## What is an exception?

An Exception is an unwanted event that interrupts the normal flow of the program. When an exception occurs program execution gets terminated. In such cases we get a system generated error message. The good thing about exceptions is that they can be handled in Java. By handling the exceptions we can provide a meaningful message to the user about the issue rather than a system generated message, which may not be understandable to a user.

### Why an exception occurs?

There can be several reasons that can cause a program to throw exception. For example: Opening a non-existing file in your program, Network connection problem, bad input data provided by user etc.

## Exception Handling

If an exception occurs, which has not been handled by programmer then program execution gets terminated and a system generated error message is shown to the user. For example look at the system generated exception below:  
An exception generated by the system is given below

Exception in thread "main" java.lang.ArithmeticException: / by zero at ExceptionDemo.main(ExceptionDemo.java:5)

ExceptionDemo : The class name

main : The method name

ExceptionDemo.java : The filename

java:5 : Line number

This message is not user friendly so a user will not be able to understand what went wrong. In order to let them know the reason in simple language, we handle exceptions. We handle such conditions and then prints a user friendly warning message to user, which lets them correct the error as most of the time exception occurs due to bad data provided by user.

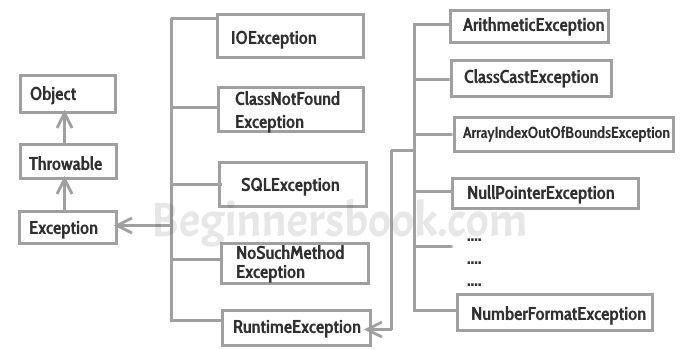
### Advantage of exception handling

Exception handling ensures that the flow of the program doesn’t break when an exception occurs. For example, if a program has bunch of statements and an exception occurs mid way after executing certain statements then the statements after the exception will not execute and the program will terminate abruptly.  
By handling we make sure that all the statements execute and the flow of program doesn’t break.

## Difference between error and exception

Errors indicate that something severe enough has gone wrong, the application should crash rather than try to handle the error.

Exceptions are events that occurs in the code. A programmer can handle such conditions and take necessary corrective actions. Few examples:  
NullPointerException – When you try to use a reference that points to null.  
ArithmeticException – When bad data is provided by user, for example, when you try to divide a number by zero this exception occurs because dividing a number by zero is undefined.  
ArrayIndexOutOfBoundsException – When you try to access the elements of an array out of its bounds, for example array size is 5 (which means it has five elements) and you are trying to access the 10th element.



## Types of exceptions

There are two types of exceptions in Java:  
1)Checked exceptions  
2)Unchecked exceptions

### Checked exceptions

All exceptions other than Runtime Exceptions are known as Checked exceptions as the compiler checks them during compilation to see whether the programmer has handled them or not. If these exceptions are not handled/declared in the program, you will get compilation error. For example, SQLException, IOException, ClassNotFoundException etc.

### Unchecked Exceptions

Runtime Exceptions are also known as Unchecked Exceptions. These exceptions are not checked at compile-time so compiler does not check whether the programmer has handled them or not but it’s the responsibility of the programmer to handle these exceptions and provide a safe exit. For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc.

**Compiler will never force you to catch such exception or force you to declare it in the method using throws keyword.**

There are two types of exceptions: checked exception and unchecked exception. In this guide, we will discuss them. The main difference between checked and unchecked exception is that the checked exceptions are checked at compile-time while unchecked exceptions are checked at runtime.

## What are checked exceptions?

Checked exceptions are checked at compile-time. It means if a method is throwing a checked exception then it should handle the exception using try-catch block or it should declare the exception using throws keyword, otherwise the program will give a compilation error.

Lets understand this with the help of an example:

### Checked Exception Example

In this example we are reading the file myfile.txt and displaying its content on the screen. In this program there are three places where a checked exception is thrown as mentioned in the comments below. FileInputStream which is used for specifying the file path and name, throws FileNotFoundException. The read() method which reads the file content throws IOException and the close() method which closes the file input stream also throws IOException.

import java.io.\*;

class Example {

public static void main(String args[])

{

FileInputStream fis = null;

/\*This constructor FileInputStream(File filename)

\* throws FileNotFoundException which is a checked

\* exception

\*/

fis = new FileInputStream("B:/myfile.txt");

int k;

/\* Method read() of FileInputStream class also throws

\* a checked exception: IOException

\*/

while(( k = fis.read() ) != -1)

{

System.out.print((char)k);

}

/\*The method close() closes the file input stream

\* It throws IOException\*/

fis.close();

}

}

Output:

Exception in thread "main" java.lang.Error: Unresolved compilation problems:

Unhandled exception type FileNotFoundException

Unhandled exception type IOException

Unhandled exception type IOException

Why this compilation error? As I mentioned in the beginning that checked exceptions gets checked during compile time. Since we didn’t handled/declared the exceptions, our program gave the compilation error.  
How to resolve the error? There are two ways to avoid this error. We will see both the ways one by one.

Method 1: Declare the exception using throws keyword.  
As we know that all three occurrences of checked exceptions are inside main() method so one way to avoid the compilation error is: Declare the exception in the method using throws keyword. You may be thinking that our code is throwing FileNotFoundException and IOException both then why we are declaring the IOException alone. The reason is that IOException is a parent class of FileNotFoundException so it by default covers that. If you want you can declare them like this public static void main(String args[]) throws IOException, FileNotFoundException.

import java.io.\*;

class Example {

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

{

FileInputStream fis = null;

fis = new FileInputStream("B:/myfile.txt");

int k;

while(( k = fis.read() ) != -1)

{

System.out.print((char)k);

}

fis.close();

}

}

Output:  
File content is displayed on the screen.

Method 2: Handle them using try-catch blocks.  
The approach we have used above is not good at all. It is not the best exception handling practice. You should give meaningful message for each exception type so that it would be easy for someone to understand the error. The code should be like this:

import java.io.\*;

class Example {

public static void main(String args[])

{

FileInputStream fis = null;

try{

fis = new FileInputStream("B:/myfile.txt");

}catch(FileNotFoundException fnfe){

System.out.println("The specified file is not " +

"present at the given path");

}

int k;

try{

while(( k = fis.read() ) != -1)

{

System.out.print((char)k);

}

fis.close();

}catch(IOException ioe){

System.out.println("I/O error occurred: "+ioe);

}

}

}

This code will run fine and will display the file content.

Here are the few other Checked Exceptions –

* SQLException
* IOException
* ClassNotFoundException
* InvocationTargetException

## What are Unchecked exceptions?

Unchecked exceptions are not checked at compile time. It means if your program is throwing an unchecked exception and even if you didn’t handle/declare that exception, the program won’t give a compilation error. Most of the times these exception occurs due to the bad data provided by user during the user-program interaction. It is up to the programmer to judge the conditions in advance, that can cause such exceptions and handle them appropriately. All Unchecked exceptions are direct sub classes of RuntimeException class.

Lets understand this with an example:

### Unchecked Exception Example

class Example {

public static void main(String args[])

{

int num1=10;

int num2=0;

/\*Since I'm dividing an integer with 0

\* it should throw ArithmeticException

\*/

int res=num1/num2;

System.out.println(res);

}

}

If you compile this code, it would compile successfully however when you will run it, it would throw ArithmeticException. That clearly shows that unchecked exceptions are not checked at compile-time, they occurs at runtime. Lets see another example.

class Example {

public static void main(String args[])

{

int arr[] ={1,2,3,4,5};

/\* My array has only 5 elements but we are trying to

\* display the value of 8th element. It should throw

\* ArrayIndexOutOfBoundsException

\*/

System.out.println(arr[7]);

}

}

This code would also compile successfully since ArrayIndexOutOfBoundsException is also an unchecked exception.  
Note: It doesn’t mean that compiler is not checking these exceptions so we shouldn’t handle them. In fact we should handle them more carefully. For e.g. In the above example there should be a exception message to user that they are trying to display a value which doesn’t exist in array so that user would be able to correct the issue.

class Example {

public static void main(String args[]) {

try{

int arr[] ={1,2,3,4,5};

System.out.println(arr[7]);

}

catch(ArrayIndexOutOfBoundsException e){

System.out.println("The specified index does not exist " +

"in array. Please correct the error.");

}

}

}

Output:

The specified index does not exist in array. Please correct the error.

Here are the few unchecked exception classes:

* NullPointerException
* ArrayIndexOutOfBoundsException
* ArithmeticException
* IllegalArgumentException
* NumberFormatException

## Try block

The try block contains set of statements where an exception can occur. A try block is always followed by a catch block, which handles the exception that occurs in associated try block. A try block must be followed by catch blocks or finally block or both.

### Syntax of try block

try{

//statements that may cause an exception

}

While writing a program, if you think that certain statements in a program can throw a exception, enclosed them in try block and handle that exception

## Catch block

A catch block is where you handle the exceptions, this block must follow the try block. A single try block can have several catch blocks associated with it. You can catch different exceptions in different catch blocks. When an exception occurs in try block, the corresponding catch block that handles that particular exception executes. For example if an arithmetic exception occurs in try block then the statements enclosed in catch block for arithmetic exception executes.

### Syntax of try catch in java

try

{

//statements that may cause an exception

}

catch (exception(type) e(object))‏

{

//error handling code

}

## Example: try catch block

If an exception occurs in try block then the control of execution is passed to the corresponding catch block. A single try block can have multiple catch blocks associated with it, you should place the catch blocks in such a way that the generic exception handler catch block is at the last(see in the example below).  
The generic exception handler can handle all the exceptions but you should place is at the end, if you place it at the before all the catch blocks then it will display the generic message. You always want to give the user a meaningful message for each type of exception rather then a generic message.

class Example1 {

public static void main(String args[]) {

int num1, num2;

try {

  /\* We suspect that this block of statement can throw

\* exception so we handled it by placing these statements

\* inside try and handled the exception in catch block

\*/

num1 = 0;

num2 = 62 / num1;

System.out.println(num2);

System.out.println("Hey I'm at the end of try block");

}

  catch (ArithmeticException e) {

/\* This block will only execute if any Arithmetic exception

\* occurs in try block

\*/

System.out.println("You should not divide a number by zero");

}

catch (Exception e) {

/\* This is a generic Exception handler which means it can handle

\* all the exceptions. This will execute if the exception is not

\* handled by previous catch blocks.

\*/

System.out.println("Exception occurred");

}

System.out.println("I'm out of try-catch block in Java.");

}

}

Output:

You should not divide a number by zero

I'm out of try-catch block in Java.

## Multiple catch blocks in Java

The example we seen above is having multiple catch blocks, lets see few rules about multiple catch blocks with the help of examples. To read this in detail, see catching multiple exceptions in java.  
1. As I mentioned above, a single try block can have any number of catch blocks.  
2. A generic catch block can handle all the exceptions. Whether it is ArrayIndexOutOfBoundsException or ArithmeticException or NullPointerException or any other type of exception, this handles all of them. To see the examples of NullPointerException and ArrayIndexOutOfBoundsException, refer this article: Exception Handling example programs.

catch(Exception e){

  //This catch block catches all the exceptions

}

If you are wondering why we need other catch handlers when we have a generic that can handle all. This is because in generic exception handler you can display a message but you are not sure for which type of exception it may trigger so it will display the same message for all the exceptions and user may not be able to understand which exception occurred. Thats the reason you should place is at the end of all the specific exception catch blocks

3. If no exception occurs in try block then the catch blocks are completely ignored.  
4. Corresponding catch blocks execute for that specific type of exception:  
catch(ArithmeticException e) is a catch block that can hanlde ArithmeticException  
catch(NullPointerException e) is a catch block that can handle NullPointerException  
5. You can also throw exception, which is an advanced topic and I have covered it in separate tutorials: user defined exception, throws keyword, throw vs throws.

### Example of Multiple catch blocks

class Example2{

public static void main(String args[]){

try{

int a[]=new int[7];

a[4]=30/0;

System.out.println("First print statement in try block");

}

catch(ArithmeticException e){

System.out.println("Warning: ArithmeticException");

}

catch(ArrayIndexOutOfBoundsException e){

System.out.println("Warning: ArrayIndexOutOfBoundsException");

}

catch(Exception e){

System.out.println("Warning: Some Other exception");

}

System.out.println("Out of try-catch block...");

}

}

Output:

Warning: ArithmeticException

Out of try-catch block...

In the above example there are multiple catch blocks and these catch blocks executes sequentially when an exception occurs in try block. Which means if you put the last catch block ( catch(Exception e)) at the first place, just after try block then in case of any exception this block will execute as it can handle all exceptions. This catch block should be placed at the last to avoid such situations.

## Finally block

I have covered this in a separate tutorial here: java finally block. For now you just need to know that this block executes whether an exception occurs or not. You should place those statements in finally blocks, that must execute whether exception occurs or not.

A finally block contains all the crucial statements that must be executed whether exception occurs or not. The statements present in this block will always execute regardless of whether exception occurs in try block or not such as closing a connection, stream etc.

## Syntax of Finally block

try {

//Statements that may cause an exception

}

catch {

//Handling exception

}

finally {

//Statements to be executed

}

## A Simple Example of finally block

Here you can see that the exception occurred in try block which has been handled in catch block, after that finally block got executed.

class Example

{

public static void main(String args[]) {

try{

int num=121/0;

System.out.println(num);

}

catch(ArithmeticException e){

System.out.println("Number should not be divided by zero");

}

/\* Finally block will always execute

\* even if there is no exception in try block

\*/

finally{

System.out.println("This is finally block");

}

System.out.println("Out of try-catch-finally");

}

}

Output:

Number should not be divided by zero

This is finally block

Out of try-catch-finally

## Few Important points regarding finally block

1. A finally block must be associated with a try block, you cannot use finally without a try block. You should place those statements in this block that must be executed always.

2. Finally block is optional, as we have seen in previous tutorials that a try-catch block is sufficient for exception handling, however if you place a finally block then it will always run after the execution of try block.

3. In normal case when there is no exception in try block then the finally block is executed after try block. However if an exception occurs then the catch block is executed before finally block.

4. An exception in the finally block, behaves exactly like any other exception.

5. The statements present in the finally block execute even if the try block contains control transfer statements like return, break or continue.  
Lets see an example to see how finally works when return statement is present in try block:

### Another example of finally block and return statement

You can see that even though we have return statement in the method, the finally block still runs.

class JavaFinally

{

public static void main(String args[])

{

System.out.println(JavaFinally.myMethod());

}

public static int myMethod()

{

try {

return 112;

}

finally {

System.out.println("This is Finally block");

System.out.println("Finally block ran even after return statement");

}

}

}

Output of above program:

This is Finally block

Finally block ran even after return statement

112

To see more examples of finally and return refer: Java finally block and return statement  
.

## Cases when the finally block doesn’t execute

The circumstances that prevent execution of the code in a finally block are:  
– The death of a Thread  
– Using of the System. exit() method.  
– Due to an exception arising in the finally block.

## Finally and Close()

close() statement is used to close all the open streams in a program. Its a good practice to use close() inside finally block. Since finally block executes even if exception occurs so you can be sure that all input and output streams are closed properly regardless of whether the exception occurs or not.

For example:

....

try{

OutputStream osf = new FileOutputStream( "filename" );

OutputStream osb = new BufferedOutputStream(opf);

ObjectOutput op = new ObjectOutputStream(osb);

try{

output.writeObject(writableObject);

}

finally{

op.close();

}

}

catch(IOException e1){

System.out.println(e1);

}

...

## Finally block without catch

A try-finally block is possible without catch block. Which means a try block can be used with finally without having a catch block.

...

InputStream input = null;

try {

input = new FileInputStream("inputfile.txt");

}

finally {

if (input != null) {

try {

in.close();

}catch (IOException exp) {

System.out.println(exp);

}

}

}

...

## Finally block and System.exit()

System.exit() statement behaves differently than return statement. Unlike return statement whenever System.exit() gets called in try block then Finally block doesn’t execute. Here is a code snippet that demonstrate the same:

....

try {

//try block

System.out.println("Inside try block");

System.exit(0)

}

catch (Exception exp) {

System.out.println(exp);

}

finally {

System.out.println("Java finally block");

}

....

In the above example if the System.exit(0) gets called without any exception then finally won’t execute. However if any exception occurs while calling System.exit(0) then finally block will be executed.

## try-catch-finally block

* Either a try statement should be associated with a catch block or with finally.
* Since catch performs exception handling and finally performs the cleanup, the best approach is to use both of them.

Syntax:

try {

//statements that may cause an exception

}

catch (…)‏ {

//error handling code

}

finally {

//statements to be executed

}

### Examples of Try catch finally blocks

Example 1: The following example demonstrate the working of finally block when no exception occurs in try block

class Example1{

public static void main(String args[]){

try{

System.out.println("First statement of try block");

int num=45/3;

System.out.println(num);

}

catch(ArrayIndexOutOfBoundsException e){

System.out.println("ArrayIndexOutOfBoundsException");

}

finally{

System.out.println("finally block");

}

System.out.println("Out of try-catch-finally block");

}

}

Output:

First statement of try block

15

finally block

Out of try-catch-finally block

Example 2: This example shows the working of finally block when an exception occurs in try block but is not handled in the catch block:

class Example2{

public static void main(String args[]){

try{

System.out.println("First statement of try block");

int num=45/0;

System.out.println(num);

}

catch(ArrayIndexOutOfBoundsException e){

System.out.println("ArrayIndexOutOfBoundsException");

}

finally{

System.out.println("finally block");

}

System.out.println("Out of try-catch-finally block");

}

}

Output:

First statement of try block

finally block

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

at beginnersbook.com.Example2.main(Details.java:6)

As you can see that the system generated exception message is shown but before that the finally block successfully executed.

Example 3: When exception occurs in try block and handled properly in catch block

class Example3{

public static void main(String args[]){

try{

System.out.println("First statement of try block");

int num=45/0;

System.out.println(num);

}

catch(ArithmeticException e){

System.out.println("ArithmeticException");

}

finally{

System.out.println("finally block");

}

System.out.println("Out of try-catch-finally block");

}

}

Output:

First statement of try block

ArithmeticException

finally block

Out of try-catch-finally block

**In my last tutorial**, we discussed about finally block, which is used with a try block and always execute whether exception occurs or not. Here we will see few examples to understand the behaviour of finally block when a return statement is encountered in try block.

Lets see this code snippet, What do you think? Will finally would execute even if there is a return statement in try block as well as in catch block?

try {

//try block

...

return success;

}

catch (Exception ex) {

//catch block

.....

return failure;

}

finally {

System.out.println("Inside finally");

}

The answer is yes. finally block will execute. The only case where it will not execute is when it encounters System.exit().

## Finally: Example with return statement

class FinallyDemo

{

public static int myMethod()

{

try {

//try block

return 0;

}

finally {

//finally

System.out.println("Inside Finally block");

}

}

public static void main(String args[])

{

System.out.println(FinallyDemo.myMethod());

}

}

Output:

Inside Finally block

0

## Does finally block Override the values returned by try-catch block?

Yes. Finally block overrides the value returned by try and catch blocks, lets take an example to understand this:

public static int myTestingFuncn(){

try{

....

return 5;

}

finally {

....

return 19;

}

}

This program would return value 19 since the value returned by try has been overridden by finally

**Here we will see few examples** of throws keyword. To understand these programs you should have the knowledge of throws keyword in java.

## Example 1: Exception propagation using throws keyword

As you can see that we have an exception occurred in method1 which has been handled in the chain-calling method method3(). This example shows how exception propagation works.

class Example1{

void method1() throws ArithmeticException{

throw new ArithmeticException("Calculation error");

}

void method2() throws ArithmeticException{

method1();

}

void method3(){

try{

method2();

}

catch(ArithmeticException e){

System.out.println("ArithmeticException handled");

}

}

public static void main(String args[]){

Example1 obj=new Example1();

obj.method3();

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

}

}

Output:

ArithmeticException handled

End Of Program

## Example 2: When you don’t handle exception and instead declare it at all the places

The ideal way to use throws is by declaring the exceptions in method signature and handle the exceptions using try-catch in calling method. Lets see what happens when we declare the exception at both the places, in method signature as well as in calling method.

class ExceptionExample{

void method()throws ArithmeticException{

throw new ArithmeticException("ArithmeticException Occurred");

}

}

class Example1{

public static void main(String args[])throws ArithmeticException{

ExceptionExample obj=new ExceptionExample();

obj.method();

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

}

}

Output:

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

ArithmeticException Occurred

at ExceptionExample.method(Example1.java:4)

at Example1.main(Example1.java:10)

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. Throw keyword can also be used for throwing custom exceptions, I have covered that in a separate tutorial, see Custom Exceptions in Java.

Syntax of throw keyword:

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

For example:

throw new ArithmeticException("dividing a number by 5 is not allowed in this program");

## Example of throw keyword

Lets say we have a requirement where we we need to only register the students when their age is less than 12 and weight is less than 40, if any of the condition is not met then the user should get an ArithmeticException with the warning message “Student is not eligible for registration”. We have implemented the logic by placing the code in the method that checks student eligibility if the entered student age and weight doesn’t met the criteria then we throw the exception using throw keyword.

/\* In this program we are checking the Student age

\* if the student age<12 and weight <40 then our program

\* should return that the student is not eligible for registration.

\*/

public class ThrowExample {

static void checkEligibilty(int stuage, int stuweight){

if(stuage<12 && stuweight<40) {

throw new ArithmeticException("Student is not eligible for registration");

}

else {

System.out.println("Student Entry is Valid!!");

}

}

public static void main(String args[]){

System.out.println("Welcome to the Registration process!!");

checkEligibilty(10, 39);

System.out.println("Have a nice day..");

}

}

Output:

Welcome to the Registration process!!Exception in thread "main"

java.lang.ArithmeticException: Student is not eligible for registration

at beginnersbook.com.ThrowExample.checkEligibilty(ThrowExample.java:9)

at beginnersbook.com.ThrowExample.main(ThrowExample.java:18)

In the above example we have throw an unchecked exception, same way we can throw uncheked and user-defined exception as well.

In java we have already defined, exception classes such as ArithmeticException, NullPointerException etc. These exceptions are already set to trigger on pre-defined conditions such as when you divide a number by zero it triggers ArithmeticException, In the last tutorial we learnt how to throw these exceptions explicitly based on your conditions using [throw keyword](https://beginnersbook.com/2013/04/throw-in-java/).

In java we can create our own exception class and throw that exception using throw keyword. These exceptions are known as user-defined or custom exceptions. In this tutorial we will see how to create your own custom exception and throw it on a particular condition.

To understand this tutorial you should have the basic knowledge of [try-catch block](https://beginnersbook.com/2013/04/try-catch-in-java/) and [throw in java](https://beginnersbook.com/2013/04/throw-in-java/).

## Example of User defined exception in Java

/\* This is my Exception class, I have named it MyException

\* you can give any name, just remember that it should

\* extend Exception class

\*/

class MyException extends Exception{

String str1;

/\* Constructor of custom exception class

\* here I am copying the message that we are passing while

\* throwing the exception to a string and then displaying

\* that string along with the message.

\*/

MyException(String str2) {

str1=str2;

}

public String toString(){

return ("MyException Occurred: "+str1) ;

}

}

class Example1{

public static void main(String args[]){

try{

System.out.println("Starting of try block");

// I'm throwing the custom exception using throw

throw new MyException("This is My error Message");

}

catch(MyException exp){

System.out.println("Catch Block") ;

System.out.println(exp) ;

}

}

}

Output:

Starting of try block

Catch Block

MyException Occurred: This is My error Message

Explanation:  
You can see that while throwing custom exception I gave a string in parenthesis ( throw new MyException("This is My error Message");). That’s why we have a parameterized constructor (with a String parameter) in my custom exception class.

Notes:  
1. User-defined exception must extend Exception class.  
2. The exception is thrown using throw keyword.

## Another Example of Custom Exception

In this example we are throwing an exception from a method. In this case we should use throws clause in the method signature otherwise you will get compilation error saying that “unhandled exception in method”. To understand how throws clause works, refer this guide: throws keyword in java.

class InvalidProductException extends Exception

{

public InvalidProductException(String s)

{

// Call constructor of parent Exception

super(s);

}

}

public class Example1

{

void productCheck(int weight) throws InvalidProductException{

if(weight<100){

throw new InvalidProductException("Product Invalid");

}

}

public static void main(String args[])

{

Example1 obj = new Example1();

try

{

obj.productCheck(60);

}

catch (InvalidProductException ex)

{

System.out.println("Caught the exception");

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

}

}

}

Output:

Caught the exception

Product Invalid