

# EXCEPTION

## Exceptional Handling

An exception is a problem that arises during the execution of a program. An exception can occur for many different reasons, including the following:

- A user has entered invalid data.
- A file that needs to be opened cannot be found.
- A network connection has been lost in the middle of communications, or the JVM has run out of memory

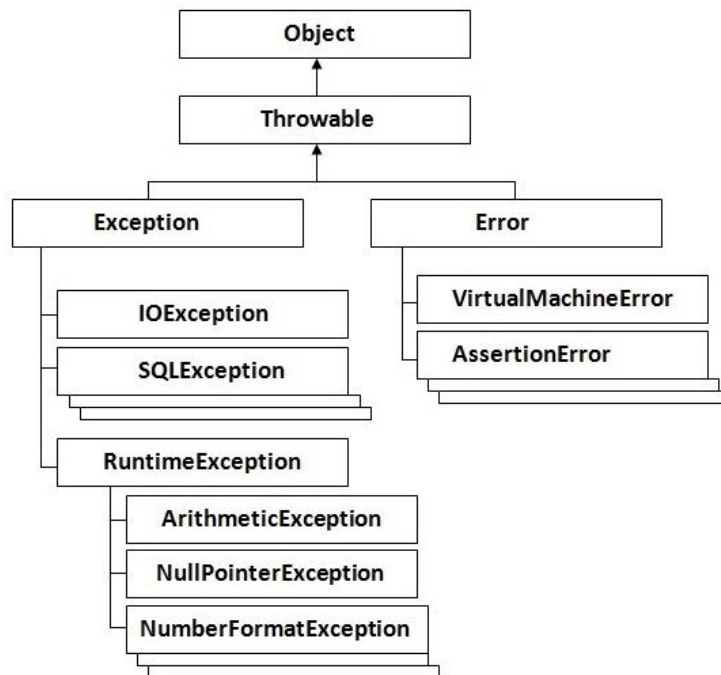
*“Exceptional Handling is a task to maintain normal flow of the program. For this we should try to catch the exception object thrown by the error condition and then display appropriate message for taking corrective actions”*

## Types of Exceptions

- 1. Checked Exception:** A checked exception is an exception that is typically a user error or a problem that cannot be foreseen by the programmer. Checked exception can also be defined as *“The classes that extend the Throwable class except RuntimeException and Error are known as Checked Exceptions”*. For example, if a file is to be opened, but the file cannot be found, an exception occurs. These exceptions are checked at compile-time and cannot simply be ignored at the time of compilation. Example of Checked Exception are IOException, SQLException etc.
- 2. Unchecked Exception:** Also known as Runtime Exceptions and they are ignored at the time of compilation but checked during execution of the program. Unchecked Exceptions can also be defined as *“The Classes that extend the RuntimeException class are known as Unchecked Exceptions”*. Example are ArithmeticException, NullPointerException etc.
- 3. Error:** These are not exceptions at all, but problems that 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.

## Hierarchy of Exception

All exception classes are subtypes of the java.lang.Exception class. The exception class is a subclass of the Throwable class. Other than the exception class there is another subclass called Error which is derived from the Throwable class.



## **Table of JAVA – Built in Exceptions**

Following is the list of Java Unchecked RuntimeException

Exception	Description
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.
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.
StringIndexOutOfBoundsException	Attempt to index outside the bounds of a string.
UnsupportedOperationException	An unsupported operation was encountered.

Following is the list of Java Checked Exceptions Defined in java.lang

Exception	Description
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.

## Handling Exceptions in Java


Following five keywords are used to handle an exception in Java:

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

### try –catch block

A method catches an exception using a combination of the **try and catch** keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following:

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



Write block of code here that is likely to cause an error condition and throws an exception

A catch statement involves declaring the type of exception you are trying to catch. If an exception occurs in protected code, the catch block (or blocks) that follows the try is checked. If the type of exception that occurred is listed in a catch block, the exception is passed to the catch block much as an argument is passed into a method parameter.

## Example of Program without Exceptional Handling

```
class excep1
{
    public static void main(String args[])
    {
        int i=100/Integer.parseInt(args[0]);
        System.out.println("Value of i is:"+i);
    }
}
```

This statement can cause error as divide by Zero is an ArithmeticException

### Output:

```
C:\Achin Jain>java excep1 12
Value of i is:8

C:\Achin Jain>java excep1 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
at excep1.main(excep1.java:5)
```

## Same Program with Exception Handling

```
class excep1
{
    public static void main(String args[])
    {
        try
        {
            int i=100/Integer.parseInt(args[0]);
            System.out.println("Value of i is:"+i);
        }
        catch(ArithmeticException e)
        {
            System.out.println(e);
        }
        System.out.println("Code after try-catch");
    }
}
```

Now as the statement which can cause error condition is wrapped under try block and catch block is also present to handle the exception object thrown. In this case even if there is an error rest of the program will execute normally

### Output

```
C:\Achin Jain>java excep1 12
Value of i is:8
Code after try-catch

C:\Achin Jain>java excep1 0
java.lang.ArithmeticException: / by zero
Code after try-catch
```

## Multiple Catch Blocks:

A try block can be followed by multiple catch blocks, but when we use multiple catch statements it is important that exception subclasses must come before any of their superclasses. The reason is “a catch statement with superclass will catch exceptions of that type plus any of its subclass, thus causing a catch statement with subclass exception a non-reachable code which is error in JAVA”.

### Example:

```
class excep1
{
    public static void main(String args[])
    {
        try
        {
            int i=100/Integer.parseInt(args[0]);
            System.out.println("Value of i is:"+i);
        }
        catch(Exception e1)
        {
            System.out.println(e1);
        }
        catch(ArithmeticException e)
        {
            System.out.println(e);
        }
        System.out.println("Code after try-catch");
    }
}
```

In the example, two catch statement are used but first one is of type Exception which is a superclass of ArithmeticException (used in second catch). So any exception thrown will be caught by first catch block which makes second block unreachable and error is shown during compile time

### Output

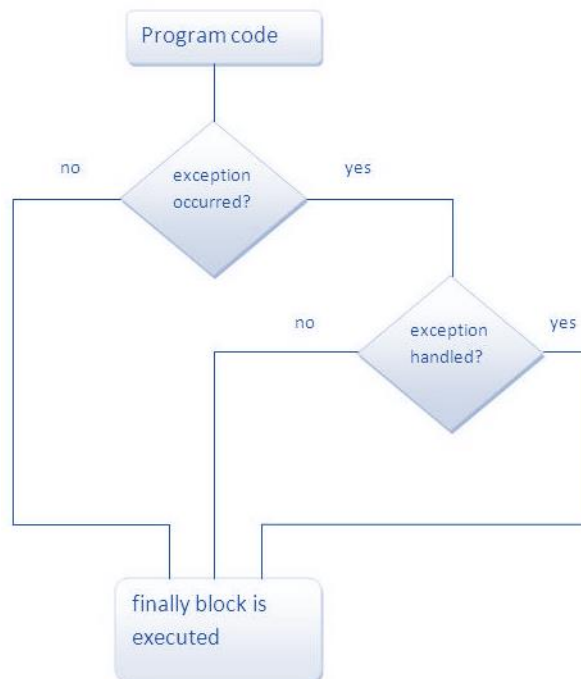
```
C:\Achin Jain>javac excep1.java
excep1.java:14: error: exception ArithmeticException has already been caught
    catch(ArithmeticException e)
    ^
1 error
```

However if the order of the catch blocks is reversed like shown below, then program will execute normally

```
catch(ArithmeticException e)
{
    System.out.println(e);
}
catch(Exception e1)
{
    System.out.println(e1);
}
```

## Finally Block

The finally keyword is used to create a block of code that follows a try block. A finally block of code always executes, whether or not an exception has occurred. Using a finally block allows you to run any cleanup-type statements that you want to execute, no matter what happens in the protected code.



## Example of Finally Statement

```
class excep1
{
    public static void main(String args[])
    {
        try
        {
            int i=100/Integer.parseInt(args[0]);
            System.out.println("Value of i is:"+i);
        }
        catch(ArithmeticException e1)
        {
            System.out.println(e1);
        }
        finally
        {
            System.out.println("Finally Code Executed");
        }
        System.out.println("Code after try-catch");
    }
}
```

## Output 1

In the first case no command line arguments are passed which will throw `ArrayIndexOutOfBoundsException` and in the above code we are handling only `ArithmeticException` which will cause the system to terminate and remaining program will not run. But in this case also the statement written in the finally block will get executed as shown below:

```
C:\Achin Jain>java excep1
Finally Code Executed
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0
    at excep1.main(excep1.java:7)
```

In second case '0' is passed as command line argument to let program throw `ArithmeticException` which will eventually be handled by catch block. See the output below which clearly shows that remaining part of the code will also run along with finally statement.

```
C:\Achin Jain>java excep1 0
java.lang.ArithmeticException: / by zero
Finally Code Executed
Code after try-catch
```

In third case '5' is passed as command line argument which is perfectly fine and in this case no exception will be thrown. Now see the output below, in this case also finally statement will get executed.

```
C:\Achin Jain>java excep1 5
Value of i is:20
Finally Code Executed
Code after try-catch
```

## Throw Keyword

The throw keyword is used to explicitly throw an exception. We can throw either checked or unchecked exception. The throw keyword is normally used to throw custom exception.

## Example

In the example shown below a method `validage(int i)` is used which will check the value of passed parameter *i* and if the value is less than 18 then a `ArithmeticException` is thrown. Now as you can see when we have called the method no try catch block is used which results in termination of the program and message is displayed as "not valid" which is passed during throwing of `ArithmeticException` object.

```

class excep1
{
    static void validage(int i)
    {
        if(i<18)
        {
            throw new ArithmeticException("not valid");
        }
        else
        {
            System.out.println("Welcome");
        }
    }
    public static void main(String args[])
    {
        validage(12);
        System.out.println("Code after try-catch");
    }
}

```

### Output

```

C:\Achin Jain>javac excep1.java
C:\Achin Jain>java excep1
Exception in thread "main" java.lang.ArithmeticException: not valid
    at excep1.validage(excep1.java:7)
    at excep1.main(excep1.java:16)

```

However if during call of validage method try-catch block has been used then the program will run normally

```

try
{
    validage(12);
}
catch(ArithmeticException e)
{
    System.out.println(e);
}

```

### Output

```

C:\Achin Jain>javac excep1.java
C:\Achin Jain>java excep1
java.lang.ArithmeticException: not valid
Code after try-catch

```



## Throws Keyword

The throws keyword is used to declare the exception, it provide information to the programmer that there may occur an exception so during call of that method, and programmer must use exceptional handling mechanism. Throws keyword is also used to propagate checked exception.

## Example

In this example, exception is created by extending Exception class and the custom exception is declared in the method validage(int i)

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

class excep2
{
    static void validage(int i) throws ajexception
    {
        if(i<18)
        {
            throw new ajexception("not valid");
        }
    }
    public static void main(String args[])
    {
        validage(12);
        System.out.println("Code after try-catch");
    }
}
```

Code to create custom exception with name "ajexception"

**Case 1:** During call of validage method exceptional handling is not used and code looks like this and error is displayed in the compilation of the code.

```
public static void main(String args[])
{
    validage(12);
    System.out.println("Code after try-catch");
}
```

## Output

```
C:\Achin Jain>javac excep2.java
excep2.java:19: error: unreported exception ajexception; must be caught or declared to be thrown
    validage(12);
    ^
```

**Case 2:** During call of method validage exceptional handling is used with try-catch keyword like this and the program runs as expected.

```
try
{
    validage(12);
}
catch(ajexception aj)
{
    System.out.println(aj);
}
```

### **Output:**

```
C:\Achin Jain>javac excep2.java
C:\Achin Jain>java excep2
ajexception: not valid
Code after try-catch
```

**Case 3:** During call of method validage exceptional handling is used without try-catch keyword and throws keyword is used in main method as shown below

```
public static void main(String args[]) throws ajexception
{
    validage(12);
    System.out.println("Code after try-catch");
}
```

There will be no error now during compile time, but program will gets terminated when exception event takes place.

### **Output**

```
C:\Achin Jain>javac excep2.java
C:\Achin Jain>java excep2
Exception in thread "main" ajexception: not valid
    at excep2.validage(excep2.java:14)
    at excep2.main(excep2.java:19)
```

**Case 4:** Try to propagate custom exception not of type RuntimeException without declaring in method using throws keyword. This will give compile time error

```
static void validage(int i) //throws ajexception
{
    if(i<18)
    {
        throw new ajexception("not valid");
    }
}
```

### Output:

```
C:\Achin Jain>javac excep2.java
excep2.java:14: error: unreported exception ajexception; must be caught or declared to be thrown
        throw new ajexception("not valid");
              ^
1 error
```

**Case 5:** Make custom exception by extending RuntimeException Class and try the same method as use for Case 4. There will be no error now and the program runs as expected

```
class ajexception extends RuntimeException
{
    ajexception(String s)
    {
        super(s);
    }
}
```

### Output

```
C:\Achin Jain>javac excep2.java
C:\Achin Jain>java excep2
ajexception: not valid
Code after try-catch
```

### Important Points in Exceptional Handling

- A catch clause cannot exist without a try statement.
- It is not compulsory to have finally clauses whenever a try/catch block is present.
- The try block cannot be present without either catch clause or finally clause.
- Any code cannot be present in between the try, catch, finally blocks.

### Declaring your Own Exceptions

You can create your own exceptions in Java. Keep the following points in mind when writing your own exception classes:

- All exceptions must be a child of Throwable.
- If you want to write a checked exception that is automatically enforced by the Handle or Declare Rule, you need to extend the Exception class.
- If you want to write a runtime exception, you need to extend the RuntimeException class.

[Example to create custom exception](#) is shown in the section above.