

7.1 Exception and its types

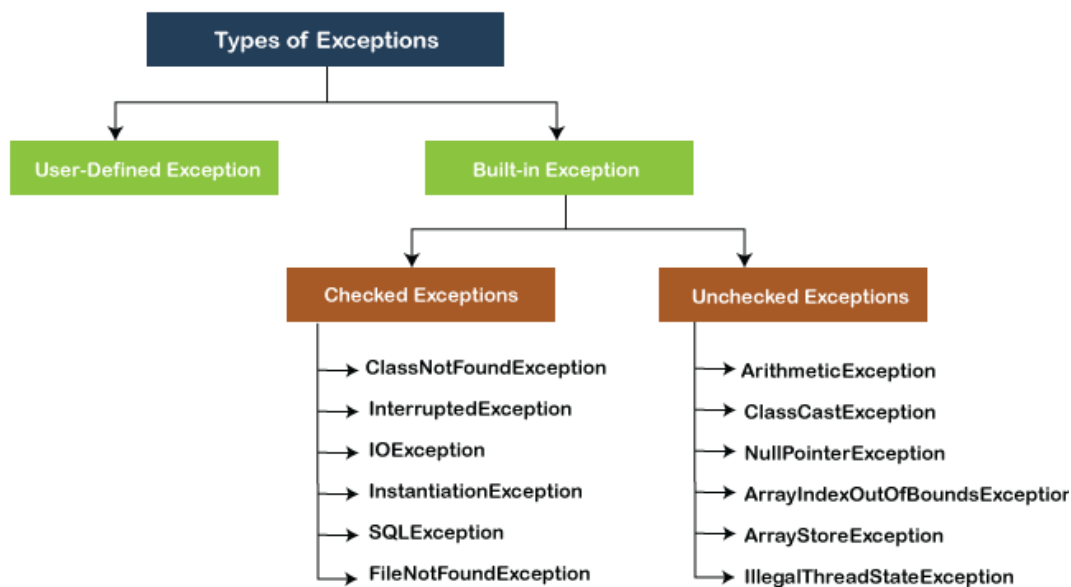
Exceptions are the unwanted errors or bugs or events that restrict the normal execution of a program. Each time an exception occurs, program execution gets disrupted. An error message is displayed on the screen.

There are several reasons behind the occurrence of exceptions. These are some conditions where an exception occurs:

- Whenever a user provides invalid data.
- The file requested to be accessed does not exist in the system.
- When the **Java Virtual Machine (JVM)** runs out of memory.
- Network drops in the middle of communication.
- The parent class of all the exception classes is the **java.lang.Exception** class. Figure 1 illustrates the different types of Java exceptions.

Exceptions can be categorized into two ways:

1. Built-in Exceptions
 - Checked Exception
 - Unchecked Exception
2. User-Defined Exceptions



Built-in Exception

Exceptions that are already available in **Java libraries** are referred to as **built-in exception**. These exceptions are able to define the error situation so that we can understand the reason of getting this error. It can be categorized into two broad categories, i.e., **checked exceptions** and **unchecked exception**.

Checked Exception

Checked exceptions are called **compile-time** exceptions because these exceptions are checked at compile-time by the compiler. The compiler ensures whether the programmer handles the exception or not. The programmer should have to handle the exception; otherwise, the system has shown a compilation error.

CheckedExceptionExample.java

```
1. import java.io.*;
2. class CheckedExceptionExample {
3.     public static void main(String args[]) {
4.         FileInputStream file_data = null;
5.         file_data = new FileInputStream("C:/Users/ajeet/OneDrive/Desktop/Hello.txt");
6.         int m;
7.         while(( m = file_data.read() ) != -1) {
8.             System.out.print((char)m);
9.         }
10.        file_data.close();
11.    }
12. }
```

In the above code, we are trying to read the **Hello.txt** file and display its data or content on the screen. The program throws the following exceptions:

1. The **FileInputStream(File filename)** constructor throws the **FileNotFoundException** that is checked exception.
2. The **read()** method of the **FileInputStream** class throws the **IOException**.
3. The **close()** method also throws the **IOException**.

Output:



```
C:\Windows\System32\cmd.exe
input1 = new FileInputStream("C:/Users/ajeet/OneDrive/Desktop/programs.txt"
);
    ^
Exception.java:9: error: unreported exception IOException; must be caught or declar
ed to be thrown
    while(( m = input1.read() ) != -1) {
    ^
Exception.java:13: error: unreported exception IOException; must be caught or decla
red to be thrown
    input1.close();
    ^
3 errors
C:\Users\ajeet\OneDrive\Desktop\programs>
```

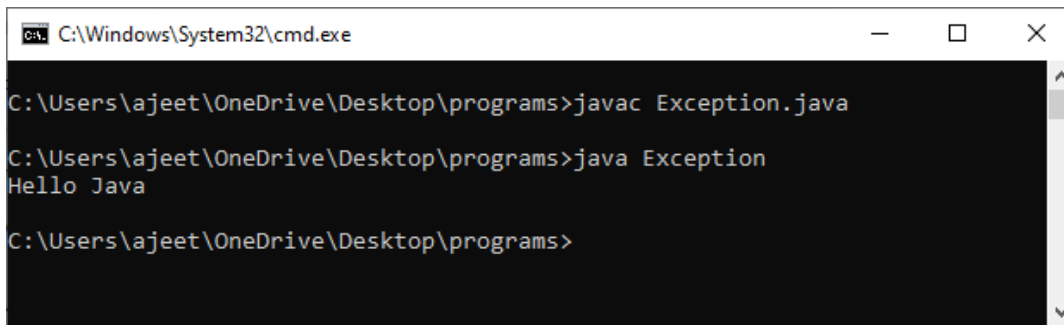
How to resolve the error?

There are basically two ways through which we can solve these errors.

1) The exceptions occur in the main method. We can get rid from these compilation errors by declaring the exception in the main method using **the throws**. We only declare the **IOException**, not **FileNotFoundException**, because of the child-parent relationship. The **IOException** class is the parent class of **FileNotFoundException**, so this exception will automatically cover by **IOException**. We will declare the exception in the following way:

```
1. class Exception{
2.     public static void main(String args[]) throws IOException {
3.         ...
4.         ...
5.     }
```

If we compile and run the code, the errors will disappear, and we will see the data of the file.



```
C:\Windows\System32\cmd.exe

C:\Users\ajeet\OneDrive\Desktop\programs>javac Exception.java

C:\Users\ajeet\OneDrive\Desktop\programs>java Exception
Hello Java

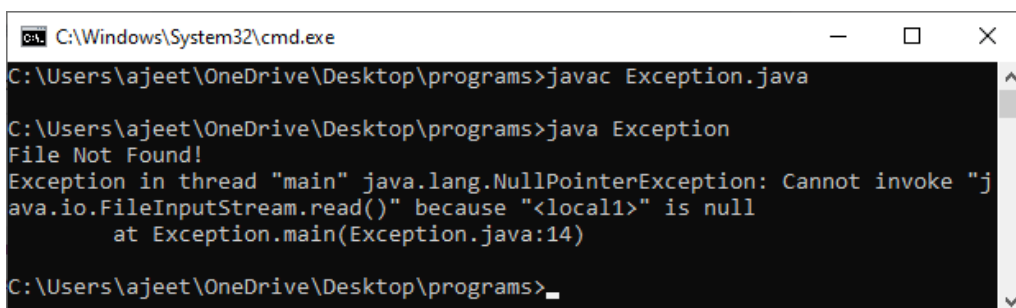
C:\Users\ajeet\OneDrive\Desktop\programs>
```

2) We can also handle these exception using **try-catch**. However, the way which we have used above is not correct. We have to give meaningful message for each exception type. By doing that it would be easy to understand the error. We will use the try-catch block in the following way:

Exception.java

```
1. import java.io.*;
2. class Exception{
3.     public static void main(String args[]) {
4.         FileInputStream file_data = null;
5.         try{
6.             file_data = new FileInputStream("C:/Users/ajeet/OneDrive/Desktop/programs/Hell.txt");
7.         }catch(FileNotFoundException fnfe){
8.             System.out.println("File Not Found!");
9.         }
10.        int m;
11.        try{
12.            while(( m = file_data.read() ) != -1) {
13.                System.out.print((char)m);
14.            }
15.            file_data.close();
16.        }catch(IOException ioe){
17.            System.out.println("I/O error occurred: "+ioe);
18.        }
19.    }
20. }
```

We will see a proper error message **"File Not Found!"** on the console because there is no such file in that location.



```
C:\Windows\System32\cmd.exe

C:\Users\ajeet\OneDrive\Desktop\programs>javac Exception.java

C:\Users\ajeet\OneDrive\Desktop\programs>java Exception
File Not Found!
Exception in thread "main" java.lang.NullPointerException: Cannot invoke "java.io.FileInputStream.read()" because "<local1>" is null
    at Exception.main(Exception.java:14)

C:\Users\ajeet\OneDrive\Desktop\programs>
```

Unchecked Exceptions

The **unchecked** exceptions are just opposite to the **checked** exceptions. The compiler will not check these exceptions at compile time. In simple words, if a program throws an unchecked exception, and even if we didn't handle or declare it, the program would not give a compilation error. Usually, it occurs when the user provides bad data during the interaction with the program.

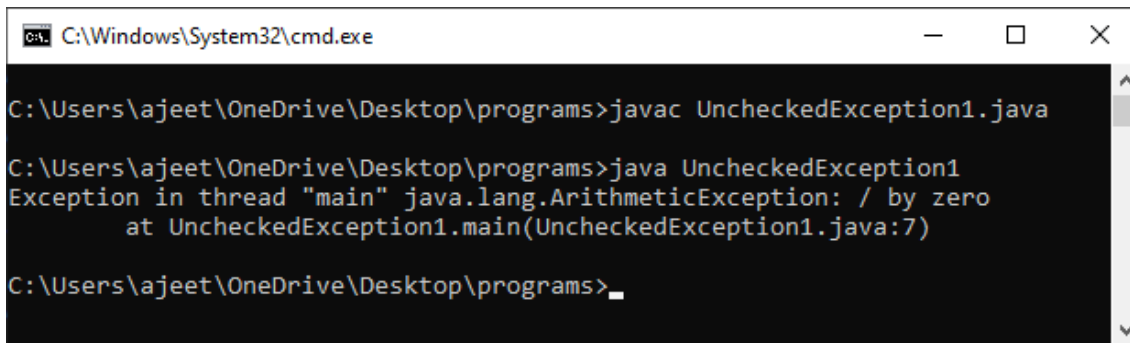
Note: The RuntimeException class is able to resolve all the unchecked exceptions because of the child-parent relationship.

UncheckedExceptionExample1.java

```
1. class UncheckedExceptionExample1 {
2.     public static void main(String args[])
3.     {
4.         int positive = 35;
5.         int zero = 0;
6.         int result = positive/zero;
7.         //Give Unchecked Exception here.
8.         System.out.println(result);
9.     }
10. }
```

In the above program, we have divided 35 by 0. The code would be compiled successfully, but it will throw an ArithmeticException error at runtime. On dividing a number by 0 throws the divide by zero exception that is a unchecked exception.

Output:



```
C:\Windows\System32\cmd.exe

C:\Users\ajeet\OneDrive\Desktop\programs>javac UncheckedException1.java

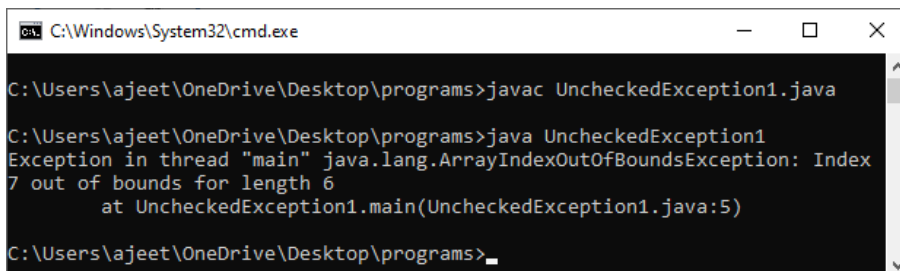
C:\Users\ajeet\OneDrive\Desktop\programs>java UncheckedException1
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at UncheckedException1.main(UncheckedException1.java:7)

C:\Users\ajeet\OneDrive\Desktop\programs>_
```

UncheckedException1.java

```
1. class UncheckedException1 {
2.     public static void main(String args[])
3.     {
4.         int num[]={10,20,30,40,50,60};
5.         System.out.println(num[7]);
6.     }
7. }
```

Output:



```
C:\Windows\System32\cmd.exe

C:\Users\ajeet\OneDrive\Desktop\programs>javac UncheckedException1.java

C:\Users\ajeet\OneDrive\Desktop\programs>java UncheckedException1
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index
7 out of bounds for length 6
    at UncheckedException1.main(UncheckedException1.java:5)

C:\Users\ajeet\OneDrive\Desktop\programs>_
```

In the above code, we are trying to get the element located at position 7, but the length of the array is 6. The code compiles successfully, but throws the ArrayIndexOutOfBoundsException at runtime.

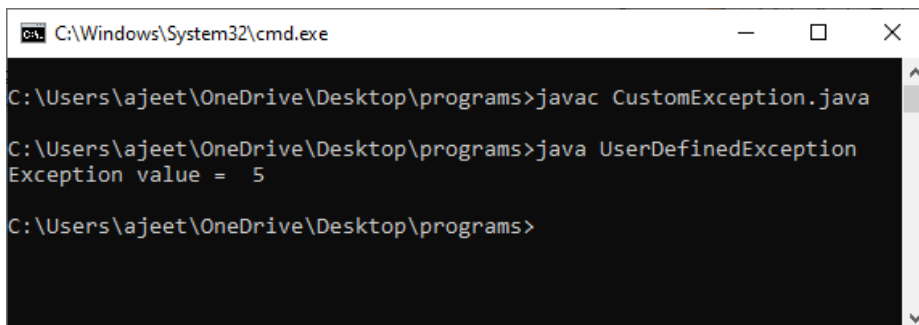
User-defined Exception

In [Java](#), we already have some built-in exception classes like **ArrayIndexOutOfBoundsException**, **NullPointerException**, and **ArithmeticException**. These exceptions are restricted to trigger on some predefined conditions. In Java, we can write our own exception class by extends the **Exception** class. We can throw our own exception on a particular condition using the **throw** keyword. For creating a user-defined exception, we should have basic knowledge of the **try-catch** block and [throw](#) keyword.

UserDefinedException.java

```
1. import java.util.*;
2. class UserDefinedException{
3.     public static void main(String args[]){
4.         try{
5.             throw new NewException(5);
6.         }
7.         catch(NewException ex){
8.             System.out.println(ex);
9.         }
10.    }
11. }
12. class NewException extends Exception{
13.     int x;
14.     NewException(int y) {
15.         x=y;
16.     }
17.     public String toString(){
18.         return ("Exception value = "+x);
19.     }
20. }
```

Output:



```
C:\Windows\System32\cmd.exe
C:\Users\ajeet\OneDrive\Desktop\programs>javac CustomException.java
C:\Users\ajeet\OneDrive\Desktop\programs>java UserDefinedException
Exception value = 5
C:\Users\ajeet\OneDrive\Desktop\programs>
```

Description:

In the above code, we have created two classes, i.e., **UserDefinedException** and **NewException**. The **UserDefinedException** has our main method, and the **NewException** class is our user-defined exception class, which extends **exception**. In the **NewException** class, we create a variable **x** of type integer and assign a value to it in the constructor. After assigning a value to that variable, we return the exception message.

In the **UserDefinedException** class, we have added a **try-catch** block. In the try section, we throw the exception, i.e., **NewException** and pass an integer to it. The value will be passed to the **NewException** class and return a message. We catch that message in the catch block and show it on the screen.

7.2 Exception handling fundamentals

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

he 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 need to handle exceptions. Let's consider a scenario:

1. statement 1;
2. statement 2;
3. statement 3;
4. statement 4;
5. statement 5;//exception occurs
6. statement 6;
7. statement 7;
8. statement 8;
9. statement 9;
10. statement 10;

Suppose there are 10 statements in a Java program and an exception occurs at statement 5; the rest of the code will not be executed, i.e., statements 6 to 10 will not be executed. However, when we perform exception handling, the rest of the statements will be executed. That is why we use exception handling in Java.

Blocks and Keywords used for Exception Handling

Try

The [try](#) block contains a set of statements where an exception can occur.

```
try
{
    // statement(s) that might cause exception
}
```

Catch

The catch block is used to handle the uncertain condition of a try block. A try block is always followed by a catch block, which handles the exception that occurs in the associated try block.

```
catch
{
    // statement(s) that handle an exception
    // examples, closing a connection, closing
    // file, exiting the process after writing
    // details to a log file.
}
```

Throw

The throw keyword is used to transfer control from the try block to the catch block.

Below is the implementation of the above approach:

```
// Java program that demonstrates the use of throw
class ThrowExcep {
    static void help()
    {
        try {
            throw new NullPointerException("error_unknown");
        }
        catch (NullPointerException e) {
            System.out.println("Caught inside help().");
            // rethrowing the exception
        }
    }
}
```

```
        throw e;
    }
}

public static void main(String args[])
{
    try {
        help();
    }
    catch (NullPointerException e) {
        System.out.println(
            "Caught in main error name given below:");
        System.out.println(e);
    }
}
}
```

Output

Caught inside help().

Caught in main error name given below:

java.lang.NullPointerException: error_unknown

Throws

The [throws](#) keyword is used for exception handling without try & catch block. It specifies the exceptions that a method can throw to the caller and does not handle itself.

Below is the implementation of the above approach:

```
// Java program to demonstrate working of throws
class ThrowsExcep {

    // This method throws an exception
    // to be handled
    // by caller or caller
    // of caller and so on.
    static void fun() throws IllegalAccessException
    {
        System.out.println("Inside fun(). ");
        throw new IllegalAccessException("demo");
    }

    // This is a caller function
    public static void main(String args[])
    {
        try {
            fun();
        }
        catch (IllegalAccessException e) {
            System.out.println("caught in main.");
        }
    }
}
```

Output

Inside fun().

caught in main.

Finally

It is executed after the catch block. We use it to put some common code (to be executed irrespective of whether an exception has occurred or not) when there are multiple catch blocks.

An example of an exception generated by the system is given below:

```
Exception in thread "main"  
java.lang.ArithmeticException: divide  
by zero at ExceptionDemo.main(ExceptionDemo.java:5)  
ExceptionDemo: The class name  
main:The method name  
ExceptionDemo.java:The file name  
java:5:line number
```

Below is the implementation of the above approach:

```
// Java program to demonstrate working of try,  
// catch and finally  
  
class Division {  
    public static void main(String[] args)  
    {  
        int a = 10, b = 5, c = 5, result;  
        try {  
            result = a / (b - c);  
            System.out.println("result" + result);  
        }  
  
        catch (ArithmeticException e) {  
            System.out.println("Exception caught:Division by zero");  
        }  
  
        finally {  
            System.out.println("I am in final block");  
        }  
    }  
}
```

Output

```
Exception caught:Division by zero  
I am in final block
```

Java supports extending interfaces, allowing you to create a new interface that inherits the method declarations of one or more existing interfaces. Here's an example:

7.3 Using try and catch

Java try block

Java **try** block is used to enclose the code that might throw an exception. It must be used within the method.

If an exception occurs at the particular statement in the try block, the rest of the block code will not execute. So, it is recommended not to keep the code in try block that will not throw an exception.

Java try block must be followed by either catch or finally block.

Syntax of Java try-catch

1. try{
2. //code that may throw an exception
3. }catch(Exception_class_Name ref){ }

Syntax of try-finally block

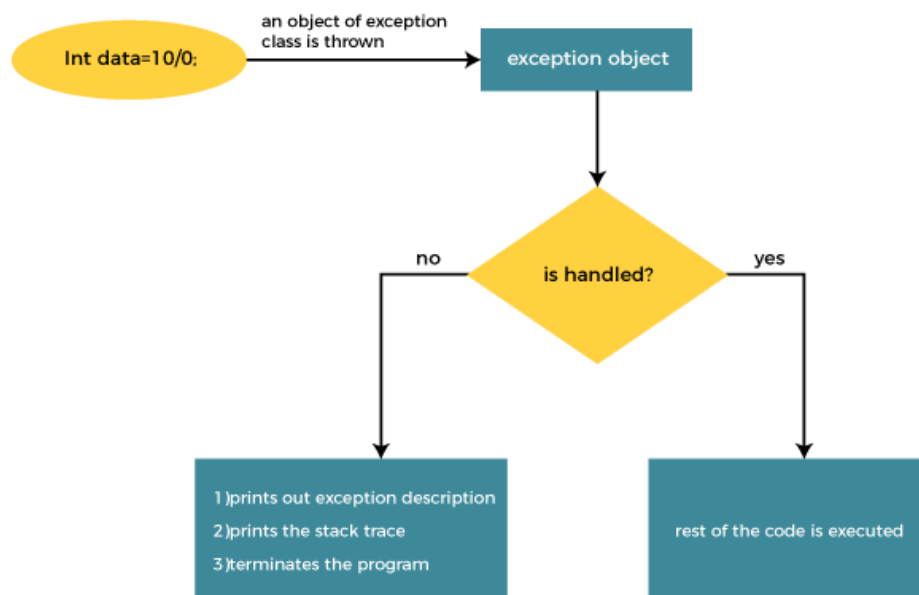
1. try{
2. //code that may throw an exception
3. }finally{ }

Java catch block

Java catch block is used to handle the Exception by declaring the type of exception within the parameter. The declared exception must be the parent class exception (i.e., Exception) or the generated exception type. However, the good approach is to declare the generated type of exception.

The catch block must be used after the try block only. You can use multiple catch block with a single try block.

Internal Working of Java try-catch block



The JVM firstly checks whether the exception is handled or not. If exception is not handled, JVM provides a default exception handler that performs the following tasks:

- Prints out exception description.
- Prints the stack trace (Hierarchy of methods where the exception occurred).
- Causes the program to terminate.

But if the application programmer handles the exception, the normal flow of the application is maintained, i.e., rest of the code is executed.

Problem without exception handling

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

Example 1

TryCatchExample1.java

```
1. public class TryCatchExample1 {
2.
3.     public static void main(String[] args) {
4.
5.         int data=50/0; //may throw exception
6.
7.         System.out.println("rest of the code");
8.
9.     }
10.
11. }
```

Output:

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

As displayed in the above example, the **rest of the code** is not executed (in such case, the **rest of the code** statement is not printed).

There might be 100 lines of code after the exception. If the exception is not handled, all the code below the exception won't be executed.

Solution by exception handling

Let's see the solution of the above problem by a java try-catch block.

Example 2

TryCatchExample2.java

```
1. public class TryCatchExample2 {
2.
3.     public static void main(String[] args) {
4.         try
5.         {
6.             int data=50/0; //may throw exception
7.         }
8.         //handling the exception
9.         catch(ArithmeticException e)
10.        {
11.            System.out.println(e);
12.        }
13.        System.out.println("rest of the code");
14.    }
15.
16. }
```

Output:

java.lang.ArithmeticException: / by zero
rest of the code

As displayed in the above example, the **rest of the code** is executed, i.e., the **rest of the code** statement is printed.

Example 3

In this example, we also kept the code in a try block that will not throw an exception.

TryCatchExample3.java

```
1. public class TryCatchExample3 {
2.
3.     public static void main(String[] args) {
4.         try
5.         {
6.             int data=50/0; //may throw exception
7.             // if exception occurs, the remaining statement will not execute
8.             System.out.println("rest of the code");
9.         }
10.        // handling the exception
11.        catch(ArithmeticException e)
12.        {
13.            System.out.println(e);
14.        }
15.
16.    }
17.
18. }
```

Output:

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

Here, we can see that if an exception occurs in the try block, the rest of the block code will not execute.

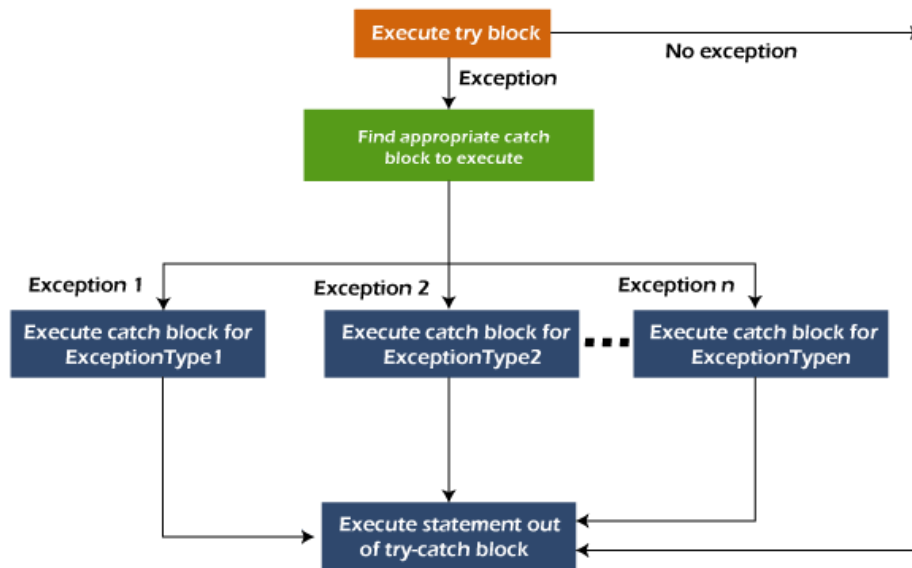
Java Multi-catch block

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.

Flowchart of Multi-catch Block



Example 1

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

MultipleCatchBlock1.java

```

1. public class MultipleCatchBlock1 {
2.
3.     public static void main(String[] args) {
4.
5.         try{
6.             int a[]=new int[5];
7.             a[5]=30/0;
8.         }
9.         catch(ArithmeticException e)
10.        {
11.            System.out.println("Arithmetic Exception occurs");
12.        }
13.        catch(ArrayIndexOutOfBoundsException e)
14.        {
15.            System.out.println("ArrayIndexOutOfBoundsException occurs");
16.        }
17.        catch(Exception e)
18.        {
19.            System.out.println("Parent Exception occurs");
20.        }
21.        System.out.println("rest of the code");
22.    }
23. }
  
```

Output:

Arithmetic Exception occurs
rest of the code

Example 2

MultipleCatchBlock2.java

```
1. public class MultipleCatchBlock2 {
2.
3.     public static void main(String[] args) {
4.
5.         try{
6.             int a[]=new int[5];
7.
8.             System.out.println(a[10]);
9.         }
10.        catch(ArithmeticException e)
11.        {
12.            System.out.println("Arithmetic Exception occurs");
13.        }
14.        catch(ArrayIndexOutOfBoundsException e)
15.        {
16.            System.out.println("ArrayIndexOutOfBoundsException Exception occurs");
17.        }
18.        catch(Exception e)
19.        {
20.            System.out.println("Parent Exception occurs");
21.        }
22.        System.out.println("rest of the code");
23.    }
24. }
```

Output:

```
ArrayIndexOutOfBoundsException Exception occurs
rest of the code
```

In this example, try block contains two exceptions. But at a time only one exception occurs and its corresponding catch block is executed.

7.4 Using throw and throws

In Java, exceptions allows us to write good quality codes where the errors are checked at the compile time instead of runtime and we can create custom exceptions making the code recovery and debugging easier.

Java throw keyword

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

We specify the **exception** object which is to be thrown. The Exception has some message with it that provides the error description. These exceptions may be related to user inputs, server, etc.

We can throw either checked or unchecked exceptions in Java by throw keyword. It is mainly used to throw a custom exception. We will discuss custom exceptions later in this section.

We can also define our own set of conditions and throw an exception explicitly using throw keyword. For example, we can throw ArithmeticException if we divide a number by another number. Here, we just need to set the condition and throw exception using throw keyword.

The syntax of the Java throw keyword is given below.

throw Instance i.e.,

1. `throw new exception_class("error message");`

Let's see the example of throw IOException.

1. `throw new IOException("sorry device error");`

Where the Instance must be of type Throwable or subclass of Throwable. For example, Exception is the sub class of Throwable and the user-defined exceptions usually extend the Exception class.

Java throw keyword Example

Example 1: Throwing Unchecked Exception

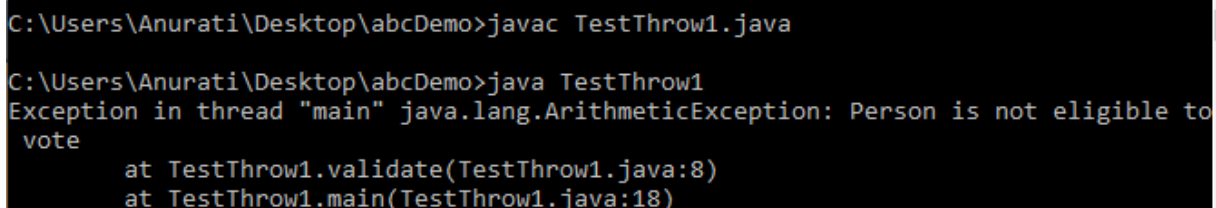
In this example, we have created a method named validate() that accepts an integer as a parameter. If the age is less than 18, we are throwing the ArithmeticException otherwise print a message welcome to vote.

TestThrow1.java

In this example, we have created the validate method that takes integer value as a parameter. If the age is less than 18, we are throwing the ArithmeticException otherwise print a message welcome to vote.

```
1. public class TestThrow1 {
2.     //function to check if person is eligible to vote or not
3.     public static void validate(int age) {
4.         if(age<18) {
5.             //throw Arithmetic exception if not eligible to vote
6.             throw new ArithmeticException("Person is not eligible to vote");
7.         }
8.         else {
9.             System.out.println("Person is eligible to vote!!");
10.        }
11.    }
12.    //main method
13.    public static void main(String args[]){
14.        //calling the function
15.        validate(13);
16.        System.out.println("rest of the code...");
17.    }
18. }
```

Output:



```
C:\Users\Anurati\Desktop\abcDemo>javac TestThrow1.java
C:\Users\Anurati\Desktop\abcDemo>java TestThrow1
Exception in thread "main" java.lang.ArithmeticException: Person is not eligible to
vote
    at TestThrow1.validate(TestThrow1.java:8)
    at TestThrow1.main(TestThrow1.java:18)
```

The above code throw an unchecked exception. Similarly, we can also throw unchecked and user defined exceptions.

Note: If we throw unchecked exception from a method, it is must to handle the exception or declare in throws clause.

If we throw a checked exception using throw keyword, it is must to handle the exception using catch block or the method must declare it using throws declaration.

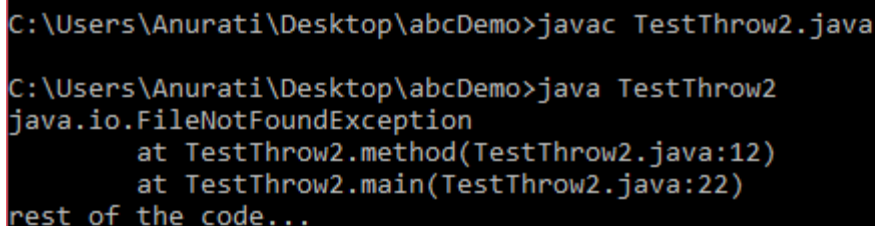
Example 2: Throwing Checked Exception

Note: Every subclass of Error and RuntimeException is an unchecked exception in Java. A checked exception is everything else under the Throwable class.

TestThrow2.java

```
1. import java.io.*;
2.
3. public class TestThrow2 {
4.
5.     //function to check if person is eligible to vote or not
6.     public static void method() throws FileNotFoundException {
7.
8.         FileReader file = new FileReader("C:\\Users\\Anurati\\Desktop\\abc.txt");
9.         BufferedReader fileInput = new BufferedReader(file);
10.
11.
12.         throw new FileNotFoundException();
13.
14.     }
15.     //main method
16.     public static void main(String args[]){
17.         try
18.         {
19.             method();
20.         }
21.         catch (FileNotFoundException e)
22.         {
23.             e.printStackTrace();
24.         }
25.         System.out.println("rest of the code...");
26.     }
27. }
```

Output:



```
C:\Users\Anurati\Desktop\abcDemo>javac TestThrow2.java
C:\Users\Anurati\Desktop\abcDemo>java TestThrow2
java.io.FileNotFoundException
    at TestThrow2.method(TestThrow2.java:12)
    at TestThrow2.main(TestThrow2.java:22)
rest of the code...
```

Example 3: Throwing User-defined Exception

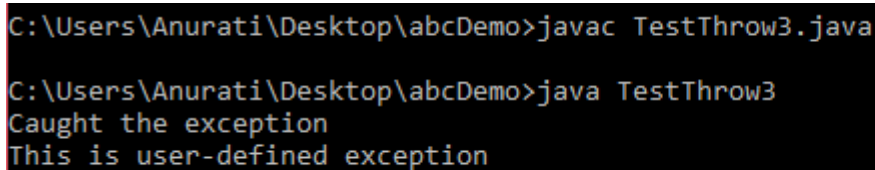
exception is everything else under the Throwable class.

TestThrow3.java

```
1. // class represents user-defined exception
```

```
2. class UserDefinedException extends Exception
3. {
4.     public UserDefinedException(String str)
5.     {
6.         // Calling constructor of parent Exception
7.         super(str);
8.     }
9. }
10. // Class that uses above MyException
11. public class TestThrow3
12. {
13.     public static void main(String args[])
14.     {
15.         try
16.         {
17.             // throw an object of user defined exception
18.             throw new UserDefinedException("This is user-defined exception");
19.         }
20.         catch (UserDefinedException ude)
21.         {
22.             System.out.println("Caught the exception");
23.             // Print the message from MyException object
24.             System.out.println(ude.getMessage());
25.         }
26.     }
27. }
```

Output:



```
C:\Users\Anurati\Desktop\abcDemo>javac TestThrow3.java
C:\Users\Anurati\Desktop\abcDemo>java TestThrow3
Caught the exception
This is user-defined exception
```

Java throws keyword

The **Java throws keyword** is used to declare an exception. It gives an information to the programmer that there may occur an exception. So, it is better for the programmer to provide the exception handling code so that the normal flow of the program can be maintained.

Exception Handling is mainly used to handle the checked exceptions. If there occurs any unchecked exception such as `NullPointerException`, it is programmers' fault that he is not checking the code before it being used.

Syntax of Java throws

```
return_type method_name() throws exception_class_name{

    //method code

}
```

Java throws Example

Let's see the example of Java throws clause which describes that checked exceptions can be propagated by throws keyword.

Testthrows1.java

```
import java.io.IOException;
```



```
class Testthrows1{
    void m() throws IOException{
        throw new IOException("device error");//checked exception
    }
    void n() throws IOException{
        m();
    }
    void p(){
        try{
            n();
        }catch(Exception e){System.out.println("exception handled");}
    }
    public static void main(String args[]){
        Testthrows1 obj=new Testthrows1();
        obj.p();
        System.out.println("normal flow...");
    }
}
```

Output:

```
exception handled
normal flow...
```

Rule: If we are calling a method that declares an exception, we must either caught or declare the exception.