Exceptions

“An exception is an unexpected event that occurs during the execution of a program that disrupts the normal flow of instructions.”

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](https://beginnersbook.com/2013/04/try-catch-in-java/) or it should declare the exception using [throws keyword](https://beginnersbook.com/2013/04/java-throws/), 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](https://beginnersbook.com/2013/04/java-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.");

}

}

}

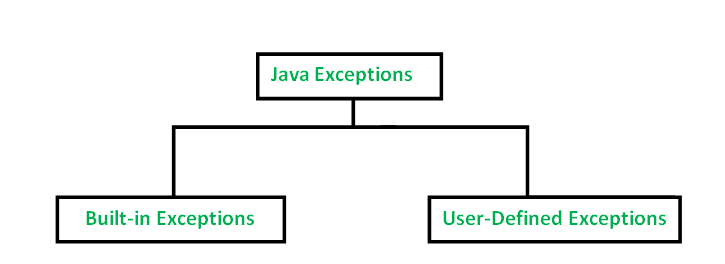
Output:

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

Here are the few unchecked exception classes:

* NullPointerException
* ArrayIndexOutOfBoundsException
* ArithmeticException
* IllegalArgumentException
* NumberFormatException

# **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**](https://www.geeksforgeeks.org/built-exceptions-java-examples/)

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. **Arithmetic Exception**  
   It is thrown when an exceptional condition has occurred in an arithmetic operation.
2. **ArrayIndexOutOfBoundException**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

**Examples of Built-in Exception:**

* **Arithmetic exception**

filter\_none

edit

play\_arrow

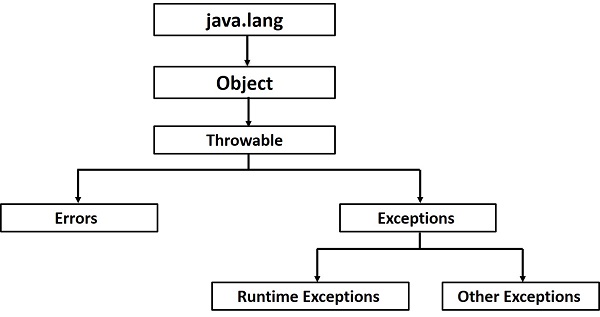
brightness\_4

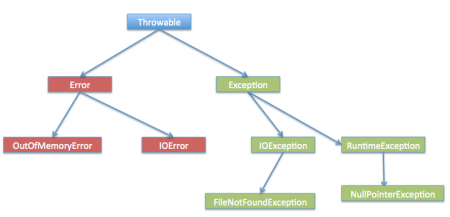
|  |
| --- |
| // Java program to demonstrate ArithmeticException  class ArithmeticException\_Demo  {      public static void main(String args[])      {          try {              int a = 30, b = 0;              int c = a/b;  // cannot divide by zero              System.out.println ("Result = " + c);          }          catch(ArithmeticException e) {              System.out.println ("Can't divide a number by 0");          }      }  } |

**Output:**

Can't divide a number by 0

Exceptions Hierarchi





## Exceptions Methods

Following is the list of important methods available in the Throwable class.

|  |  |
| --- | --- |
| **Sr.No.** | **Method & Description** |
| 1 | **public String getMessage()**  Returns a detailed message about the exception that has occurred. This message is initialized in the Throwable constructor. |
| 2 | **public Throwable getCause()**  Returns the cause of the exception as represented by a Throwable object. |
| 3 | **public String toString()**  Returns the name of the class concatenated with the result of getMessage(). |
| 4 | **public void printStackTrace()**  Prints the result of toString() along with the stack trace to System.err, the error output stream. |
| 5 | **public StackTraceElement [] getStackTrace()**  Returns an array containing each element on the stack trace. The element at index 0 represents the top of the call stack, and the last element in the array represents the method at the bottom of the call stack. |
| 6 | **public Throwable fillInStackTrace()**  Fills the stack trace of this Throwable object with the current stack trace, adding to any previous information in the stack trace. |

## User-defined Exceptions

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

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

We can define our own Exception class as below −

class MyException extends Exception {

}

You just need to extend the predefined **Exception** class to create your own Exception. These are considered to be checked exceptions. The following **InsufficientFundsException** class is a user-defined exception that extends the Exception class, making it a checked exception. An exception class is like any other class, containing useful fields and methods.

### Example

// File Name InsufficientFundsException.java

import java.io.\*;

public class InsufficientFundsException extends Exception {

private double amount;

public InsufficientFundsException(double amount) {

this.amount = amount;

}

public double getAmount() {

return amount;

}

}

To demonstrate using our user-defined exception, the following CheckingAccount class contains a withdraw() method that throws an InsufficientFundsException.

// File Name CheckingAccount.java

import java.io.\*;

public class CheckingAccount {

private double balance;

private int number;

public CheckingAccount(int number) {

this.number = number;

}

public void deposit(double amount) {

balance += amount;

}

public void withdraw(double amount) throws InsufficientFundsException {

if(amount <= balance) {

balance -= amount;

}else {

double needs = amount - balance;

throw new InsufficientFundsException(needs);

}

}

public double getBalance() {

return balance;

}

public int getNumber() {

return number;

}

}

The following BankDemo program demonstrates invoking the deposit() and withdraw() methods of CheckingAccount.

// File Name BankDemo.java

public class BankDemo {

public static void main(String [] args) {

CheckingAccount c = new CheckingAccount(101);

System.out.println("Depositing $500...");

c.deposit(500.00);

try {

System.out.println("\nWithdrawing $100...");

c.withdraw(100.00);

System.out.println("\nWithdrawing $600...");

c.withdraw(600.00);

} catch (InsufficientFundsException e) {

System.out.println("Sorry, but you are short $" + e.getAmount());

e.printStackTrace();

}

}

}

Compile all the above three files and run BankDemo. This will produce the following result −

### Output

Depositing $500...

Withdrawing $100...

Withdrawing $600...

Sorry, but you are short $200.0

InsufficientFundsException

at CheckingAccount.withdraw(CheckingAccount.java:25)

at BankDemo.main(BankDemo.java:13)

There are four keywords used in java exception handling.

1. **throw**: Sometimes we explicitly want to create exception object and then throw it to halt the normal processing of the program. **throw** keyword is used to throw exception to the runtime to handle it.
2. **throws**: When we are throwing any checked exception in a method and not handling it, then we need to use throws keyword in method signature to let caller program know the exceptions that might be thrown by the method. The caller method might handle these exceptions or propagate it to it’s caller method using throws keyword. We can provide multiple exceptions in the throws clause and it can be used with **main()** method also.
3. **try-catch**: We use try-catch block for exception handling in our code. try is the start of the block and catch is at the end of try block to handle the exceptions. We can have multiple catch blocks with a try and try-catch block can be nested also. catch block requires a parameter that should be of type Exception.
4. **finally**: finally block is optional and can be used only with try-catch block. Since exception halts the process of execution, we might have some resources open that will not get closed, so we can use finally block. finally block gets executed always, whether exception occurrs or not.

Thethrows keyword is used to specify that a method may raise an exception during its execution. It enforces explicit exception handling when calling a method:

public void simpleMethod() throws Exception {

// ...

}

Thethrow keyword allows us to throw an exception object to interrupt the normal flow of the program. This is most commonly used when a program fails to satisfy a given condition:

if (task.isTooComplicated()) {

throw new TooComplicatedException("The task is too complicated");

}

## What Is the OutOfMemoryError in Java?

The OutOfMemoryError in Java is a subclass of the java.lang. VirtualMachineError and it’s thrown by the JVM when it runs out of heap memory.

The figure below illustrates the class hierarchy of the Error class.

We can fix this error by providing more memory to run the Java application through Java options.

$CODEgt;java MyProgram -Xms1024m -Xmx1024m -XX:PermSize=64M -XX:MaxPermSize=256m

## What Is the Difference Between Final, Finally, and Finalize in Java?

**1. final:** is used to apply restrictions on the class, method, and variable. The finalclass can't be inherited — nor can it be overridden or changed.

**2. finally:** this keyword is used with the try-catch block to provide statements that will always get executed even if some exception arises. Usually, finally is used to close resources.

**3. finalize:** is used to perform clean up processing just before the object is garbage collected.