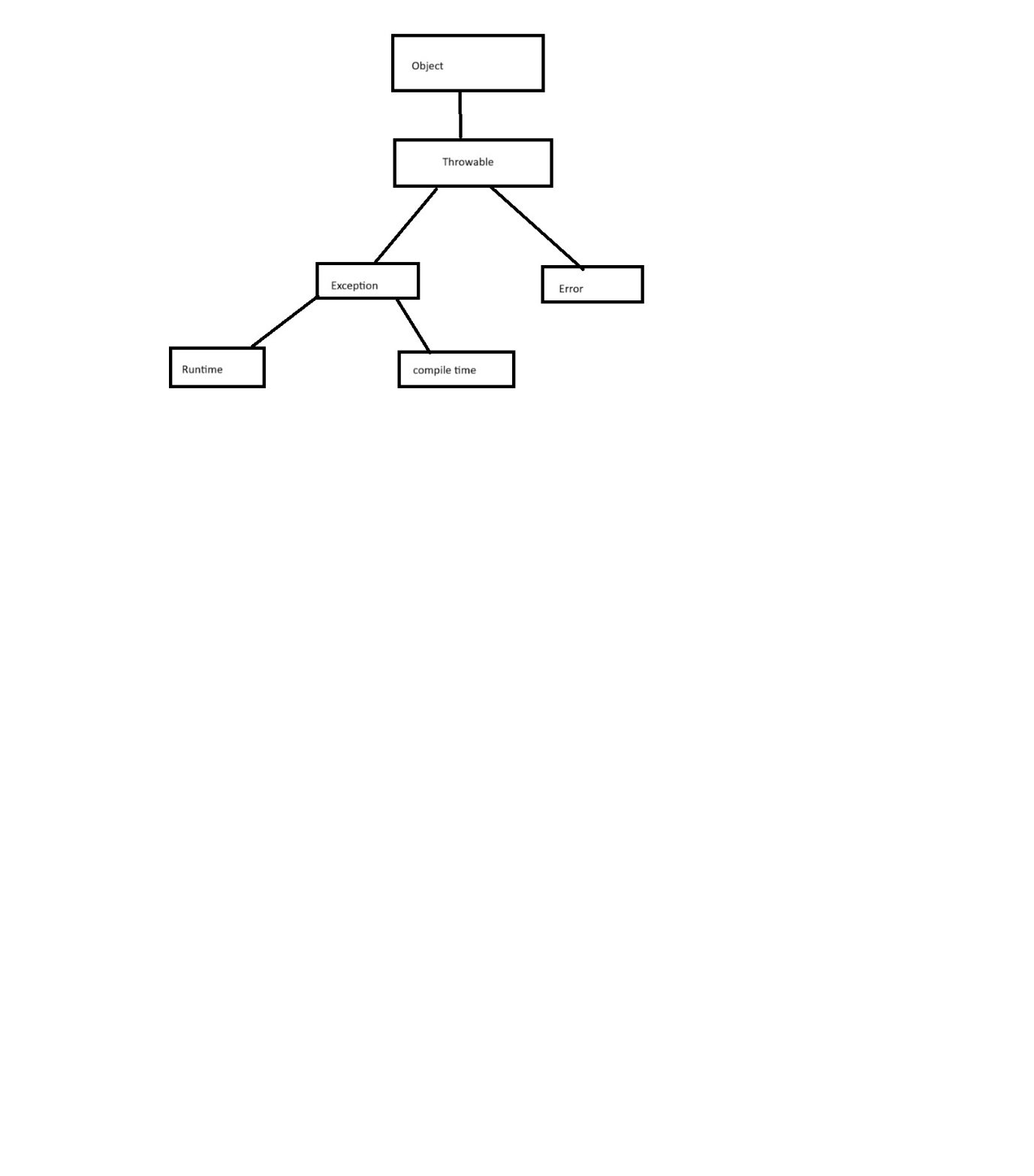
****

**Exception Handling**

**What is an Error?**

An Error indicates a serious problem that a reasonable application should not try to catch. We have the following two types of errors:

1. **Compile Time Error**
2. **Run Time Error**

**Compile Time Error**

Errors that occur at the time of compilation of the program are called compile-time errors. Compile-time errors occurred because if we don’t follow the java syntaxes properly, java programming rules properly, etc. Compile-time errors are identified by the java compiler. So in simple words, we can say that compile-time errors occur due to a poor understanding of the programming language. These errors can be identified by the programmer and can be rectified before the execution of the program only. So these errors do not cause any harm to the program execution.

**Run Time Error**

Errors that occur at the time of execution in the program are called runtime errors. Run-Time Errors are also called Exceptions. Exceptions may occur because programmer logic fails or JVM fails. Exceptions are identified by JVM.

**Note:** The Runtime errors are dangerous because whenever they occur in the program, the program terminates abnormally on the same line where the error gets occurred without executing the next line of code.

**What is an Exception?**

Exceptions are the run-time errors that occur during the execution of the program. The exception will cause the abnormal termination of the program execution. An Exception is an unwanted event that interrupts the normal flow of the program. When an exception occurs program execution gets terminated. It is an object which is thrown at runtime.

**Why an exception occurs?**

These errors occurred when we enter the wrong data into a variable, try to open a file for which there is no permission, try to connect to the database with the wrong user id and password, the wrong implementation of logic, missing required resources, etc. There can be several reasons that can cause a program to throw an exception. For example: Opening a non-existing file in your program, Network connection problem, bad input data provided by the user, etc.

**What happens when an exception is raised in the program?**

Program execution is terminated abnormally. It means statements placed after exception-causing statements are not executed but the statements placed before that exception-causing statements are executed by JVM.

**What JVM does do when a logical mistake occurred in the program?**

It creates an exception class object that is associated with that logical mistake and terminates the current method execution by throwing this exception object by using the “throw” keyword. So we can say an exception is an event that occurs during the execution of a program that disrupts the normal flow of instruction execution.

**Example: The below example shows program execution without exception**

**public** **class** ExceptionHandlingDemo

**{**

**public** **static** **void** main **(String[]**args**)**

**{**

**int** a = 20;

**int** b = 10;

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

System.out.println **(**"b value = " + b**)**;

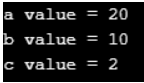
**int** c = a / b;

System.out.println **(**"c value = " + c**)**;

**}**

**}**

**Output:**



**Example: The following example shows program execution with the exception**

**public** **class** ExceptionHandlingDemo

**{**

**public** **static** **void** main **(String[]**args**)**

**{**

**int** a = 20;

**int** b = 0;

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

System.out.println **(**"b value = " + b**)**;

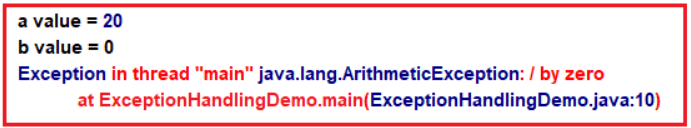
**int** c = a / b;

System.out.println **(**"c value = " + c**)**;

**}**

**}**

**Output:**



**Explanation of the above Program:**

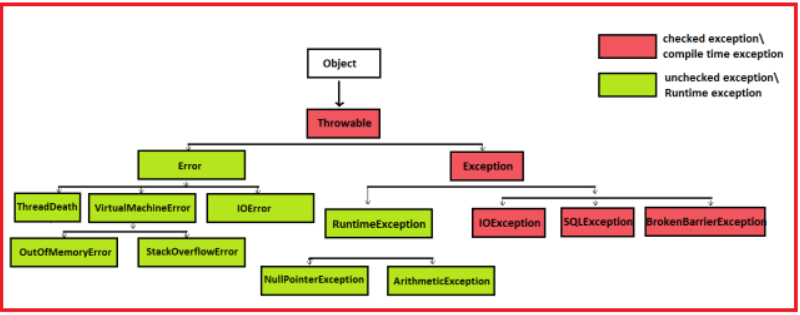
After printing the value of a and b, JVM terminates this program execution by throwing ArithmeticException because the logical mistake we committed is dividing integer numbers by integer zero. As we know it is not possible to divide an integer number by zero. But it is possible to divide a number with double zero (0.0).

From the above program, we can define the exception technically as

1. An exception is an event because when an exception is raised JVM internally executes some logic to prepare that exception-related messages.
2. The exception is a signal because by looking into the exception message developer will take necessary actions against that exception.
3. An exception is an object because for throwing an exception, JVM or we should create an appropriate class object.

**Exception Hierarchy in Java**

The **java.lang.Throwable** class is the root class of the Java Exception hierarchy which is inherited by two subclasses: Exception and Error. All exception and error types are subclasses of class Throwable, which is the base class of the 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, Error is used by the Java run-time system([JVM](https://www.geeksforgeeks.org/jvm-works-jvm-architecture/)) to indicate errors having to do with the run-time environment itself(JRE). StackOverflowError is an example of such an error. A hierarchy of Java Exception classes is given below:



Here,

**Object:** Object is the super most class of all classes available in java.

**Throwable:** For all types of exceptions java.lang.Throwable is the superclass. It has two main subclasses

1. **Error**
2. **Exception**

**Exception:** Exception is the super most class of all exceptions that may occur because of programmer logic failure. These exceptions we can handle.

**Error:** Error is the super-most class of all exceptions that occur because of JVM failure. These errors we cannot handle.

**What are the differences between Error and Exception?**

An Exception is an exception that can be handled. It means when an exception happens the programmer can do something to avoid any harm. But an Error is an exception that cannot be handled. It means it happens and the programmer cannot do anything. Let’s see in detail

**Difference 1:**

Error type exception is thrown due to the problem that occurred inside JVM logic, like If there is no memory in the java stack area to create a new Stackframe to execute method then the JVM process is killed by throwing Error type exception “java.lang.StackOverfolwError”. If there is no memory in the heap area to create a new object then the JVM process is killed by throwing the Error type exception “**java.lang.OutOfMemoryError**”.

Exception type exceptions are thrown due to the problem that occurred in java program logic, like If we divide an integer number with zero, then JVM terminates program execution by throwing Exception type exception “java.lang.ArithmeticException”. If we pass the array size as a negative number, then JVM terminates the program execution by throwing the exception type exception “java.lang.NegativeArraySizeException”.

**Difference 2:**

We cannot catch an Error type exception because an error type exception is not thrown in our application and once this error type exception is thrown JVM is terminated.

We can catch an Exception type exception because an exception type exception is thrown in our program and moreover JVM is not directly terminated because of an exception type exception. JVM is only terminated if the thrown exception is not caught.

**Types of Exceptions in Java:**

We have the following two types of Exceptions:

1. **Checked Exceptions**
2. **Unchecked Exceptions**

**Checked Exceptions in Java:**

Exceptions that are identified at compilation time and occurred at runtime are called checked exceptions. These checked exceptions are also called Compile Time Exceptions. An exception said to be checked exception whose exception handling is mandatory as per the compiler. Example: **IOException, ClassNotFoundException, CloneNotSupportedException, etc.**

**Unchecked Exceptions in Java**

Exceptions that are identified and occurred at run-time are called Unchecked Exceptions. These Unchecked Exceptions are also called Runtime Exceptions. An exception is said to be an unchecked exception whose exception handling is optional as per the compiler. Example: **Arithmetic Exception, NumberFormatException, NoSuchMethodError, etc.**

**Note:**All child classes of Error and Runtime Exception classes are called the unchecked exception and the remaining classes are called checked exceptions.

**Example of getting clarity about the exception:**

Check the below program, in this program, there is a chance of raising three exception

**public** **class** Division

**{**

**public** **static** **void** main **(String[]**args**)**

**{**

**int** a = **Integer**.parseInt **(**args**[**0**])**;

**int** b = **Integer**.parseInt **(**args**[**1**])**;

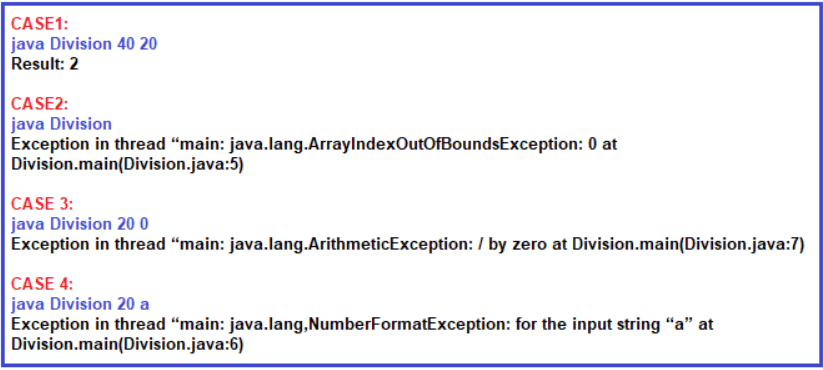
**int** c = a / b;

System.out.println **(**"Result : " + c**)**;

**}**

**}**

**Output:**



**Is the above exception messages user-understandable?**

Definitely, no, users cannot understand the above exception messages because they are java based on exception messages. So the user cannot take further decisions alone to resolve the above problem. Developers should guide to solve the above problem.

**What is the solution to the above problem?**

It is the developer’s responsibility to convert java exception messages into user-understandable message format. To solve this problem developer should write exception handling code in the java program. Using exception handling code, developers can catch the exception and can print and pass user understandable messages.

**What is exception handling in Java?**

The process of catching the exception for converting JVM given exception message to end-user understandable message or for stopping the abnormal termination of the program is called exception handling.

The process of handling these run-time errors or exceptions is called Exception Handling. Once we handle the exception in a program we will be getting the following advantages

1. We can stop the abnormal termination
2. We can perform any corrective action that may resolve the problem occurring due to abnormal termination.
3. Displaying a user-friendly error message, so that the client can resolve the problem provided if it is under his control.

**Why do we need Exception Handling in Java?**

In projects, the exception is handled

1. To stop the abnormal termination of the program
2. To provide user understandable messages when an exception is raised. So that users can take decisions without the developer’s help.

Basically by implementing Exception handling, we are providing life to a program to talk to the user on behalf of the developer.

**What is the procedure to Handle Exception in Java?**

The Exception Handling in Java is a 4 steps procedure

1. Preparing the exception object appropriate to the current logical mistake.
2. Throwing that exception to the appropriate exception handler.
3. Catching that exception
4. Taking necessary actions against that exception

**How can we handle an exception in Java?**

There are two ways to handle the exception in Java.

1. **Logical implementation**
2. **Try catch implementation**

**What is logical implementation?**

In this method, we handle the exception by using logical statements. In real-time programming, the first and foremost importance always given to logical implementation only. If it is not possible to handle the exception using logical implementation then we only need to go for try-catch implementation.

**import** *java.io.*\*;

**public** **class** Main

**{**

**public** **static** **void** main **(String[]**args**)** **throws** IOException

**{**

BufferedReader br = new BufferedReader**(**new InputStreamReader**(**System.in**))**;

**int** a, b, c;

System.out.println**(**"Enter Any 2 Numbers"**)**;

a = **Integer**.parseInt**(**br.readLine**())**;

b = **Integer**.parseInt**(**br.readLine**())**;

**if** **(**b == 0**)**

**{**

System.out.println**(**"second number should not be zero"**)**;

**}**

**else**

**{**

c = a / b;

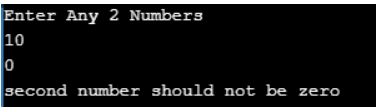
System.out.println**(**"C VALUE = " + c**)**;

**}**

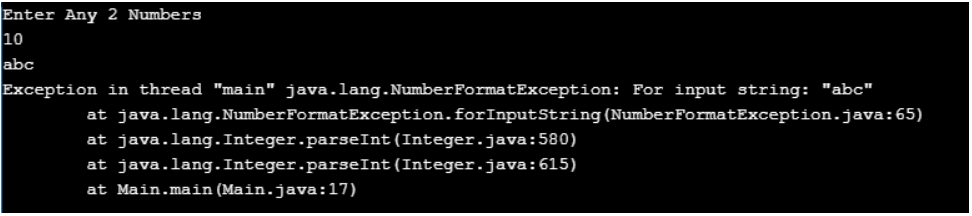
**}**

**}**

**Output:**



In the above example when we entered the second number is zero, then the exception will be raised and that is handled using logical implementation. But while we are entering two numbers instead of the number if we entered any character then it will give you one exception which is FormatException which is not handled in this program as shown in the below image.



Whenever an exception has occurred within the particular method then the corresponding exception object is created by the particular method and given to JVM, then JVM will search for exception handling code within the particular method if not available it will search in its caller method even there also it is not available, then finally JVM will call “Default Exception Handler” Program which will display the Exception object and terminate the method from java stack (abnormal termination). In order to handle such a type of exception, we need to go for Try catch implementation.

**Exception handling in Java using the Try Catch implementation with Examples**

In Java, Exception Handling can be done by using five Java keywords:

1. **Try**
2. **Catch**
3. **Finally**
4. **Throw**
5. **throws**

Let us first understand how try-catch works and then we will proceed with the rest three keywords.

**Try block:**

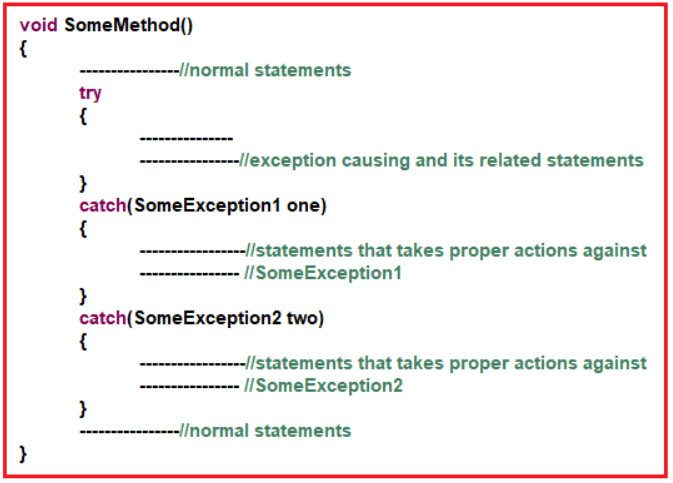
try keyword establishes a block in which we need to write the exception causing and its related statements. That means exception-causing statements must be placed in the try block so that we can handle and catch that exception for stopping abnormal termination and to display end-user understandable messages.

**Catch block:**

The catch block is used to catch the exception that is thrown from its corresponding try block. It has logic to take necessary on that caught exception. Catch block syntax is look like a constructor. It does not take accessibility modifier, normal modifier, or return type.

It takes a single parameter of type Throwable or its subclasses. Throwable is the superclass of all exception sub-classes. Inside the catch block, we can write any statement which is legal in java including raising an exception. A try block can be followed by multiple catch blocks.

**Syntax to use Try Catch in Java:**



**Exception Handling Example in Java**

Below program shows handling and catching exceptions to print user understandable messages relevant to the thrown exception.

**import** *java.io.*\*;

**public** **class** Main

**{**

**public** **static** **void** main **(String[]**args**)** **throws** IOException

**{**

**try**

**{**

BufferedReader br = new BufferedReader **(**new InputStreamReader **(**System.in**))**;

**int** a, b, c;

System.out.println **(**"Enter Any 2 Numbers"**)**;

a = **Integer**.parseInt **(**br.readLine **())**;

b = **Integer**.parseInt **(**br.readLine **())**;

c = a / b;

System.out.println **(**"C VALUE = " + c**)**;

**}**

**catch** **(**ArithmeticException ae**)**

**{**

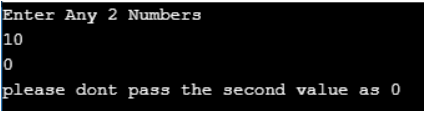
System.out.println **(**"please dont pass the second value as 0"**)**;

**}**

**}**

**}**

**Output:**



**Rules in using try Catch in java**

1. **Rule1:** try must follow either zero or n number of catch blocks or 1 finally block else it leads to Compilation Error: “try without a catch or finally”
2. **Rule2:**catch must be placed immediately after try block else it leads to Compilation Error: “catch without try:”
3. **Rule3**: finally must be placed either immediately after try or after try/catch else it leads to Compilation Error: “finally without try”
4. **Rule4:** the catch block parameter must be of type java.lang.Throwable or its sub-classes else it leads to Compilation Error: “incompatible types”
5. **Rule5:** try/catch/finally blocks are not allowed at the class level directly because logic is not allowed at the class level directly.

**Multiple catch blocks in Java:**

It is possible in Java to write multiple catch blocks for a single try block. Check the below program.

**import** *java.io.*\*;

**public** **class** Main

**{**

**public** **static** **void** main **(String[]**args**)** **throws** IOException

**{**

**try**

**{**

BufferedReader br = new BufferedReader **(**new InputStreamReader **(**System.in**))**;

**int** a, b, c;

System.out.println **(**"Enter Any 2 Numbers"**)**;

a = **Integer**.parseInt **(**br.readLine **())**;

b = **Integer**.parseInt **(**br.readLine **())**;

c = a / b;

System.out.println **(**"C VALUE = " + c**)**;

**}**

**catch** **(**NumberFormatException nfe**)**

**{**

System.out.println **(**"please pass only integer values"**)**;

**}**

**catch** **(**ArithmeticException ae**)**

**{**

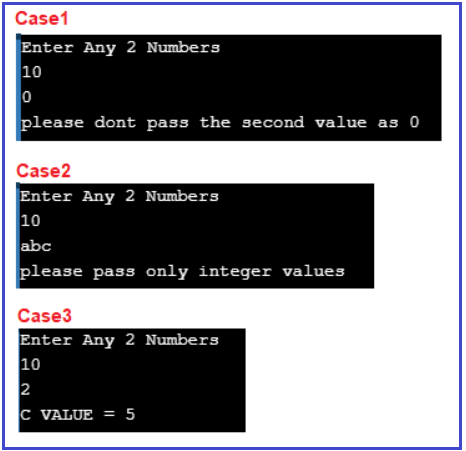
System.out.println **(**"please dont pass the second value as 0"**)**;

**}**

**}**

**}**

**Output:**



**Note:** After the try block, we can write multiple catch blocks to catch every exception thrown from its corresponding try block.

**Can we catch all exceptions using a single catch block?**

Yes, we can catch all exceptions with a single catch block with the parameter “**java.lang.Exception**” We should use this catch block only for stopping abnormal termination irrespective of the exceptions thrown from its corresponding try block. An example is given below.

**import** *java.io.*\*;

**public** **class** Main

**{**

**public** **static** **void** main **(String[]**args**)** **throws** IOException

**{**

**try**

**{**

BufferedReader br = new BufferedReader **(**new InputStreamReader **(**System.in**))**;

**int** a, b, c;

System.out.println **(**"Enter Any 2 Numbers"**)**;

a = **Integer**.parseInt **(**br.readLine **())**;

b = **Integer**.parseInt **(**br.readLine **())**;

c = a / b;

System.out.println **(**"C VALUE = " + c**)**;

**}**

**catch** **(**Exception ex**)**

**{**

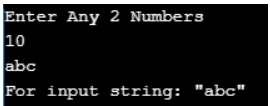
System.out.println **(**ex.getMessage**())**;

**}**

**}**

**}**

**Output:**



It is always recommended to write catch blocks with exception parameters even though we are writing multiple catch blocks. It acts as a backup catch block.

**When should we write multiple catch blocks for a single try block?**

We should write multiple catch blocks for a single try block because of the following reasons

1. To print a message specific to an exception or
2. To execute some logic specific to an exception

In our multiple catch block programs, we have placed catch blocks with different exception classes to print messages relevant to the caught exception.

**Rules for Writing Multiple catch block in java:**

1. **Rule1**: catch block should not be duplicated.
2. **Rule2**: superclass parameter catch block should not be placed before the child class parameter catch block. Violation of any of the above rules leads to CE: “**exception has already been caught**”
3. **Note**: we write multiple catch blocks not only for printing exception-specific messages but also for executing some logic specific to the exception raised in the corresponding try statement.

**Example: using multiple catch blocks along with generic catch block**

**import** *java.io.*\*;

**public** **class** Main

**{**

**public** **static** **void** main **(String[]**args**)** **throws** IOException

**{**

**try**

**{**

BufferedReader br = new BufferedReader **(**new InputStreamReader **(**System.in**))**;

**int** a, b, c;

System.out.println **(**"Enter Any 2 Numbers"**)**;

a = **Integer**.parseInt **(**br.readLine **())**;

b = **Integer**.parseInt **(**br.readLine **())**;

c = a / b;

System.out.println **(**"C VALUE = " + c**)**;

**}**

**catch** **(**NumberFormatException nfe**)**

**{**

System.out.println **(**"please pass only integer values"**)**;

**}**

**catch** **(**ArithmeticException ae**)**

**{**

System.out.println **(**"please dont pass the second value as 0"**)**;

**}**

**catch** **(**Exception ex**)**

**{**

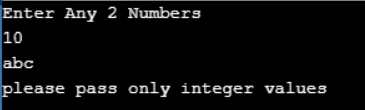
System.out.println **(**"Generic Exception"**)**;

**}**

**}**

**}**

**Output:**



**How to display Exception or Runtime Error Message?**

The throwable class has three methods to print the exception messages. These methods are useful when we write catch blocks with a superclass as a parameter. JVM printed Exception message format is shown below.



**Using printStackTrace():**

If we use the printstackTrace() method it will display the Exception information in detail like reason, Exception class name, program name where it has occurred, in what method, and in what line.

Using printStackTrace()

**Using toString():**

If we use the toString() method it will display the reason why the Exception has occurred and the Exception class name.



**Using getMessage():**

If we use the getMessage() method it will display only the reason why the Exception has occurred.

Using getMessage()

**Below example shows calling the above three methods:**

**public** **class** Main

**{**

**public** **static** **void** main **(String[]**args**)**

**{**

**try**

**{**

System.out.println **(**10 / 0**)**;

**}**

**catch** **(**ArithmeticException ae**)**

**{**

System.out.println **(**"getMessage method output"**)**;

System.out.println **(**ae.getMessage **())**;

System.out.println **(**"toString method output"**)**;

System.out.println **(**ae.toString **())**;

System.out.println **(**"printStackTrace() method output"**)**;

ae.printStackTrace **()**;

System.out.println **(**"JVM default output"**)**;

**throw** ae;

//by using above statement we are just re-throwing the caught exception this

//exception is caught by JVM default handler and prints full exception messages

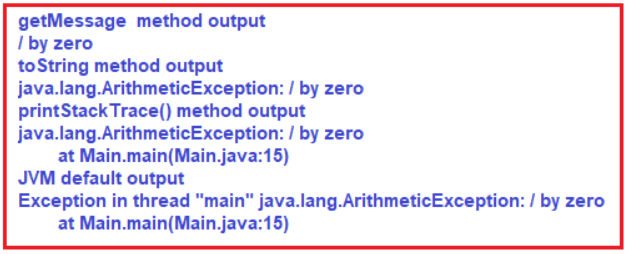
//along with thread name

**}**

**}**

**}**

**Output:**



**What is finally block in Java?**

Finally establishes a block that definitely executes statements placed in it. Statements that are placed in finally block are always executed irrespective of the way the control is coming out from the try block either by completing normally or throwing exceptions by catching or not catching.

**Why we need finally block in real-time applications?**

As per the coding standard in the finally block we should write resource releasing logic or clean up code.

Resource releasing logic means unreferenced objects that are created in the try block. For example, in real-time projects, we create JDBC objects in the try block and at the end of the try block, we must close these objects. Since the statements written in try and catch block are not guaranteed to be executed we must place them in the finally block.

For example, if we want to close JDBC objects such as Connection object, ResultSet object, etc we must call the close() method in both try as well as in catch block to guarantee its execution. Instead of placing the same close() method call statements in multiple places if we write it in the finally block it is always executed irrespective of the exception raised or not raised.

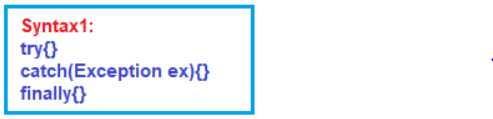
**Syntax to use Finally block in Java:**

We can use finally in two ways

1. **try/catch/finally**
2. **try/finally**

**Try/Catch/Finally:**

We use this syntax to catch the exception and also to execute some statements definitely



**Try/Finally:**

We use this syntax if we don’t want to catch the exception but if we want to execute some statements definitely. In this case, program execution is terminated abnormally.



**Example: try/catch/finally without exception**

In the below example, we have implemented the try, catch and finally block. We also place one statement after the finally block. When we execute the below example, first it will execute the try block. As there is no exception in the try block, so the catch block will not be executed, and then finally block will execute. Once the finally block executed, then the statement present after the finally block gets executed. Run the program and see the output.

**public** **class** Main

**{**

**public** **static** **void** main **(String[]**args**)**

**{**

**try**

**{**

System.out.println **(**"in try"**)**;

**}**

**catch** **(**ArithmeticException ae**)**

**{**

System.out.println **(**"in catch"**)**;

**}**

**finally**

**{**

System.out.println **(**"in finally"**)**;

**}**

System.out.println **(**"after try catch finally"**)**;

**}**

**}**

**Output:**  
**in try**  
**in finally**  
**after try catch finally**

**Example: try/catch/finally with exception and catch block is matched**

In the below example, we have implemented try, catch and finally block. We also placed one statement after the finally block. When we execute the below code, it will execute the try block first. And in the try block, one exception has occurred i.e. Arithmetic Exception. As soon as the exception occurred, it will search for a catch block to handle that exception, and here, the catch block will be executed. Once the catch block completes its execution, then the finally block will execute. Once the finally block executed, then the statement present after the finally block gets executed. Run the program and see the output for better understanding.

**public** **class** Main

**{**

**public** **static** **void** main **(String[]**args**)**

**{**

**try**

**{**

System.out.println **(**"in try"**)**;

System.out.println **(**10 / 0**)**;

**}**

**catch** **(**ArithmeticException ae**)**

**{**

System.out.println **(**"in catch"**)**;

**}**

**finally**

**{**

System.out.println **(**"in finally"**)**;

**}**

System.out.println **(**"after try catch finally"**)**;

**}**

**}**

**Output:**  
**in try**  
**in catch**  
**in finally**  
**after try catch finally**

**Example: try/catch/finally with exception and catch block is not matched**

In the below example, we have implemented try, catch and finally block. When we execute the following example, it will start execution from the try block. And in the try block, one exception has occurred i.e. Arithmetic Exception. As soon as the exception occurred, the controller will search for a catch block to handle that exception. But there is no catch block to handle the Arithmetic Exception. So, it will then execute the finally block and throw the exception.  Here, the statement present after the finally will not be executed. Run the application and see the output for better understanding.

**public** **class** Main

**{**

**public** **static** **void** main **(String[]**args**)**

**{**

**try**

**{**

System.out.println **(**"in try"**)**;

System.out.println **(**10 / 0**)**;

**}**

**catch** **(**NullPointerException npe**)**

**{**

System.out.println **(**"in catch"**)**;

**}**

**finally**

**{**

System.out.println **(**"in finally"**)**;

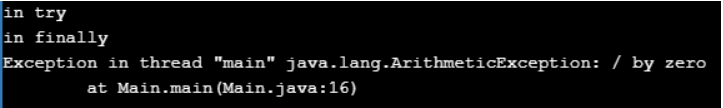
**}**

System.out.println **(**"after try cat finally"**)**;

**}**

**}**

**Output:**



**Example: try/finally without exception**

In the below example, we have implemented try and finally block. When we execute the following example. first, it will execute the try block and then it will execute the finally block. Once the finally block executed, then it will execute the statement present after the finally block. Run the application and see the output for better understanding.

**public** **class** Main

**{**

**public** **static** **void** main **(String[]**args**)**

**{**

**try**

**{**

System.out.println **(**"in try"**)**;

**}**

**finally**

**{**

System.out.println **(**"in finally"**)**;

**}**

System.out.println **(**"after try catch finally"**)**;

**}**

**}**

**Output:**  
**in try**  
**in finally**  
**after try catch finally**

**Example: try/finally with exception**

In the below example, we have implemented try and finally block. When we execute the following example. first, it will execute the try block and in the try block exception has been occurred. And there is no catch block at all. So, it will then execute the finally block and throw an exception. In this case, it will not execute the statement present after the finally block. Run the program and see the output for better understanding.

**public** **class** Main

**{**

**public** **static** **void** main **(String[]**args**)**

**{**

**try**

**{**

System.out.println **(**"in try"**)**;

System.out.println **(**10 / 0**)**;

**}**

**finally**

**{**

System.out.println **(**"in finally"**)**;

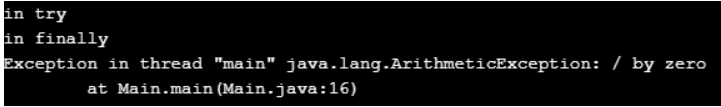
**}**

System.out.println **(**"after try cat finally"**)**;

**}**

**}**

**Output:**



**Rules for writing try-catch-finally Block**

1. Try must be followed by either catch block or finally block or both.
2. The catch must be followed by either catch block or finally block
3. The catch must precede by either catch block or try block
4. Finally must precede by either catch block or try block
5. We can write only one finally statement and try block and many catch blocks in the try-catch-finally chain.
6. Nesting try-catch-finally is also possible.
7. When we write multiple catch blocks if Exception is not having any Is-A relation then we can write catch block in any order otherwise we must write catch block in order like first child class and follow by the parent class.
8. We cannot write two catch blocks that are going to catch the same Exceptions.

**How JVM execute try-catch-finally block in java?**

First, it will execute the statements written before try-catch blocks next JVM will enter into try block and if no exception occurred then it will execute all the statements of the try block and it will skip all the catch block and control will be given after try-catch blocks.

JVM will enter into try block and if an exception occurred then it will skip the remaining statements of try block and control will be given to any one of the matching catch blocks and executes the statements available in the catch block and the next control will be given after try-catch blocks.

**Note:**

1. When an Exception occurred then control will be given to any of the matching catch blocks and the remaining catch blocks will be skipped and control will be given after try-catch blocks.
2. If no catch block is matching then JVM will call or invoke “Default Exception Handler” and which always causes abnormal termination.
3. If the Exception occurred or not in both the cases finally block will be executed.

**What is an unreachable statement in Java?**

If we place the return statement in the finally block and if we can place statement after finally block it leads to CE: “**unreachable statements**”. Let us see a program for better understanding

**public** **class** Main

**{**

**public** **static** **void** main **(String[]**args**)**

**{**

System.out.println **(**m1 **())**;

**}**

**static** **int** m1 **()**

**{**

**try**

**{**

System.out.println **(**"in try"**)**;

**return** 10;

**}**

**catch** **(**ArithmeticException ae**)**

**{**

System.out.println **(**"in catch"**)**;

**return** 20;

**}**

**finally**

**{**

System.out.println **(**"in finally"**)**;

**return** 30;

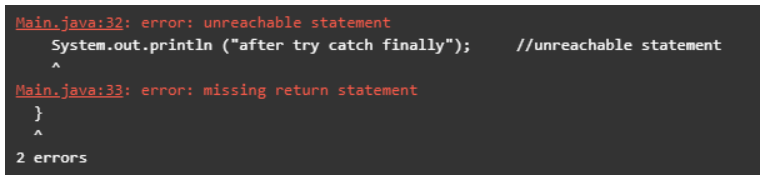
**}**

System.out.println **(**"after try catch finally"**)**; //unreachable statement

**}**

**}**

**Output:**



**Difference Between Final, Finally and Finalize**

|  |  |  |
| --- | --- | --- |
| **Final** | **finally** | **Finalize** |
| The Final is used to apply restrictions on class, method, and variable. The final class can’t be inherited, the final method can’t be overridden and the final variable value can’t be changed. | Finally is used to place important code, it will be executed whether an exception is handled or not. | Finalize is used to perform clean-up processing just before the object is garbage collected. |
| The Final is a keyword. | Finally is a block. | Finalize is a method. |

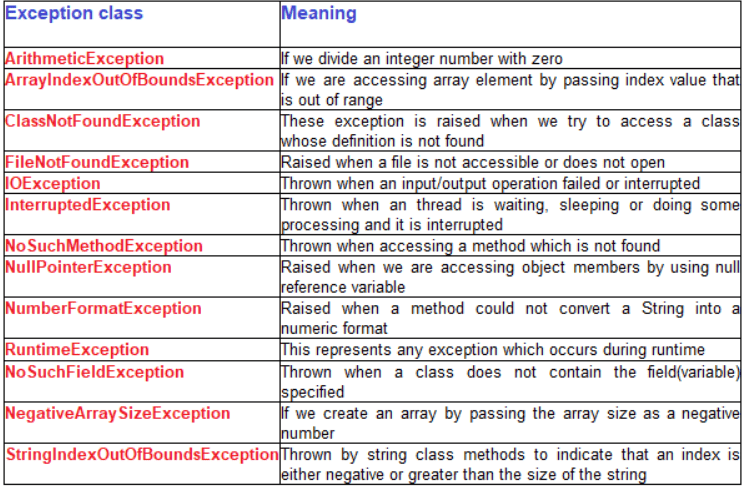
**Types of Exceptions in Java:**

There are two types of exceptions available in java

1. **Built-in Exceptions**
2. **User-Defined or Custom Exceptions**

**Built-in exceptions in java:**

Built-in exceptions are the exceptions that are all ready available in java. These exceptions are suitable to explain certain error situations. Some built-in exceptions are given below.



**User-defined or Custom Exception in Java:**

The new exception class developed by a developer is called the custom exception or user-defined exception. These classes must be a subclass of either Throwable or any one of its subclasses.

Most of the cases in the project’s custom exceptions classes are derived from the Exception class. All exception classes defined by SUN are based on the java language and API requirements. So according to the business requirement developer must write their own exception class

**How to Create a User-Defined or Custom Exception class in Java Application?**

It is a two-step process

1. Define a packaged public class deriving from java.lang.Exception
2. Define public no-arg and String parameter constructor with super() call.

**Note:** No-arg constructor for creating exception objects without a message and parameterized constructor for creating an exception object with a message

**Why did you extend the custom exception class from java.lang.Exception class?**

We created custom exception classes for throwing these exceptions when a condition is failed and that exception handling wants to be validated by the compiler so they must be extended from java.lang.Exception.

We must extend them from **java.lang.RuntimeException** class if we don’t want to validate these exception handling by the compiler. Since we don’t write exception classes for handling errors in JVM internal logic we don’t derive them from java.lang.Error

Also, we don’t derive them from **java.lang.Throwable** as it created a new category of exceptions and it is not recommended to create a new category of exceptions because this custom exception is not caught by the catch (Exception e). In projects, developers write catch(Exception e) to catch all types of exceptions and in this case, our exception is not caught

**Program for User-Defined Exception in Java**

**class** MinBalanceException **extends** Exception

**{**

**public** MinBalanceException **()**

**{**

System.out.println **(**"Balance is low"**)**;

**}**

**}**

**public** **class** Main

**{**

**public** **static** **void** main **(String[]**args**)**

**{**

**try**

**{**

**int** acc**[]** = **{** 100, 101, 102, 103, 104, 105 **}**; // input can be got from runtime too

**double** balance**[]** = **{** 900, 2000, 1500, 1560, 1765.50 **}**;

System.out.println **(**"Account No\t" + "Balance\t"**)**;

**for** **(int** i = 0; i **<** 5; i++**)**

**{**

System.out.println **(**acc**[**i**]** + "\t\t" + balance**[**i**]** + "\t"**)**;

**if** **(**balance**[**i**]** **<** 1000**)**

**{**

**throw** new MinBalanceException **()**; //throwing user defined exception

**}**

**}**

**}**

**catch** **(**MinBalanceException e**)**

**{**

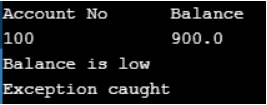
System.out.println **(**"Exception caught"**)**;

**}**

**}**

**}**

**Output:**



**One Real-Time Example of Custom or User-Defined Exception in Java Application:**

Define custom exceptions **InvalidAmountException**, **InsufficientFundsException** to handle wrong operations done by customers in deposit, and withdrawal operations.

**TestCases:**

1. Throw InvalidAmountException if the user enters zero or –ve amount in deposit and withdraw operations.
2. Throw InsufficientFundsException if the user enters the amount greater than the balance in case of withdrawing operations.

**InvalidAmountException.java**

**package** *com.pkr.exceptions*;

**public** **class** InvalidAmountException **extends** Exception

**{**

**public** InvalidAmountException **()**

**{**

super **()**;

**}**

**public** InvalidAmountException **(String** msg**)**

**{**

super **(**msg**)**;

**}**

**}**

**InsufficientFundsException.java**

**package** *com.pkr.exceptions*;

**public** **class** InsufficientFundsException **extends** Exception

**{**

**public** InsufficientFundsException **()**

**{**

super **()**;

**}**

**public** InsufficientFundsException **(String** msg**)**

**{**

super **(**msg**)**;

**}**

**}**

**Bank.java**

**package** *com.pkr.blogic*;

**import** *com.pkr.exceptions.*\*;

**public** **interface** Bank

**{**

**public** **void** deposite **(double** amount**)** **throws** InvalidAmountException;

**public** **double** withdraw **(double** amount**)** **throws** InsufficientFundsException;

**public** **void** balanceEnquiry **()**;

**}**

**HDFCBank.java**

**import** *com.pkr.exceptions.InsufficientFundsException*;

**import** *com.pkr.exceptions.InvalidAmountException*;

**public** **class** HDFCBank **implements** Bank

**{**

**private** **double** balance;

**public** **void** deposite **(double** amount**)** **throws** InvalidAmountException

**{**

**if** **(**amount **<**= 0**)**

**{**

**throw** new InvalidAmountException **(**amount + "is not valid"**)**;

**}**

balance = balance + amount;

**}**

**public** **double** withdraw **(double** amount**)** **throws** InsufficientFundsException

**{**

**if** **(**balance **<** amount**)**

**{**

**throw** new InsufficientFundsException **(**"insufficient funds"**)**;

**}**

balance = balance - amount;

**return** amount;

**}**

**public** **void** balanceEnquiry **()**

**{**

System.out.println **(**"current balance = " + balance**)**;

**}**

**}**

**Cleark.java**

**package** *com.pkr.user*;

**import** *com.pkr.blogic.*\*;

**import** *com.pkr.exceptions.*\*;

**import** *java.util.*\*;

**public** **class** Clerk

**{**

**public** **static** **void** main **(String[]**args**)**

**{**

**try**

**{**

Scanner sc = new Scanner **(**System.in**)**;

Bank acc1 = new HDFCBank **()**;

**String** option = "";

**do**

**{**

System.out.println **(**"1. DEPOSITE"**)**;

System.out.println **(**"2. WITHDRAW"**)**;

System.out.println **(**"3. BALANCE ENQUIRY"**)**;

System.out.println **(**"ENTER OPTION"**)**;

option = sc.next **()**;

**switch** **(**option**)**

**{**

**case** "1":

**{**

System.out.println **(**"ENTER DEPOSIT AMOUNT"**)**;

**double** amt = sc.nextDouble **()**;

acc1.deposite **(**amt**)**;

acc1.balanceEnquiry **()**;

**break**;

**}**

**case** "2":

**{**

System.out.println **(**"ENTER WITHDRAW AMOUNT"**)**;

**double** amt = sc.nextDouble **()**;

**double** wd = acc1.withdraw **(**amt**)**;

System.out.println **(**"WITHDRAW AMOUNT IS :" + wd**)**;

acc1.balanceEnquiry **()**;

**break**;

**}**

**case** "3":

**{**

acc1.balanceEnquiry **()**;

**break**;

**}**

**default**:

System.out.println **(**"INVALID OPTION"**)**;

**}**

System.out.println **(**"DO YOU WANT TO CONTINUE (YES/NO)"**)**;

option = sc.next **()**;

**}**

**while(**option.equalsIgnoreCase **(**"YES"**))**;

**}**

**catch** **(**InvalidAmountException iae**)**

**{**

System.out.println **(**iae.getMessage **())**;

**}**

**catch** **(**InsufficientFundsException ife**)**

**{**

System.out.println **(**ife.getMessage **())**;

**}**

**catch** **(**NumberFormatException nfe**)**

**{**

System.out.println **(**nfe.getMessage **())**;

**}**

**}**

**}**

**Advantages of Exception Handling in Java**

1. **Separating Error-Handling Code from Regular Code**: Exceptions provide the means to separate the details of what to do when something out of the ordinary happens from the main logic of a program.
2. **Propagating Errors Up the Call Stack:** Ability to propagate error reporting up the call stack of methods.
3. **Grouping and Differentiating Error Types:** Because all exceptions thrown within a program are objects, the grouping or categorizing of exceptions is a natural outcome of the class hierarchy.

**throw keyword in Java**

By default, all predefined exceptions are created and thrown implicitly and identified by JVM. But if we want to throw the exceptions explicitly then we have to use the throw keyword.

**Syntax: throw exception;**

The “throw” keyword is used to throw an exception manually. In most cases, we use it for throwing checked exceptions explicitly. The “throw” keyword must follow the Throwable type of object and It must be used in method logic. Since it is a transfer statement, we cannot place a statement after the throw keyword. It leads to a compile-time error “**Unreachable statement**”.

**For example:**

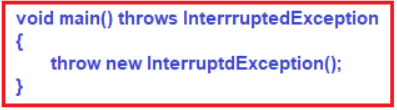


ArithmeticException is a runtime exception so the compiler does not check its Exception Handling.



InterruptedException is a direct subclass of the Exception class which means it is a checked exception. So we must catch or report this exception using the throws keyword. As we have not done either of both compiler throws CE: “**Unreported exception InterruptedException must be caught or declared to be thrown**”.

The below code shows the correct syntax for throwing a checked exception



**Sample Program for Java Throw Keyword**

**public** **class** Main

**{**

**static** **void** validate **(int** age**)**

**{**

**if** **(**age **<** 18**)**

**throw** new ArithmeticException **(**"not valid"**)**;

**else**

System.out.println **(**"welcome to vote"**)**;

**}**

**public** **static** **void** main **(String** args**[])**

**{**

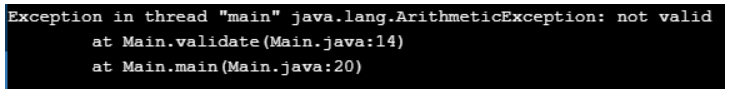
validate **(**13**)**;

System.out.println **(**"rest of the code..."**)**;

**}**

**}**

**Output:**

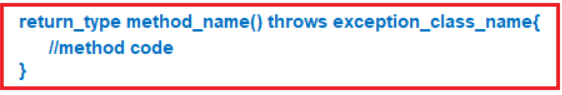


**throws keyword in Java**

The “throws” keyword is used to report that raised exception to the caller. It is mandatory for checked exceptions for reporting if they are not handled.

This keyword is used to transfer the responsibility of Exception handling to its caller method. 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 a method signature.

**Syntax:**



**What is the need for having throws keyword when you can handle exceptions using try-catch?**

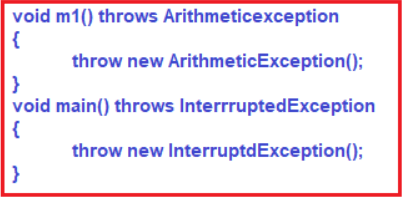
Suppose you have several such methods that can cause exceptions, in that case, it would be tedious to write this try-catch for each method. The code will become unnecessarily long and will be less readable.

One way to overcome this problem is by using throws like this: declare the exceptions in the method signature using throws and handle the exceptions where you are calling this method by using try-catch.

Another advantage of using this approach is that you will be forced to handle the exception when you call this method, all the exceptions that are declared using throws, must be handled where you are calling this method else you will get a compilation error.

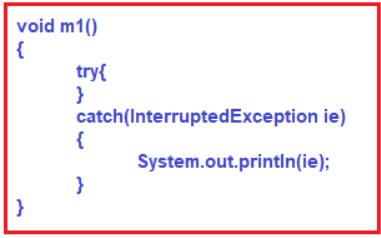
**Examples to understand java throws keywords:**

The “throws” keyword must throw a Throwable type of class name. It must be used in the method prototype after the method parenthesis.

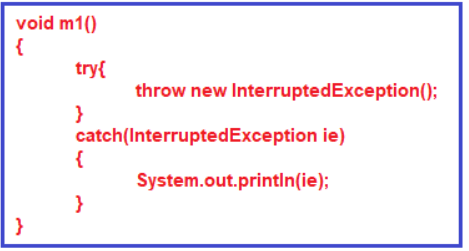


We are not allowed to write a catch block with checked exceptions without throwing it from the try block. It leads to CE: “**exception never thrown from the corresponding try statement**”.

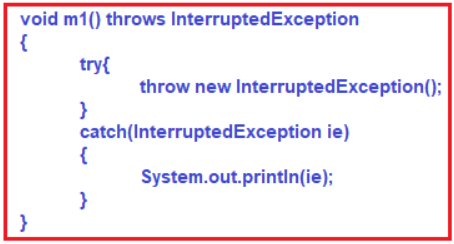
**In below program catch block leads to the above exception**



**The below program compiles fine**



**We can catch and also report using throws**



**Sample Program using throws Keyword in Java**

**import** *java.io.IOException*;

**public** **class** Main

**{**

**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**[])**

**{**

Main obj = new Main**()**;

obj.p **()**;

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

**}**

**}**

**Output:**  
**exception handled**  
**normal flow…**

**Note:**It is always recommended to use try-catch-finally blocks to handle the exceptions. But it is not recommended to transfer the exception handling job to its caller method by using the throws keyword which always leads to abnormal termination.

**Difference Between Throw and Throws keywords in java**

The “throws” clause is used when the programmer does not want to handle the exception and throw it out of a method whereas the “throw” clause is used when the programmer wants to throw an exception explicitly and wants to handle it using a catch block. Hence throws and throw are contradictory.

|  |  |
| --- | --- |
| **throw** | **throws** |
| Java throw keyword is used to explicitly throw an exception | Java throws keyword is used to declare an exception |
| A checked exception cannot be propagated using throw only | A checked exception can be propagated using throws |
| The throw is followed by an instance | Throws is followed by a class |
| The throw is used within the method | Throws is used with the method signature |
| You cannot throw multiple exceptions | Using throws you can declare multiple exceptions |