Exception Handling in Java

1) Error

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

There are three kinds of errors: syntax errors, runtime errors, and logic errors.

**Syntax errors/compile time error**

These are errors where the compiler finds something wrong with your program, and you can't even try to execute it. For example, you may have incorrect punctuation, or may be trying to use a variable that hasn't been declared.

Syntax errors are the easiest to find and correct. The compiler will tell you where it got into trouble, and its best guess as to what you did wrong. Usually the error is on the exact line indicated by the compiler, or the line just before it; however, if the problem is incorrectly nested braces, the actual error may be at the beginning of the nested block.

**public class threadeg1 {**

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

**{**

**System.out.println("Hello")**

**System.out.println("Hi")**

**}**

**}**

**Output :**

**$javac threadeg1.java**  
threadeg1.java:6: error: ';' expected

System.out.println("Hello")

^

threadeg1.java:7: error: ';' expected

System.out.println("Hi")

^

2 errors

**Runtime errors**

If there are no syntax errors, Java may detect an error while your program is running. You will get an error message telling you the kind of error, and a **stack trace** that tells not only where the error occurred, but also what other [method](https://www.cis.upenn.edu/~matuszek/General/JavaSyntax/methods.html) or methods you were in. For example,

**public class threadeg1 {**

**public static void main()**

**{**

**System.out.println("Hello");**

**System.out.println("Hi");**

**}**

**}**

**output**

Error: Main method not found in class threadeg1, please define the main method as:

public static void main(String[] args)

or a JavaFX application class must extend javafx.application.Application

**//code compiles but JVM cannot run it…..**

Runtime errors are intermediate in difficulty. Java tells you where it discovered that your program had gone wrong, but you need to trace back from there to figure out where the problem originated.

**Logical errors**

A logic error, or **bug**, is when your program compiles and runs, but does the wrong thing. The Java system, of course, has no idea what your program is *supposed* to do, so it provides no additional information to help you find the error.

Ways to track down a logic error include:

* Think about what the program must have done in order to produce the results it did. This will lead you to where the error must have occurred.
* Put in [print statements](https://www.cis.upenn.edu/~matuszek/General/JavaSyntax/print-statements.html) to help you figure out what the program is actually doing.
* Use a **debugger** to step through your program and watch what it does.

**public** **class** threadeg1 {

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

{

Double sal= 5000.00;

sal = sal\*15/100;

System.***out***.println("salary =” + sal);

}

}

Output : 750 .00

What is Exception in Java

**Dictionary Meaning:** Exception is an abnormal condition/runtime error .

In Java, an exception is an event that disrupts the normal flow of the program. **It is an object which is thrown at runtime.**

An exception can occur for many different reasons. Following are some scenarios where an exception occurs.

* A user has entered an 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.

Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.

Types of Java Exceptions

There are mainly two types of exceptions: checked and unchecked. Here, an error is considered as the unchecked exception.

1. Checked Exception
2. Unchecked Exception

Difference between Checked and Unchecked Exceptions

1) Checked Exception

**a.** A checked exception is an exception that is checked (notified) by the compiler at compilation-time, these are also called as **compile time exceptions**. These exceptions cannot simply be ignored, the programmer should take care of (handle) these exceptions.

**b.** For example, if you use **FileReader** class in your program to read data from a file, if the file specified in its constructor doesn't exist, then a *FileNotFoundException* occurs, and the compiler prompts the programmer to handle the exception.

### Example

import java.io.File;

import java.io.FileReader;

public class FilenotFound\_Demo {

public static void main(String args[]) {

File file = new File("E://file.txt");

FileReader fr = new FileReader(file);

}

}

If you try to compile the above program, you will get the following exceptions.

### Output

C:\>javac FilenotFound\_Demo.java

FilenotFound\_Demo.java:8: error: unreported exception FileNotFoundException; must be caught or declared to be thrown

FileReader fr = new FileReader(file);

^

1 error

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

2) Unchecked Exception

**a.** An unchecked exception is an exception that occurs at the time of execution. These are also called as **Runtime Exceptions**. These include programming bugs, such as logic errors or improper use of an API. Runtime exceptions are ignored at the time of compilation.

**b.** For example, if you have declared an array of size 5 in your program, and trying to call the 6th element of the array then an *ArrayIndexOutOfBoundsExceptionexception* occurs.

### Example

public class Unchecked\_Demo {

public static void main(String args[]) {

int num[] = {1, 2, 3, 4};

System.out.println(num[5]);

}

}

If you compile and execute the above program, you will get the following exception.

### Output

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5

at Exceptions.Unchecked\_Demo.main(Unchecked\_Demo.java:8)

**c.**

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

What is Exception Handling

**Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.**

Advantage of Exception Handling

The core advantage of exception handling is **to maintain the normal flow of the application**. An exception normally disrupts the normal flow of the application that is why we use exception handling. Let's take 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 your program and there occurs an exception at statement 5, the rest of the code will not be executed i.e. statement 6 to 10 will not be executed. If we perform exception handling, the rest of the statement will be executed. That is why we use exception handling in [Java](https://www.javatpoint.com/java-tutorial).

Hierarchy of Java Exception classes

The **java.lang.Throwable** class is the root class of Java Exception hierarchy which is inherited by two subclasses: **Exception and Error**. A hierarchy of Java Exception classes are given below:



throwable :

1. It is a class that represents all errors and exceptions which may occur in java
2. Exception is the super class of all exceptions in java
3. Exception can be handled by programmer but error which cannot be , never be handled …

Java Exception Handling Example

**//without try and catch**

public class JavaExceptionExample{

public static void main(String args[]){

//code that may raise exception

int data=100/0;

//rest code of the program

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

}

}

Let's see an example of Java Exception Handling where we using a try-catch statement to handle the exception.

**public** **class** JavaExceptionExample{

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

**try**{

//code that may raise exception

**int** data=100/0;

}**catch**(ArithmeticException e){System.out.println(e);} //to understand problem

//rest code of the program

finally {

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

}

}

Output:

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

**rest of the code...**

**In the above example, 100/0 raises an ArithmeticException which is handled by a try-catch block.**

Common Scenarios of Java Exceptions

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

**1) A scenario where ArithmeticException occurs**

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

**int** a=50/0;//ArithmeticException

**2) A scenario where NullPointerException occurs**

If we have a null value in any [variable](https://www.javatpoint.com/java-variables), performing any operation on the variable throws a NullPointerException.

String s=**null**;

System.out.println(s.length());//NullPointerException

**3) A scenario where NumberFormatException occurs**

The wrong formatting of any value may occur NumberFormatException. Suppose I have a [string](https://www.javatpoint.com/java-string) variable that has characters, converting this variable into digit will occur NumberFormatException.

1. String s="abc";
2. **int** i=Integer.parseInt(s);//NumberFormatException

**4) A scenario where ArrayIndexOutOfBoundsException occurs**

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

**int** a[]=**new** **int**[5];

a[10]=50; //ArrayIndexOutOfBoundsException

Java try-catch block

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 of try block, the rest of the block code will not execute. So, it is recommended not to keeping 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

**try**{

//code that may throw an exception

}**catch**(Exception\_class\_Name ref){}

Syntax of try-finally block

**try**{

//code that may throw an exception

}**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.**

Problem without exception handling

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

Example 1

**public** **class** TryCatchExample1 {

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

**int** data=50/0; //may throw exception

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

    }

}

**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 can be 100 lines of code after exception. So all the code after exception will not be executed.

Solution by exception handling

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

Example 2

**public** **class** TryCatchExample2 {

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

**try**

        {

**int** data=50/0; //may throw exception

        }

            //handling the exception

**catch**(ArithmeticException e)

        {

            System.out.println(e);

        }

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

    }

}

**[Test it Now](http://www.javatpoint.com/opr/test.jsp?filename=TryCatchExample2" \t "_blank)**

**Output:**

java.lang.ArithmeticException: / by zero

rest of the code

Now, 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.

**public** **class** TryCatchExample3 {

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

**try**

        {

**int** data=50/0; //may throw exception

                         // if exception occurs, the remaining statement will not exceute

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

        }

             // handling the exception

**catch**(ArithmeticException e)

        {

            System.out.println(e);

        }

    }

}

**[Test it Now](http://www.javatpoint.com/opr/test.jsp?filename=TryCatchExample3" \t "_blank)**

**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.

Example 4

Here, we handle the exception using the parent class exception.

**public** **class** TryCatchExample4 {

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

**try**

        {

**int** data=50/0; //may throw exception

        }

            // handling the exception by using Exception class

**catch**(Exception e)

        {

            System.out.println(e);

        }

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

    }

}

**[Test it Now](http://www.javatpoint.com/opr/test.jsp?filename=TryCatchExample4" \t "_blank)**

**Output:**

java.lang.ArithmeticException: / by zero

rest of the code

Example 5

Let's see an example to print a custom message on exception.

**public** **class** TryCatchExample5 {

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

**try**

        {

**int** data=50/0; //may throw exception

        }

             // handling the exception

**catch**(Exception e)

        {

                  // displaying the custom message

            System.out.println("Can't divided by zero");

        }

    }

}

**[Test it Now](http://www.javatpoint.com/opr/test.jsp?filename=TryCatchExample5" \t "_blank)**

**Output:**

Can't divided by zero

Example 6

Let's see an example to resolve the exception in a catch block.

**public** **class** TryCatchExample6 {

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

**int** i=50;

**int** j=0;

**int** data;

**try**

        {

        data=i/j; //may throw exception

        }

            // handling the exception

**catch**(Exception e)

        {

             // resolving the exception in catch block

            System.out.println(i/(j+2));

        }

    }

}

**[Test it Now](http://www.javatpoint.com/opr/test.jsp?filename=TryCatchExample6" \t "_blank)**

**Output:**

25

Example 7

In this example, along with try block, we also enclose exception code in a catch block.

**public** **class** TryCatchExample7 {

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

**try**

        {

**int** data1=50/0; //may throw exception

        }

             // handling the exception

**catch**(Exception e)

        {

            // generating the exception in catch block

**int** data2=50/0; //may throw exception

        }

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

    }

}

**[Test it Now](http://www.javatpoint.com/opr/test.jsp?filename=TryCatchExample7" \t "_blank)**

**Output:**

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

**Here, we can see that the catch block didn't contain the exception code. So, enclose exception code within a try block and use catch block only to handle the exceptions.**

Example 8

In this example, we handle the generated exception (Arithmetic Exception) with a different type of exception class (ArrayIndexOutOfBoundsException).

**public** **class** TryCatchExample8 {

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

**try**

        {

**int** data=50/0; //may throw exception

        }

            // try to handle the ArithmeticException using ArrayIndexOutOfBoundsException

**catch**(ArrayIndexOutOfBoundsException e)

        {

            System.out.println(e);

        }

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

    }

}

**[Test it Now](http://www.javatpoint.com/opr/test.jsp?filename=TryCatchExample8" \t "_blank)**

**Output:**

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

Example 9

Let's see an example to handle another unchecked exception.

**public** **class** TryCatchExample9 {

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

**try**

        {

**int** arr[]= {1,3,5,7};

        System.out.println(arr[10]); //may throw exception

        }

            // handling the array exception

**catch**(ArrayIndexOutOfBoundsException e)

        {

            System.out.println(e);

        }

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

    }

}

**[Test it Now](http://www.javatpoint.com/opr/test.jsp?filename=TryCatchExample9" \t "_blank)**

**Output:**

java.lang.ArrayIndexOutOfBoundsException: 10

rest of the code

Example 10

Let's see an example to handle checked exception.

**import** java.io.FileNotFoundException;

**import** java.io.PrintWriter;

**public** **class** TryCatchExample10 {

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

        PrintWriter pw;

**try** {

            pw = **new** PrintWriter("jtp.txt"); //may throw exception

            pw.println("saved");

        }

// providing the checked exception handler

**catch** (FileNotFoundException e) {

            System.out.println(e);

        }

    System.out.println("File saved successfully");

    }

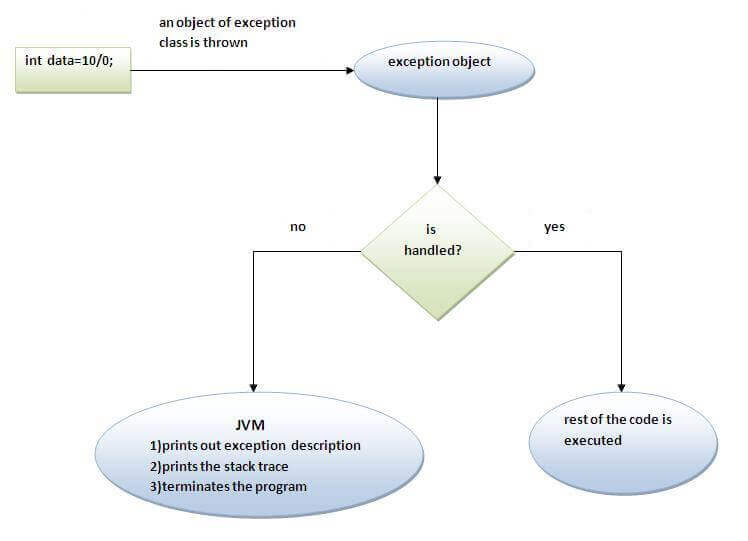
}

**[Test it Now](http://www.javatpoint.com/opr/test.jsp?filename=TryCatchExample10" \t "_blank)**

**Output:**

File saved successfully

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 exception is handled by the application programmer, normal flow of the application is maintained i.e. rest of the code is executed.

Note the following −

* 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.

Difference between throw and throws in Java

There are many differences between throw and throws keywords. A list of differences between throw and throws are given below:

|  |  |  |
| --- | --- | --- |
| **No.** | **throw** | **throws** |
| 1) | Java throw keyword is used to explicitly throw an exception. | Java throws keyword is used to declare an exception. |
| 2) | Checked exception cannot be propagated using throw only. | Checked exception can be propagated with throws. |
| 3) | Throw is followed by an instance. | Throws is followed by class. |
| 4) | Throw is used within the method. | Throws is used with the method signature. |
| 5) | You cannot throw multiple exceptions. | You can declare multiple exceptions e.g. public void method()throws IOException,SQLException. |

**Throw clause**

In Java we have already defined exception classes such as ArithmeticException, NullPointerException, ArrayIndexOutOfBounds exception etc. These exceptions are set to trigger on different-2 conditions. For example when we divide a number by zero, this triggers ArithmeticException, when we try to access the array element out of its bounds then we get ArrayIndexOutOfBoundsException.

**We can define our own set of conditions or rules and throw an exception explicitly using throw keyword.** For example, we can throw ArithmeticException when we divide number by 5, or any other numbers, what we need to do is just set the condition and throw any exception using throw keyword.

**Syntax of throw keyword:**

throw new exception\_class("error message");

**Throw Example**

public class Example1{

void checkAge(int age){

if(age<18)

throw new ArithmeticException("Not Eligible for voting");

else

System.out.println("Eligible for voting");

}

public static void main(String args[]){

Example1 obj = new Example1();

obj.checkAge(13);

System.out.println("End Of Program");

}

}

Output:

Exception in thread "main" java.lang.ArithmeticException:

Not Eligible for voting

at Example1.checkAge(Example1.java:4)

at Example1.main(Example1.java:10)

**Throws**

**Throws keyword** is used for handling checked exceptions. By using throws we can declare multiple exceptions in one go.

## What is the need of having throws keyword when you can handle exception using try-catch?

Well, that’s a valid question. We already know we can [handle exceptions](https://beginnersbook.com/2013/04/java-exception-handling/) using [try-catch block](https://beginnersbook.com/2013/04/try-catch-in-java/).  
The throws does the same thing that try-catch does but there are some cases where you would prefer throws over try-catch. For example:  
Lets say we have a method myMethod() that has statements that can throw either ArithmeticException or NullPointerException, in this case you can use try-catch as shown below:

public void myMethod()

{

try {

// Statements that might throw an exception

}

catch (ArithmeticException e) {

// Exception handling statements

}

catch (NullPointerException e) {

// Exception handling statements

}

}

But suppose you have several such methods that can cause exceptions, in that case it would be tedious to write these try-catch for each method. The code will become unnecessary 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 compilation error.

public void myMethod() throws ArithmeticException, NullPointerException

{

// Statements that might throw an exception

}

public static void main(String args[]) {

try {

myMethod();

}

catch (ArithmeticException e) {

// Exception handling statements

}

catch (NullPointerException e) {

// Exception handling statements

}

}

**Example of throws Keyword**

In this example the method myMethod() is throwing two **checked exceptions** so we have declared these exceptions in the method signature using **throws** Keyword. If we do not declare these exceptions then the program will throw a compilation error.

import java.io.\*;

class ThrowExample {

void myMethod(int num)throws IOException, ClassNotFoundException{

if(num==1)

throw new IOException("IOException Occurred");

else

throw new ClassNotFoundException("ClassNotFoundException");

}

}

public class Example1{

public static void main(String args[]){

try{

ThrowExample obj=new ThrowExample();

obj.myMethod(1);

}catch(Exception ex){

System.out.println(ex);

}

}

}

Output:

java.io.IOException: IOException Occurred

eg

public class Example1{

int division(int a, int b) throws ArithmeticException{

int t = a/b;

return t;

}

public static void main(String args[]){

Example1 obj = new Example1();

try{

System.out.println(obj.division(15,0));

}

catch(ArithmeticException e){

System.out.println("You shouldn't divide number by zero");

}

}

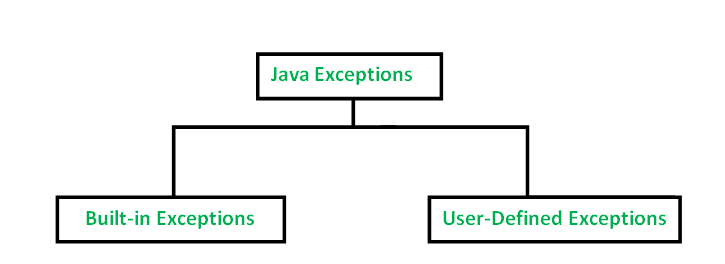
}

**Output:**

You shouldn't divide number by zero

Types of Exception in Java with Examples

Java defines several types of exceptions that relate to its various class libraries. Java also allows users to define their own exceptions.



**Built-in exceptions** are the exceptions which are available in Java libraries. These exceptions are suitable to explain certain error situations. Below is the list of important built-in exceptions in Java.

1. **ArithmeticException**  
   It is thrown when an exceptional condition has occurred in an arithmetic operation.
2. **ArrayIndexOutOfBoundsException**It is thrown to indicate that an array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the array.
3. **ClassNotFoundException**This Exception is raised when we try to access a class whose definition is not found
4. **FileNotFoundException**This Exception is raised when a file is not accessible or does not open.
5. **IOException**It is thrown when an input-output operation failed or interrupted
6. **InterruptedException**It is thrown when a thread is waiting , sleeping , or doing some processing , and it is interrupted.
7. **NoSuchFieldException**It is thrown when a class does not contain the field (or variable) specified
8. **NoSuchMethodException**It is thrown when accessing a method which is not found.
9. **NullPointerException**This exception is raised when referring to the members of a null object. Null represents nothing
10. **NumberFormatException**This exception is raised when a method could not convert a string into a numeric format.
11. **RuntimeException**This represents any exception which occurs during runtime.
12. **StringIndexOutOfBoundsException**It is thrown by String class methods to indicate that an index is either negative than the size of the string

**User-Defined Exceptions**

Sometimes, the built-in exceptions in Java are not able to describe a certain situation. In such cases, user can also create exceptions which are called ‘user-defined Exceptions’.  
Following steps are followed for the creation of user-defined Exception.

* The user should create an exception class as a subclass of Exception class. Since all the exceptions are subclasses of Exception class, the user should also make his class a subclass of it. This is done as:

class MyException extends Exception

* We can write a default constructor in his own exception class.

MyException(){}

* We can also create a parameterized constructor with a string as a parameter.  
  We can use this to store exception details. We can call super class(Exception) constructor from this and send the string there.

MyException(String str)

{

super(str);

}

* To raise exception of user-defined type, we need to create an object to his exception class and throw it using throw clause, as:
  + MyException me = new MyException(“Exception details”);

throw me;

* The following program illustrates how to create own exception class MyException.
* Details of account numbers, customer names, and balance amounts are taken in the form of three arrays.
* In main() method, the details are displayed using a for-loop. At this time, check is done if in any account the balance amount is less than the minimum balance amount to be ept in the account.
* If it is so, then MyException is raised and a message is displayed “Balance amount is less”.

|  |
| --- |
| // Java program to demonstrate user defined exception    // This program throws an exception whenever balance  // amount is below Rs 1000  public class MyException extends Exception  {      //store account information      private static int accno[] = {1001, 1002, 1003, 1004};        private static String name[] =                   {"Nish", "Shubh", "Sush", "Abhi", "Akash"};        private static double bal[] =           {10000.00, 12000.00, 5600.0, 999.00, 1100.55};        // default constructor      MyException() {    }        // parametrized constructor      MyException(String str) { super(str); }        // write main()      public static void main(String[] args)      {          try  {              // display the heading for the table              System.out.println("ACCNO" + "\t" + "CUSTOMER" +                                             "\t" + "BALANCE");                // display the actual account information              for (int i = 0; i < 5 ; i++)              {                  System.out.println(accno[i] + "\t" + name[i] +                                                 "\t" + bal[i]);                    // display own exception if balance < 1000                  if (bal[i] < 1000)                  {                      MyException me =                         new MyException("Balance is less than 1000");                      throw me;                  }              }          } //end of try            catch (MyException e) {              e.printStackTrace();          }      }  } |

* RunTime Error
* MyException: Balance is less than 1000
* at MyException.main(fileProperty.java:36)
* **Output:**
* ACCNO CUSTOMER BALANCE
* 1001 Nish 10000.0
* 1002 Shubh 12000.0
* 1003 Sush 5600.0
* 1004 Abhi 999.0