){
try{
    if(age<18)
     throw new ArithmeticException("not valid");
    else
     System.out.println("welcome to vote");
  }
   Catch(ArithmeticException e)
 {
   System.out.println("Caught inside ArithmeticExceptions.");
  throw e; // rethrow the exception
 }
 }
   public static void main(String args[]){
try{
validate(13);
}
Catch(ArithmeticException e)
 {
   System.out.println("ReCaught ArithmeticExceptions.");
 }
}
}
```

The flow of execution stops immediately after the throw statement and any subsequent statements that 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.

the throws/throw Keywords

If a method does not handle a checked exception, the method must be declared using the throws keyword. The throws keyword appears at the end of a method's signature.

The difference between throws and throw keywords is that, *throws* is used to postpone the handling of a checked exception and *throw* is used to invoke an exception explicitly.

The following method declares that it throws a Remote Exception –

Example

```
import java.io.*;
public class throw_Example1 {
public void function(int a) throws RemoteException {
// Method implementation throw
new RemoteException();


  } // Remainder of class definition
}
```

A method can declare that it throws more than one exception, in which case the exceptions are declared in a list separated by commas. For example, the following method declares that it throws a RemoteException and an ArithmeticException –

```
import java.io.*;
public class throw_Example2 {
public void function(int a) throws RemoteException,ArithmeticException {
// Method implementation
  }
 // Remainder of class definition
}
```

the Finally Block

The finally block follows a try block or a catch block. A finally block of code always executes, irrespective of the occurrence of an Exception. A finally block appears at the end of the catch blocks that follows the below syntax.

Syntax

```
try {
  // Protected code
} catch (ExceptionType1 e1) {
  // Catch block
} catch (ExceptionType2 e2) {
```

```
    // Catch block
  }
  finally {
    // The finally block always executes.
    }
```

Example

```
public class Finally_Example {
  public static void main(String args[]) {
    try {
            int a,b;
        a=0;
        b=10/a;
    } catch (ArithmeticException e) {
    System.out.println("Exception thrown :" + e);
   }finally {
    System.out.println("The finally block is executed");
    }
   }
   }
```