UNIT III EXCEPTION HANDLING AND I/O

Exceptions – exception hierarchy – throwing and catching exceptions – built-in exceptions, creating own exceptions, Stack Trace Elements. Input / Output Basics – Streams – Byte streams and Character streams – Reading and Writing Console – Reading and Writing Files.

Exception handling in Java with examples

Exception handling is one of the most important feature of java programming that allows us to handle the **runtime errors** caused by exceptions. In this guide, you will learn what is an exception, types of it, exception classes and how to handle exceptions in java with examples.

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 java developer can handle these exception in such a way so that the program doesn't get terminated abruptly and the user get a meaningful error message.

For example: You are writing a program for division and both the numbers are entered by user. In the following example, user can enter any number, if user enters the second number (divisor) as 0 then the program will terminate and throw an exception because dividing a number by zero gives undefined result. To get the user input, we are using **Scanner class**. Notice the output of the program.

```
import java.util.Scanner;public class JavaExample {

 public static void main(String[] args) {

   int num1, num2;
   Scanner scan = new Scanner(System.in);
   System.out.print("Enter first number(dividend): ");
   num1 = scan.nextInt();

   System.out.print("Enter second number(divisor): ");
   num2 = scan.nextInt();

   int div = num1/num2;
   System.out.println("Quotient: "+div);
 }}
```

Output:

```
Enter first number(dividend): 5
Enter second number(divisor): 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at JavaExample.main(JavaExample.java:14)
```

As you can see, the user input caused the program to throw Arithmetic exception, however this is not a good programming practice to leave such exceptions unhandled. Let's handle this exception.

Exception Handling in Java

Here, we are trying to handle the exception that is raised in the above program. You can see that the program ran fine and gave a meaningful error message which can be understood by the user.

Note: Do not worry about the try and catch blocks as we have covered these topics in detail in separate tutorials. For now just remember that the code that can throw exception needs to be inside try block and the catch block follows the try block, where the exception error message is set.

```
import java.util.Scanner;public class JavaExample {

 public static void main(String[] args) {

  int num1, num2;
  Scanner scan = new Scanner(System.in);
  System.out.print("Enter first number(dividend): ");
  num1 = scan.nextInt();

  System.out.print("Enter second number(divisor): ");
  num2 = scan.nextInt();
  try {
   int div = num1 / num2;
   System.out.println("Quotient: "+div);
  }catch(ArithmeticException e){
   System.out.println("Do not enter divisor as zero.");
   System.out.println("Error Message: "+e);
  }

 }}
```

Output:

```
Enter first number(dividend): 5
Enter second number(divisor): 0
Do not enter divisor as zero.
Error Message: java.lang.ArithmeticException: / by zero
```

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.

These system generated messages are **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 exceptions 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**.

Why we handle the exceptions?

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, that occur after the statement that caused 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 execution of program doesn't break.

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. Let's see few scenarios:

1. ArithmeticException:

We have already seen this exception in our example above. This exception occurs when we divide a number by zero. If we divide any number by zero.

```
int num = 25/0;//ArithmeticException
```

2. NullPointerException:

When a variable contains null value and you are performing an operation on the variable. For example, if a string variable contains null and you are comparing with another string. Another example is when you are trying to print the length of the string that contains null.

```
String str = null;
//NullPointerExceptionSystem.out.println(str.length());
```

3. NumberFormatException:

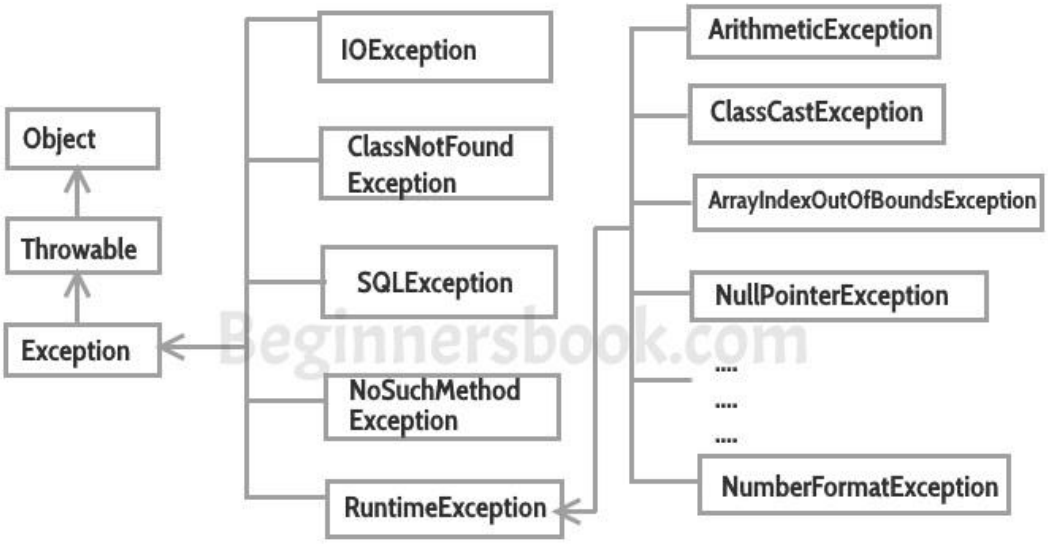
This exception occurs where there is a type mismatch. Let's say you are trying to perform an arithmetic operator on a string variable.

```
String str = "beginnersbook.com";
//NumberFormatException  int num=Integer.parseInt(str);
```

4. ArrayIndexOutOfBoundsException:

When you are trying to access the array index which is beyond the size of array. Here, we are trying to access the index 8 (9th element) but the size of the array is only 3. This exception occurs when you are accessing index which doesn't exist.

```
int arr[]=new int[3];
//ArrayIndexOutOfBoundsException
arr[8]=100;
```

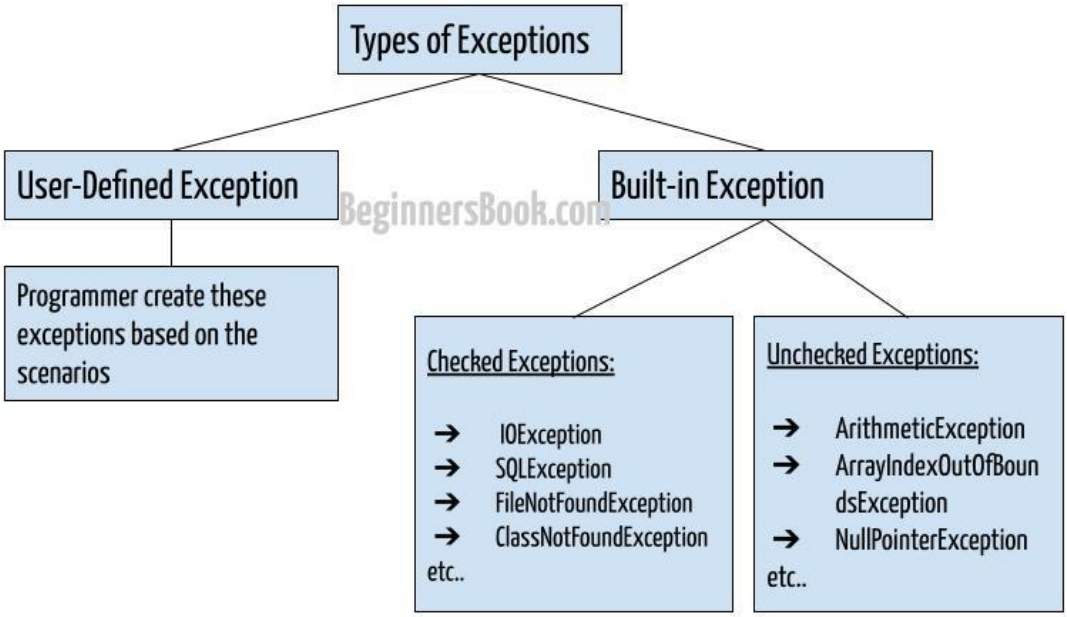


Difference between error and exception

Errors indicate that something went wrong which is not in the scope of a programmer to handle. You cannot handle an error. Also, the error doesn't occur due to bad data entered by user rather it indicates a system failure, disk crash or resource unavailability.

Exceptions are events that occurs during runtime due to bad data entered by user or an error in programming logic. 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

I have covered these topics in detail in a separate tutorial: **Checked and Unchecked exceptions in Java**.

1) 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.

2) 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. The examples that we seen above were unchecked exceptions.
Note: Compiler doesn't enforce you to catch such exceptions or ask you to declare it in the method using throws keyword.

Frequently used terms in Exception handling

try: The code that can cause the exception, is placed inside try block. The try block detects whether the exception occurs or not, if exception occurs, it transfer the flow of

program to the corresponding catch block or finally block. A try block is always followed by either a catch block or finally block.

catch: The catch block is where we write the logic to handle the exception, if it occurs. A catch block only executes if an exception is caught by the try block. A catch block is always accompanied by a try block.

finally: This block always executes whether an exception is occurred or not.

throw: It is used to explicitly throw an exception. It can be used to throw a checked or unchecked exception.

throws: It is used in method signature. It indicates that this method might throw one of the declared exceptions. While calling such methods, we need to handle the exceptions using try-catch block.

Try Catch in Java – Exception handling

Try catch block is used for **exception handling in Java**. The code (or set of statements) that can throw an exception is placed inside **try block** and if the exception is raised, it is handled by the corresponding **catch block**. In this guide, we will see various examples to understand how to use try-catch for exception handling in java.

Try block in Java

As mentioned in the beginning, try block contains set of statements where an exception can occur. A try block is always followed by a catch block or finally block, if exception occurs, the rest of the statements in the try block are skipped and the flow immediately jumps to the corresponding catch block.

Note: A try block must be followed by catch blocks or finally block or both.

Syntax of try block with catch block

```
try{
   //statements that may cause an exception}catch(Exception e){
  //statements that will execute when exception occurs}
```

Syntax of try block with finally block

```
try{
   //statements that may cause an exception}finally{
  //statements that execute whether the exception occurs or not}
```

Syntax of try-catch-finally in Java

```
try{
   //statements that may cause an exception}catch(Exception e){
  //statements that will execute if exception occurs}finally{
  //statements that execute whether the exception occurs or not}
```

Note: It is upto the programmer to choose which statements needs to be placed inside try block. If programmer thinks that certain statements in a program can throw a

exception, such statements can be enclosed inside try block and potential exceptions can be handled in catch blocks.

Catch block in Java

A catch block is where you handle the exceptions, this block must immediately placed after a 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 in Java

If an exception occurs in try block then the control of execution is passed to the corresponding catch block. As discussed earlier, 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, let's 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.

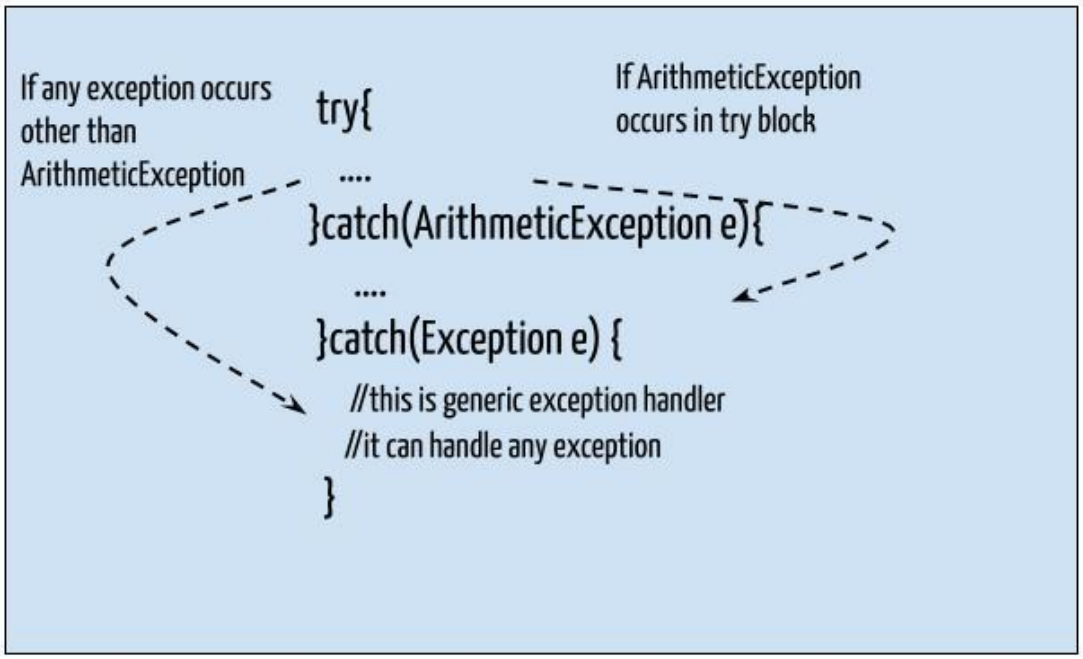
2. A generic catch block can handle all the exceptions. Whether it is ArrayIndexOutOfBoundsException or ArithmeticException or NullPointerException or any other type of exception, this handles all of them. To see the examples of NullPointerException and ArrayIndexOutOfBoundsException, refer this article: **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 handle 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: ArithmeticExceptionOut 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

A finally block contains a block of code that always executes regardless of whether an exception occurs or not. See the snippet below. I have covered this in a separate tutorial here: **java finally block**.

```
try {
   // Code that may throw an exception
} catch (ExceptionType1 e1) {
   // Handle exception of type ExceptionType1
} catch (ExceptionType2 e2) {
   // Handle exception of type ExceptionType2
} finally {
   // Code that will always execute
```

How to Catch multiple exceptions

Catching multiple exceptions

Lets take an example to understand how to handle multiple exceptions.

```
class Example{
  public static void main(String args[]){
    try{
      int arr[]=new int[7];
      arr[4]=30/0;
      System.out.println("Last Statement of try block");
    }
    catch(ArithmeticException e){
      System.out.println("You should not divide a number by zero");
    }
    catch(ArrayIndexOutOfBoundsException e){
      System.out.println("Accessing array elements outside of the limit");
    }
    catch(Exception e){
      System.out.println("Some Other Exception");
    }
    System.out.println("Out of the try-catch block");
  }}
```

Output:

```
You should not divide a number by zeroOut of the try-catch block
```

In the above example, the first catch block got executed because the code we have written in try block throws ArithmeticException (because we divided the number by zero).

Now lets change the code a little bit and see the change in output:

```
class Example{
  public static void main(String args[]){
```

```
    try{
      int arr[]=new int[7];
      arr[10]=10/5;
      System.out.println("Last Statement of try block");
    }
    catch(ArithmeticException e){
      System.out.println("You should not divide a number by zero");
    }
    catch(ArrayIndexOutOfBoundsException e){
      System.out.println("Accessing array elements outside of the limit");
    }
    catch(Exception e){
      System.out.println("Some Other Exception");
    }
    System.out.println("Out of the try-catch block");
  }}
```

Output:

```
Accessing array elements outside of the limitOut of the try-catch block
```

In this case, the second catch block got executed because the code throws ArrayIndexOutOfBoundsException. We are trying to access the 11th element of array in above program but the array size is only 7.

What did we observe from the above two examples?

1. It is clear that when an exception occurs, the specific catch block (that declares that exception) executes. This is why in first example first block executed and in second example second catch.
2. Although I have not shown you above, but if an exception occurs in above code which is not Arithmetic and ArrayIndexOutOfBounds then the last generic catch handler would execute.

Lets change the code again and see the output:

```
class Example{
  public static void main(String args[]){
    try{
      int arr[]=new int[7];
      arr[10]=10/5;
      System.out.println("Last Statement of try block");
    }
    catch(Exception e){
      System.out.println("Some Other Exception");
    }
    catch(ArithmeticException e){
      System.out.println("You should not divide a number by zero");
    }
    catch(ArrayIndexOutOfBoundsException e){
      System.out.println("Accessing array elements outside of the limit");
    }
```

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

Output:

```
Compile time error: Exception in thread "main" java.lang.Error: Unresolved compilation problems: Unreachable catch block for ArithmeticException.It is already handled by the catch block for Exception Unreachable catch block for ArrayIndexOutOfBoundsException. It is already handled by the catch block forException at Example.main(Example1.java:11)
```

Why we got this error?

This is because we placed the generic exception catch block at the first place which means that none of the catch blocks placed after this block is reachable. You should always place this block at the end of all other specific exception catch blocks.

Nested try catch block in Java – Exception handling

When a **try catch block** is present in another try block then it is called the nested try catch block. Each time a try block does not have a catch handler for a particular **exception**, then the catch blocks of parent try block are inspected for that exception, if match is found that that catch block executes.

If neither catch block nor parent catch block handles exception then the system generated message would be shown for the exception, similar to what we see when we don't handle exception.

Lets see the syntax first then we will discuss this with an example.

Syntax of Nested try Catch

```
....//Main try blocktry {
  statement 1;
  statement 2;
  //try-catch block inside another try block
  try {
    statement 3;
    statement 4;
    //try-catch block inside nested try block
    try {
      statement 5;
      statement 6;
    }
    catch(Exception e2) {
      //Exception Message
    }
  }
  catch(Exception e1) {
    //Exception Message
  }
  }//Catch of Main(parent) try blockcatch(Exception e3) {
```

```
    //Exception Message}....
```

Nested Try Catch Example

Here we have deep (two level) nesting which means we have a try-catch block inside a nested try block. To make you understand better I have given the names to each try block in comments like try-block2, try-block3 etc.

This is how the structure is: try-block3 is inside try-block2 and try-block2 is inside main try-block, you can say that the main try-block is a grand parent of the try-block3. Refer the explanation which is given at the end of this code.

```
class NestingDemo{
  public static void main(String args[]){
    //main try-block
    try{
          //try-block2
      try{
        //try-block3
        try{
          int arr[]= {1,2,3,4};
          /* I'm trying to display the value of
           * an element which doesn't exist. The
           * code should throw an exception
           */
          System.out.println(arr[10]);
        }catch(ArithmeticException e){
          System.out.print("Arithmetic Exception");
          System.out.println(" handled in try-block3");
        }
      }
      catch(ArithmeticException e){
        System.out.print("Arithmetic Exception");
        System.out.println(" handled in try-block2");
      }
  }
  catch(ArithmeticException e3){
          System.out.print("Arithmetic Exception");
          System.out.println(" handled in main try-block");
  }
  catch(ArrayIndexOutOfBoundsException e4){
          System.out.print("ArrayIndexOutOfBoundsException");
          System.out.println(" handled in main try-block");
  }
  catch(Exception e5){
          System.out.print("Exception");
          System.out.println(" handled in main try-block");
   }
 }}
```

Output:

```
ArrayIndexOutOfBoundsException handled in main try-block
```

As you can see that the ArrayIndexOutOfBoundsException occurred in the grand child try-block3. Since try-block3 is not handling this exception, the control then gets transferred to the parent try-block2 and looked for the catch handlers in try-block2. Since the try-block2 is also not handling that exception, the control gets transferred to the main (grand parent) try-block where it found the appropriate catch block for exception. This is how the the nesting structure works.

Example 2: Nested try block

```
class Nest{
  public static void main(String args[]){
             //Parent try block
    try{
           //Child try block1
       try{
         System.out.println("Inside block1");
         int b =45/0;
         System.out.println(b);
       }
       catch(ArithmeticException e1){
         System.out.println("Exception: e1");
       }
       //Child try block2
       try{
         System.out.println("Inside block2");
         int b =45/0;
         System.out.println(b);
       }
       catch(ArrayIndexOutOfBoundsException e2){
         System.out.println("Exception: e2");
       }
      System.out.println("Just other statement");
   }
   catch(ArithmeticException e3){
            System.out.println("Arithmetic Exception");
      System.out.println("Inside parent try catch block");
   }
   catch(ArrayIndexOutOfBoundsException e4){
           System.out.println("ArrayIndexOutOfBoundsException");
      System.out.println("Inside parent try catch block");
   }
   catch(Exception e5){
           System.out.println("Exception");
      System.out.println("Inside parent try catch block");
   }
   System.out.println("Next statement..");
 }}
```

Output:

```
Inside block1Exception: e1Inside block2Arithmetic ExceptionInside parent try catch blockNext statement..
```

This is another example that shows how the nested try block works. You can see that there are two try-catch block inside main try block's body. I've marked them as block 1 and block 2 in above example.

Block1: I have divided an integer by zero and it caused an ArithmeticException, since the catch of block1 is handling ArithmeticException "Exception: e1" displayed.

Block2: In block2, ArithmeticException occurred but block 2 catch is only handling ArrayIndexOutOfBoundsException so in this case control jump to the Main try-catch(parent) body and checks for the ArithmeticException catch handler in parent catch blocks. Since catch of parent try block is handling this exception using generic Exception handler that handles all exceptions, the message "Inside parent try catch block" displayed as output.

Parent try Catch block: No exception occurred here so the "Next statement.." displayed.

The important point to note here is that whenever the child catch blocks are not handling any exception, the jumps to the parent catch blocks, if the exception is not handled there as well then the program will terminate abruptly showing system generated message.

Java Finally block – Exception handling

In the previous tutorials I have covered **try-catch block** and **nested try block**. In this guide, we will see finally block which is used along with try-catch.

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 zeroThis is finally blockOut 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 blockFinally block ran even after return statement112
```

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 block15finally blockOut 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 blockfinally blockException 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 blockArithmeticExceptionfinally blockOut of try-catch-finally block
```

How to throw exception in java with example

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.

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)
```

Java Throws Keyword in Exception handling

The throws keyword is used to handle checked exceptions. As we learned in the previous article that exceptions are of two types: **checked and unchecked**. Checked exception (compile time) needs to be handled else the program won't compile. On the other hand unchecked exception (Runtime) doesn't get checked during compilation. **Throws keyword** is **used for handling checked exceptions**. You can declare multiple exceptions using throws keyword.

The throws keyword vs try-catch in Java

You may be wondering why we need throws keyword when we can handle exceptions using try-catch block in Java. Well, thats a valid question. We already know we can **handle exceptions** using **try-catch block**.

The throws keyword 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() the statements inside this method 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
```

User defined exception in java

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**.

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** and **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 blockCatch BlockMyException 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 exceptionProduct Invalid
```

Java Exception Handling Examples

We will see exception handling of ArithmeticException, ArrayIndexOutOfBoundsException, NumberFormatException, StringIndexOutOfBoundsException and NullPointerException.

Example 1: Arithmetic exception

This exception occurs when the result of a **division operation** is undefined. When a number is divided by zero, the result is undefined and that is when this exception occurs.

```
class JavaExample{
  public static void main(String args[])
  {
    try{
      int num1=30, num2=0;
      int output=num1/num2;
      System.out.println ("Result: "+output);
```

```
   }
   catch(ArithmeticException e){
     System.out.println ("You Shouldn't divide a number by zero");
   }
 }}
```

Output of above program:

```
You Shouldn't divide a number by zero
```

Example 2: ArrayIndexOutOfBounds Exception

This exception occurs when you try to access an array index that doesn't exist. For example, If array is having only 5 elements and you are trying to display 7th element then it would throw this exception.

```
class JavaExample{
  public static void main(String args[])
  {
    try{
      int a[]=new int[10];
      // This will throw exception as Array has
      // only 10 elements and we are trying to access
      // 12th element.
      a[11] = 9;
    }
    catch(ArrayIndexOutOfBoundsException e){
      System.out.println ("ArrayIndexOutOfBounds Exception occurred");
      System.out.println ("System Message: "+e);
    }
  }}
```

Output:

```
ArrayIndexOutOfBounds Exception occurred
System Message: java.lang.ArrayIndexOutOfBoundsException: 11
```

In the above example the array is initialized to store only 10 elements indexes 0 to 9. Since we are trying to access element of index 11, the program is throwing this exception.

Example 3: NumberFormat Exception

This exception occurs when a string is parsed to any numeric variable.
For example, the statement int num=Integer.parseInt ("XYZ"); would throw NumberFormatException because String "XYZ" cannot be parsed to int.

```
class JavaExample{
  public static void main(String args[])
  {
    try{
      int num=Integer.parseInt ("XYZ") ;
      System.out.println(num);
```

```
   }catch(NumberFormatException e){
     System.out.println("Number format exception occurred");
   }
  }}
```

Output:

```
Number format exception occurred
```

Example 4: StringIndexOutOfBound Exception

Class: Java.lang.StringIndexOutOfBoundsException

- A string is nothing but an **array of string** type. This exception occurs when you try to access an index that doesn't exist, similar to what we have seen in ArrayIndexOutOfBoundsException.
- Each character of a string object is stored in a particular index starting from 0. For example: In the string "beginnersbook", the char 'b' is stored at index 0, char 'e' at index 1 and so on.
- To get a character present in a particular index of a string we can use a **method charAt(int)** of **java.lang.String** where int argument is the index.

In the following example, the scope of the string "beginnersbook" is from index 0 to 12, however we are trying to access the char at index 40, which doesn't exist, hence the exception occurred.

```
class JavaExample{
  public static void main(String args[])
  {
    try{
      String str="beginnersbook";
      System.out.println(str.length());
      char c = str.charAt(40);
      System.out.println(c);
    }catch(StringIndexOutOfBoundsException e){
      System.out.println("StringIndexOutOfBoundsException.");
    }
  }}
```

Output:

```
13StringIndexOutOfBoundsException.
```

Exception occurred because the referenced index was not present in the String.

Example 5: NullPointer Exception

Class: Java.lang.NullPointer Exception

This exception occurs when you are trying to perform some operation on an object that references to the null value.

```
class JavaExample{
  public static void main(String args[])
```

```
  {
   try{
     String str=null;
     System.out.println (str.length());
   }
   catch(NullPointerException e){
     System.out.println("NullPointerException..");
   }
  }}
```

Output:

```
NullPointerException..
```

Here, length() is the function, which should be used on an object. However in the above example String object str is null so it is not an object due to which NullPointerException occurred.

Flow control in try-catch-finally in Java

Exception doesn't occur in try block

If exception doesn't occur in try block then all the catch blocks are ignored, however the **finally block** executes (if it is present) because it executes whether exception occurs or not.

Case 1: try-catch block with no finally block:

```
class JavaExample{
  public static void main(String args[])
  {
    int x = 10;
    int y = 10;
    try{
      int num= x/y;
      System.out.println("Remaining statements inside try block");
    }
    catch(Exception ex)
    {
      System.out.println("Exception caught in catch block");
    }
    System.out.println("Statements Outside of try-catch");
  }}
```

Output:

```
Remaining statements inside try blockStatements Outside of try-catch
```

Case 2: Same scenario with try-catch-finally clause:

```
public class JavaExample {
  public static void main(String args[]){
```

```
   //declared and initialized two variables
   int num1 =10, num2 = 5;
   try
   {
     int div = num1/num2;

     // if exception occurs in the above statement then this
     // statement will not execute else it will execute
     System.out.println("num1/num2: "+div);
   }
   catch(ArithmeticException e)
   {
     System.out.println("Catch block: ArithmeticException caught");
   }
   finally{
     System.out.println("Finally block: I will always execute");
   }

   // rest of the code
   System.out.println("Outside try-catch-finally");
 }}
```

Output:

```
num1/num2: 2
Finally block: I will always execute
Outside try-catch-finally
```

Exception occurred in try block and handled in catch block

If exception occurs in try block then the rest of the statements inside try block are ignored and the corresponding catch block executes. After catch block, the finally block executes and then the rest of the program.

In the following example, an Arithmetic exception occurred as the number is divided by zero, there is a catch block to handle Arithmetic exception so the control got transferred to it. After which the statements inside finally block (if present) are executed.

Case 1: try-catch without finally:

```
class JavaExample{
  public static void main(String args[])
  {
    int x = 10;
    int y = 0;
    try{
      int num= x/y;
      System.out.println("Remaining statements inside try block");
```

```
   }
   catch(Exception ex)
   {
     System.out.println("Exception caught in catch block");
   }
   System.out.println("Statements Outside of try-catch");
 }}
```

Output:

```
Exception caught in catch blockStatements Outside of try-catch
```

Point to note in above example: There are two statements present inside try block. Since exception occurred because of first statement, the second statement didn't execute. Hence we can say that if an exception occurs then the rest of the statements in try block don't execute and control passes to catch block.

Case 2: try-catch with finally:

```
public class JavaExample {
  public static void main(String args[]){
    //now the second variable is initialized with 0 value
    int num1 =10, num2 = 0;
    try
    {
      int div = num1/num2;

      // if exception occurs in the above statement then this
      // statement will not execute else it will execute
      System.out.println("num1/num2: "+div);
    }
    catch(ArithmeticException e)
    {
      System.out.println("Catch block: ArithmeticException caught");
    }
    finally{
      System.out.println("Finally block: I will always execute");
    }

    // rest of the code
    System.out.println("Outside try-catch-finally");
  }}
```

Output:

```
Catch block: ArithmeticException caught
Finally block: I will always execute
Outside try-catch-finally
```

Exception occurred in try block and not handled in catch block

If the exception raised in try block is not handled in catch block then the rest of the statements in try block and the statements after try-catch-finally doesn't execute, only the finally block executes and a system generated error message for the exception that occurred in try block.

In the following example, the ArithmeticException occurred in try block but there is no catch block that can handle ArithmeticException so after the execution of finally block, a system generated error message is displayed.

```
public class JavaExample {
  public static void main(String args[]){
    //now the second variable is initialized with 0 value
    int num1 =10, num2 = 0;
    try
    {
      int div = num1/num2;

      // if exception occurs in the above statement then this
      // statement will not execute else it will execute
      System.out.println("num1/num2: "+div);
    }
    //this cannot handle ArithmeticException
    catch(ArrayIndexOutOfBoundsException e)
    {
      System.out.println("Catch block: ArrayIndexOutOfBoundsException caught");
    }
    finally{
      System.out.println("Finally block executed");
    }

    // rest of the code
    System.out.println("Outside try-catch-finally");
  }}
```

Output:

```
Finally block executed
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at JavaExample.main(JavaExample.java:7)
```

Flow of control in try catch finally in Java:

To summarise everything we have learned so far:

1. If exception occurs in try block then control immediately transfers(**skipping rest of the statements in try block**) to the catch block. Once catch block finished execution then **finally block** and after that rest of the program.
2. If no exception occurs in try block, then try block gets executed completely and then control gets transferred to finally block (**skipping catch blocks**), after which rest of the statements after try-catch-finally are executed

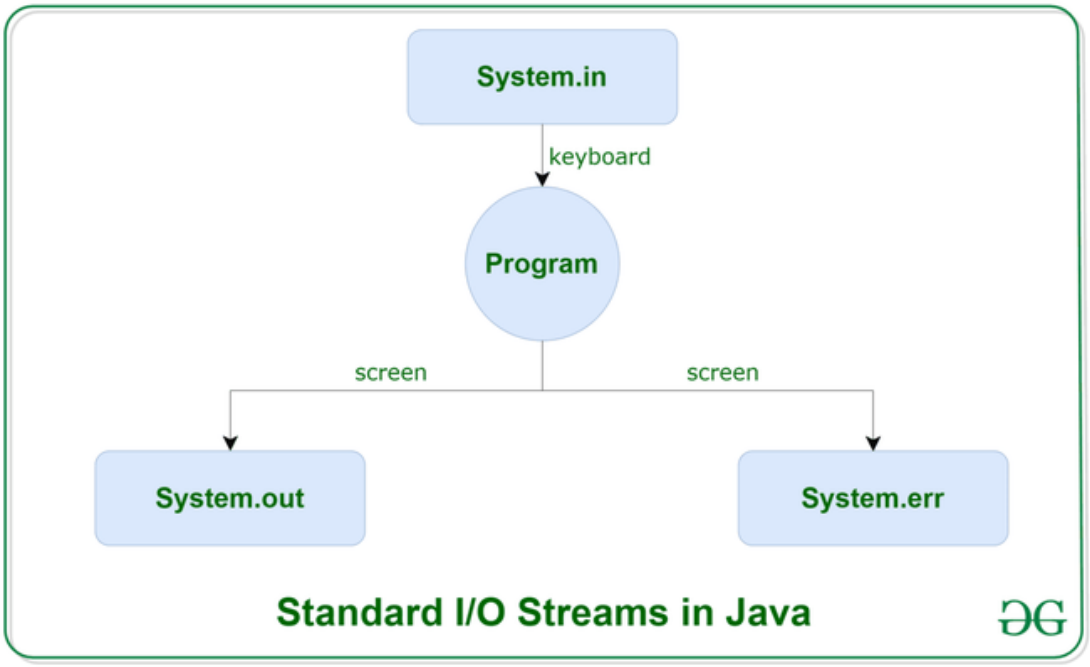
Java IO - Input/Output in Java with Examples

Java provides various **Streams** with its **I/O package** that helps the user to perform all the input-output operations. These streams support all the types of objects, data-types, characters, files, etc., to fully execute the **I/O operations**.

The image below demonstrates the **flow of data from a source to a destination.**

Standard or Default Streams in Java

Before exploring various input and output streams, let's look at **3 standard or default streams** that Java has to provide, which are also most commonly used:



- **System.in:** This is the **standard input stream (**System.in) that is used to read characters from the keyboard or any other standard input device.
- **System.out:** This is the **standard output stream (**System.out) that is used to produce the result of a program on an output device like the computer screen.

Here is a list of the various print functions that we use to output statements:

Java Print Functions Used with System.out

Now, we are going to discuss the main print function used with System.out which are listed below:

1. print()**:** This method in Java is used to display a text on the console. This text is passed as the parameter to this method in the form of String. This method prints the text on the console and the cursor remains at the end of the text at the console. The next printing takes place from just here.

Syntax:

System.out.print(parameter);

Example:

```
// Java program to illustrate print()
import java.io.*;

class Geeks {
   public static void main(String[] args)
   {
      // using print()
      // all are printed in the
      // same line
      System.out.print("GfG! ");
      System.out.print("GfG! ");
      System.out.print("GfG! ");
   }
}
```

Output

```
GfG! GfG! GfG!
```

2. println()**:** This method in Java is also used to display a text on the console. It prints the text on the console and the cursor moves to the start of the next line at the console. The next printing takes place from the next line.

Syntax:

System.out.println(parameter);

Example:

```
// Java program to illustrate println()
import java.io.*;

class Geeks {
   public static void main(String[] args)
   {
      // using println()
      // all are printed in the
      // different line
      System.out.println("GfG! ");
      System.out.println("GfG! ");
      System.out.println("GfG! ");
   }
```

```
}
```

Output

```
GfG!
GfG!
GfG!
```

3. printf()**:** This is the easiest of all methods as this is similar to **printf in C.** Note that **System.out.print()** and **System.out.println()** take a single argument, but **printf()** may take multiple arguments. This is used to format the output in Java.

Example:

```
// A Java program to demonstrate
// working of printf() in Java
class Geeks {

    public static void main(String[] args) {
        int x = 100;

        // Printing a simple integer
        System.out.printf("Printing simple integer: x = %d%n", x);

        // Printing a floating-point
        // number with precision
        System.out.printf("Formatted with precision: PI = %.2f%n", Math.PI);

        float n = 5.2f;

        // Formatting a float to 4 decimal places
        System.out.printf("Formatted to specific width: n = %.4f%n", n);

        n = 2324435.3f;

        // Right-aligning and formatting a
        // float to 20-character width
        System.out.printf("Formatted to right margin: n = %20.4f%n", n);
    }
}
```

Output

```
Printing simple integer: x = 100
Formatted with precision: PI = 3.14
Formatted to specific width: n = 5.2000
Formatted to right margin: n =         2324435.2500
```

System.err Example

It is used to display the error messages. It works similarly to System.out with **print(), println(),** and **printf()** methods.

Example:

```
// Java Program demonstrating System.errpublic class Geeks {     public static void main(String[] args) {          // Using print()       System.err.print("This is an error message using print().\n");
      // Using println()       System.err.println("This is another error message using println().");
      //Using printf()       System.err.printf("Error code: %d, Message: %s%n", 404, "Not Found");    }}
```

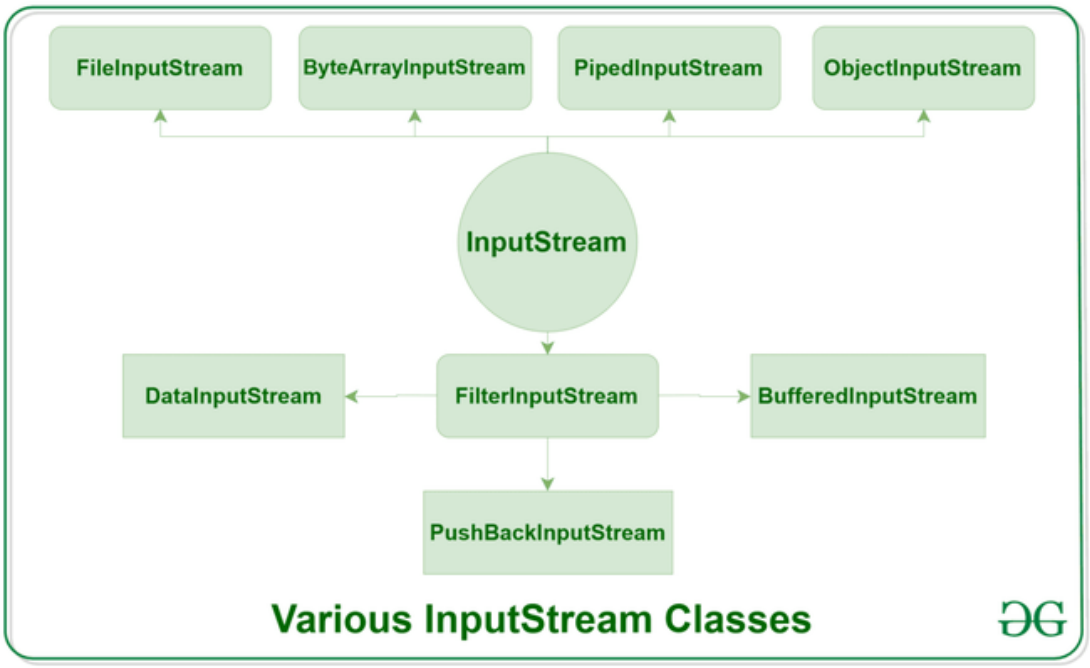
Output:

```
This is an error message using print().
This is another error message using println().
Error code: 404, Message: Not Found
```

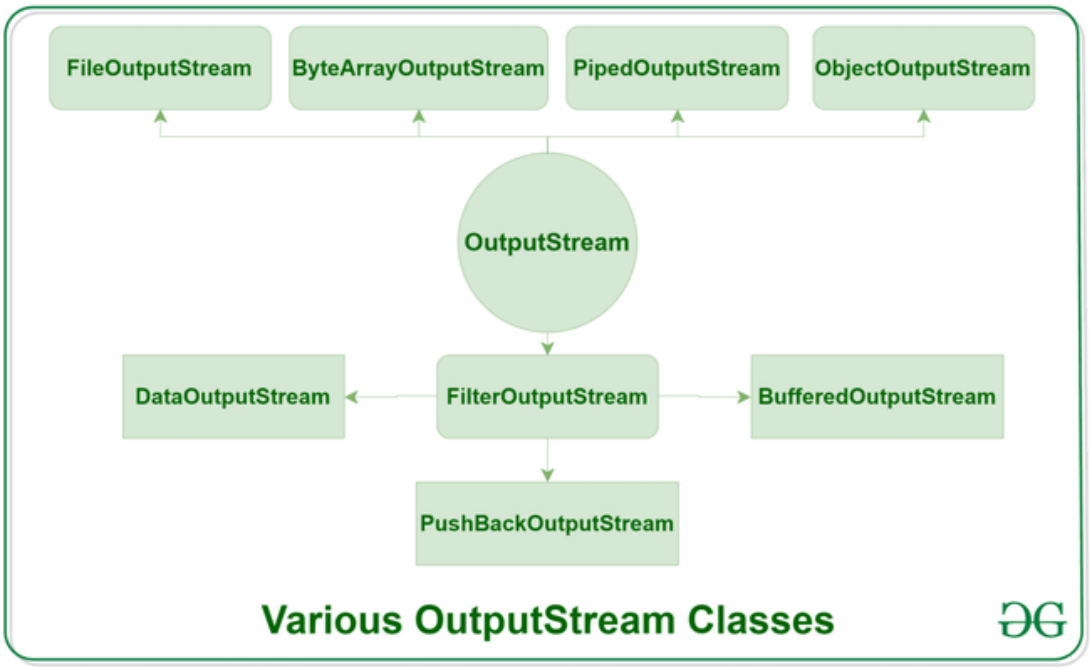
Types of Streams

Depending on the type of operations, streams can be divided into two primary classes:

1. Input Stream: These streams are used to read data that must be taken as an input from a source array or file or any peripheral device. For eg., **FileInputStream, BufferedInputStream, ByteArrayInputStream** etc.

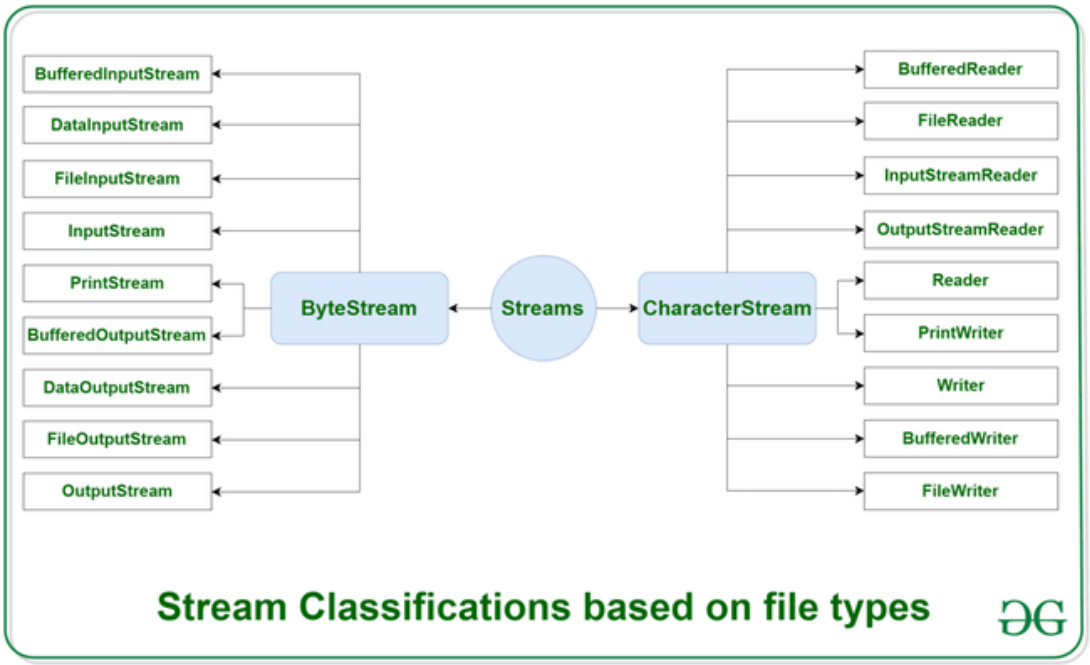


2. Output Stream: These streams are used to write data as outputs into an array or file or any output peripheral device. For eg., **FileOutputStream, BufferedOutputStream, ByteArrayOutputStream** etc.



Types of Streams Depending on the Types of File

Depending on the types of file, Streams can be divided into two primary classes which can be further divided into other classes as can be seen through the diagram below followed by the explanations.



1. ByteStream:This is used to process data byte by byte (8 bits). Though it has many classes, the FileInputStream and the FileOutputStream are the most popular ones. The FileInputStream is used to read from the source and FileOutputStream is used to write to the destination.

Here is the list of various ByteStream Classes:

Stream class	Description
BufferedInputStream	It is used for Buffered Input Stream.
DataInputStream	It contains method for reading java standard datatypes.
FileInputStream	This is used to reads from a file
InputStream	This is an abstract class that describes stream input.
PrintStream	This contains the most used print() and println() method
BufferedOutputStream	This is used for Buffered Output Stream.
DataOutputStream	This contains method for writing java standard data types.
FileOutputStream	This is used to write to a file.
OutputStream	This is an abstract class that describe stream output.

Example:

```
// Java Program illustrating the// Byte Stream to copy// contents of one file to another file.import java.io.*;public class Geeks {    public static void main(        String[] args) throws IOException    {
        FileInputStream sourceStream = null;        FileOutputStream targetStream = null;
        try {            sourceStream                = new FileInputStream("sorcefile.txt");
targetStream                = new FileOutputStream("targetfile.txt");
            // Reading source file and writing            // content to target file byte by byte
int temp;            while ((                    temp = sourceStream.read())                    != -1)
targetStream.write((byte)temp);        }        finally {            if (sourceStream != null)
sourceStream.close();            if (targetStream != null)
targetStream.close();        }    }}
```

Output:

```
Shows contents of file test.txt
```

2. CharacterStream: In Java, characters are stored using Unicode conventions (Refer this for details). Character stream automatically allows us to read/write data character by character. Though it has many classes, the FileReader and the

FileWriter are the most popular ones. FileReader and FileWriter are character streams used to read from the source and write to the destination respectively.

Here is the list of various **CharacterStream** Classes:

Stream class	Description
BufferedReader	It is used to handle buffered input stream.
FileReader	This is an input stream that reads from file.
InputStreamReader	This input stream is used to translate byte to character.
OutputStreamReader	This output stream is used to translate character to byte.
Reader	This is an abstract class that define character stream input.
PrintWriter	This contains the most used print() and println() method
Writer	This is an abstract class that define character stream output.
BufferedWriter	This is used to handle buffered output stream.
FileWriter	This is used to output stream that writes to file.

Example:

```
// Java Program illustrating that// we can read a file in a human-readable// format using FileReader
// Accessing FileReader, FileWriter,// and IOExceptionimport java.io.*;
public class Geeks {    public static void main(String[] args) throws IOException {       FileReader sourceStream = null;             try {           sourceStream = new FileReader("test.txt");
          // Reading sourcefile and          // writing content to target file          // character by character.          int temp;                   while (( temp = sourceStream.read())!= -1 )              System.out.println((char)temp);       } finally {                 // Closing stream as no longer in use          if (sourceStream != null)              sourceStream.close();       }    }
```

****