

## Java Unit III

Exception handling - Fundamentals of exception handling, Exception types, Termination or resumptive models, Uncaught exceptions, using try and catch, multiple catch clauses, nested try statements, throw, throws and finally, built- in exceptions, creating own exception sub classes.

### 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, we 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 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.

Sr. No.	Key	Error	Exception
1	Type	Classified as an unchecked type	Classified as checked and unchecked
2	Package	It belongs to java.lang.error	It belongs to java.lang.Exception
3	Recoverable/ Irrecoverable	It is irrecoverable	It is recoverable
4		It can't be occur at compile time	It can occur at run time compile time both
5	Example	OutOfMemoryError ,IOException	NullPointerException , SQLException

### Example of Error

```
public class ErrorExample {  
    public static void main(String[] args){  
        recursiveMethod(10)  
    }  
    public static void recursiveMethod(int i){  
        while(i!=0){  
            i=i+1;  
            recursiveMethod(i);  
        }  
    }  
}
```

### Output

```
Exception in thread "main" java.lang.StackOverflowError  
    at ErrorExample.ErrorExample(Main.java:42)
```

### Example of Exception

```
public class ExceptionExample {  
    public static void main(String[] args){  
        int x = 100;  
        int y = 0;  
        int z = x / y;  
    }  
}
```

### Output

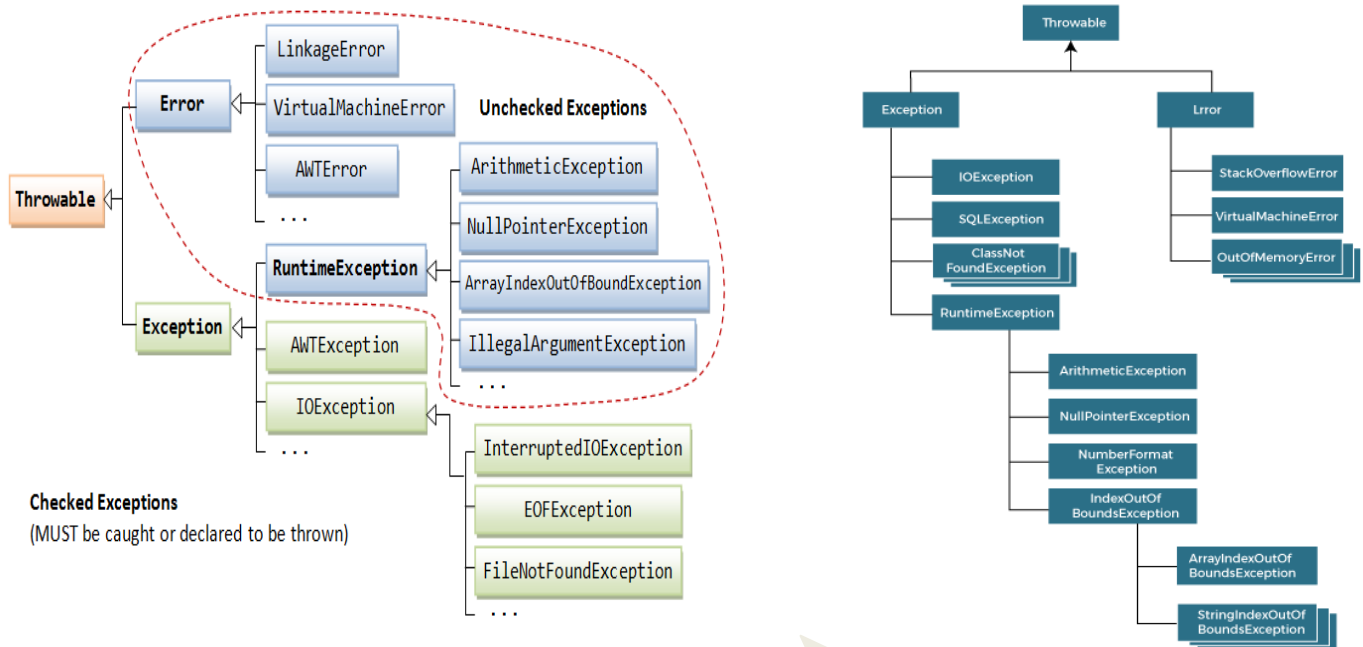
```
java.lang.ArithmeticException: / by zero    at ExceptionExample.main(ExceptionExample.java:7)
```

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.

## Java Exception Keywords

Java provides five keywords that are used to handle the exception. The following table describes each.

Keyword	Description
try	The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the necessary code of the program. It is executed whether an exception is handled or not.
throw	The "throw" keyword is used to throw an exception.
throws	The "throws" keyword is used to declare exceptions. It specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature.

### Try Catch in Java – Exception handling

In the [previous tutorial](#) we discussed what is exception handling and why we do it. In this tutorial we will see try-catch block which is used for exception handling.

#### 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 it 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 than 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`,

```
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. That's the reason you should place it 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,

### 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 execute 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.

### How to Catch multiple exceptions

In the [previous tutorial](#), I have covered how to handle exceptions using try-catch blocks. In this guide, we will see how to handle multiple exceptions and how to write them in a correct order so that user gets a meaningful message for each type of exception.

## 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 zero  
Out 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 limit  
Out 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 ArrayIndexOutOfBoundsException 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 for  
Exception 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 block
try {
    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 block
catch(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 block1
Exception: e1
Inside block2
Arithmetic Exception
Inside parent try catch block
Next 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 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);
    ObjectOutputStream 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



## How to throw exception in java with example

In Java we have already defined exception classes such as `ArithmeticException`, `NullPointerException`, `ArrayIndexOutOfBoundsException` 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 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 `unchecked` and `user-defined` exception as well.

## Throws clause in java – Exception handling

As we know that there are two types of exception **checked** and **unchecked**. Checked exception (compile time) force you to handle them, if you don't handle them then the program will not compile. On the other hand unchecked exception (Runtime) doesn't get checked during compilation. **Throws keyword** is used for handling checked exceptions. By using throws we can declare multiple exceptions in one go.

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

Well, that's a valid question. We already know we can **handle exceptions** using **try-catch block**. The throws does the same thing that try-catch does but there are some cases where you would prefer throws over try-catch. For example:

Lets say we have a method `myMethod()` that has statements that can throw either `ArithmeticException` or `NullPointerException`, in this case you can use try-catch as shown below:

```
public void myMethod()
{
    try {
        // Statements that might throw an exception
    }
    catch (ArithmeticException e) {
        // Exception handling statements
    }
    catch (NullPointerException e) {
        // Exception handling statements
    }
}
```

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

One way to overcome this problem is by using throws like this: declare the exceptions in the method signature using throws and handle the exceptions where you are calling this method by using try-catch. Another advantage of using this approach is that you will be forced to handle the exception when you call this method, all the exceptions that are declared using throws, must be handled where you are calling this method else you will get compilation error.

```
public void myMethod() throws ArithmeticException, NullPointerException
{
    // Statements that might throw an exception
}

public static void main(String args[]) {
    try {
        myMethod();
    }
```

```

}
catch (ArithmeticException e) {
    // Exception handling statements
}
catch (NullPointerException e) {
    // Exception handling statements
}
}
}

```

### Example of throws Keyword

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

```

import java.io.*;
class ThrowExample {
    void myMethod(int num)throws IOException, ClassNotFoundException{
        if(num==1)
            throw new IOException("IOException Occurred");
        else
            throw new ClassNotFoundException("ClassNotFoundException");
    }
}

public class Example1{
    public static void main(String args[]){
        try{
            ThrowExample obj=new ThrowExample();
            obj.myMethod(1);
        }catch(Exception ex){
            System.out.println(ex);
        }
    }
}

```

Output:

```

java.io.IOException: IOException Occurred

```

## 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.

### Points to Remember

1. Extend the `Exception` class to create your own exception class.
2. You don't have to implement anything inside it, no methods are required.
3. You can have a Constructor if you want.
4. You can override the `toString()` function, to display customized message.

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 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.

RK JAVA

## 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
```

### Example: Custom Exception

In this example, we are creating an exception class MyException that extends the Java Exception class and

```
class MyException extends Exception
{
    private int ex;
    MyException(int a)
    {
        ex = a;
    }
    public String toString()
    {
        return "MyException[" + ex + "] is less than zero";
    }
}

class Demo
{
    static void sum(int a,int b) throws MyException
    {
        if(a<0)
        {
            throw new MyException(a); //calling constructor of user-defined exception class
        }
        else
        {
            System.out.println(a+b);
        }
    }

    public static void main(String[] args)
    {
        try
        {
            sum(-10, 10);
        }
        catch(MyException me)
        {
            System.out.println(me); //it calls the toString() method of user-defined Exception
        }
    }
}
```

### Output:

MyException[-10] is less than zero

## Java Exception Handling examples

In this tutorial, we will see examples of few frequently used exceptions. If you looking for exception handling tutorial refer this complete guide: [Exception handling in Java](#).

### Example 1: Arithmetic exception

Class: `Java.lang.ArithmeticException`

This is a built-in-class present in `java.lang` package. This exception occurs when an integer is divided by zero.

```
class Example1
{
    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

**Explanation:** In the above example I've divided an integer by a zero and because of this `ArithmeticException` is thrown.

### Example 2: ArrayIndexOutOfBoundsException

Class: `Java.lang.ArrayIndexOutOfBoundsException`

This exception occurs when you try to access the array index which does not exist. For example, If array is having only 5 elements and we are trying to display 7th element then it would throw this exception.

```
class ExceptionDemo2
{
    public static void main(String args[])
    {
        try{
            int a[]=new int[10];
            //Array has only 10 elements
            a[11] = 9;
        }
        catch(ArrayIndexOutOfBoundsException e){
            System.out.println ("ArrayIndexOutOfBounds");
        }
    }
}
```



## Output:

### ArrayIndexOutOfBoundsException

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

## Example 3: NumberFormat Exception

Class: `Java.lang.NumberFormatException`

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 ExceptionDemo3
{
    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: StringIndexOutOfBounds Exception

Class: `Java.lang.StringIndexOutOfBoundsException`

- An object of this class gets created whenever an index is invoked of a string, which is not in the range.
- Each character of a string object is stored in a particular index starting from 0.
- 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.

E.g.

```
class ExceptionDemo4
{
    public static void main(String args[])
    {
        try{
            String str="beginnersbook";
            System.out.println(str.length());
            char c = str.charAt(0);
            c = str.charAt(40);
            System.out.println(c);
        }
    }
}
```

```

    }catch(StringIndexOutOfBoundsException e){
        System.out.println("StringIndexOutOfBoundsException!!");
    }
}
}

```

Output:

13

`StringIndexOutOfBoundsException!!`

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

### Example 5: NullPointerException

Class: `Java.lang.NullPointerException`

An object of this class gets created whenever a member is invoked with a “null” object.

```

class Exception2
{
    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.

## Unit III

**Multithreading- Differences between thread-based multitasking and process-based multitasking, Java thread model, creating threads, thread priorities, synchronizing threads, inter thread communication.**

### Multithreading in java

The java programming language allows us to create a program that contains one or more parts that can run simultaneously at the same time. This type of program is known as a multithreading program. Each part of this program is called a thread. Every thread defines a separate path of execution in java. A thread is explained in different ways, and a few of them are as specified below.

A thread is a light weight process.

A thread may also be defined as follows.

A thread is a subpart of a process that can run individually.

In java, multiple threads can run at a time, which enables the java to write multitasking programs. The multithreading is a specialized form of multitasking. All modern operating systems support multitasking. There are two types of multitasking, and they are as follows.

- Process-based multitasking
- Thread-based multitasking

It is important to know the difference between process-based and thread-based multitasking. Let's distinguish both.

Process-based multitasking	Thread-based multitasking
It allows the computer to run two or more programs concurrently	It allows the computer to run two or more threads concurrently
In this process is the smallest unit.	In this thread is the smallest unit.
Process is a larger unit.	Thread is a part of process.
Process is heavy weight.	Thread is light weight.

Process requires separate address space for each.	Threads share same address space.
Process never gain access over idle time of CPU.	Thread gain access over idle time of CPU.
Inter process communication is expensive.	Inter thread communication is not expensive.

## Java Thread Model

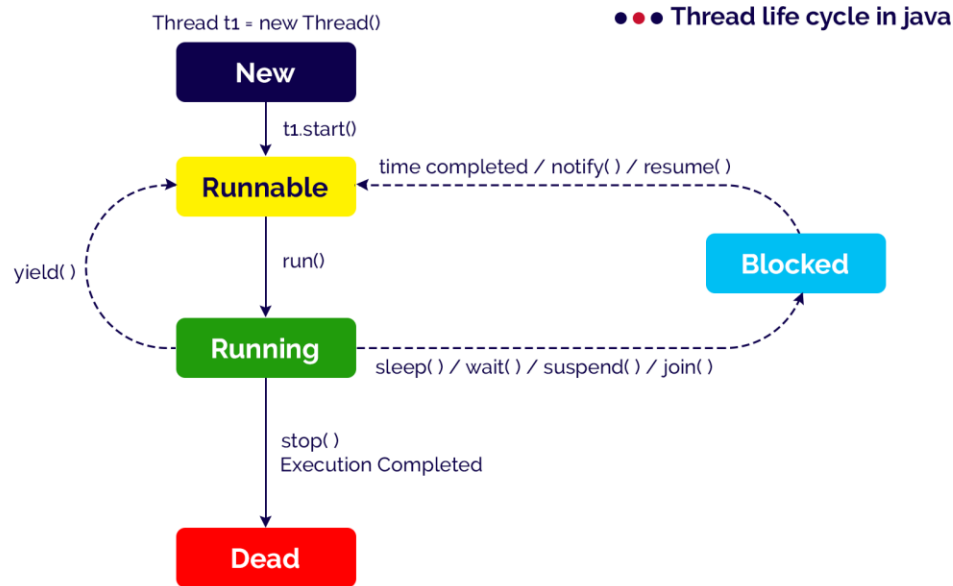
The java programming language allows us to create a program that contains one or more parts that can run simultaneously at the same time. This type of program is known as a multithreading program. Each part of this program is called a thread. Every thread defines a separate path of execution in java. A thread is explained in different ways, and a few of them are as specified below.

A thread is a light weight process.

A thread may also be defined as follows.

A thread is a subpart of a process that can run individually.

In java, a thread goes through different states throughout its execution. These stages are called thread life cycle states or phases. A thread may in any of the states like new, ready or runnable, running, blocked or wait, and dead or terminated state. The life cycle of a thread in java is shown in the following figure.



Let's look at each phase in detail.

## 1.New

When a thread object is created using `new`, then the thread is said to be in the New state. This state is also known as the Born state.

### Example

```
Thread t1 = new Thread();
```

## 2.Runnable / Ready

When a thread calls `start()` method, then the thread is said to be in the Runnable state. This state is also known as a Ready state.

### Example

```
t1.start();
```

### **3. Running**

When a thread calls run( ) method, then the thread is said to be Running. The run( ) method of a thread called automatically by the start( ) method.

### **4. Blocked / Waiting**

A thread in the Running state may move into the blocked state due to various reasons like sleep( ) method called, wait( ) method called, suspend( ) method called, and join( ) method called, etc.

When a thread is in the blocked or waiting state, it may move to Runnable state due to reasons like sleep time completed, waiting time completed, notify( ) or notifyAll( ) method called, resume( ) method called, etc.

### **Example**

```
Thread.sleep(1000);
```

```
wait(1000);
```

```
wait();
```

```
suspended();
```

```
notify();
```

```
notifyAll();
```

```
resume();
```

### **Dead / Terminated**

A thread in the Running state may move into the dead state due to either its execution completed or the stop( ) method called. The dead state is also known as the terminated state.

## Creating threads in java

In java, a thread is a lightweight process. Every java program executes by a thread called the main thread. When a java program gets executed, the main thread created automatically. All other threads called from the main thread.

The java programming language provides two methods to create threads, and they are listed below.

- **Using Thread class (by extending Thread class)**
- **Using Runnable interface (by implementing Runnable interface)**

Let's see how to create threads using each of the above.

### Extending Thread class

The java contains a built-in class Thread inside the java.lang package. The Thread class contains all the methods that are related to the threads.

To create a thread using Thread class, follow the step given below.

- **Step-1:** Create a class as a child of Thread class. That means, create a class that extends Thread class.
- **Step-2:** Override the run( ) method with the code that is to be executed by the thread. The run( ) method must be public while overriding.
- **Step-3:** Create the object of the newly created class in the main( ) method.
- **Step-4:** Call the start( ) method on the object created in the above step.

Look at the following example program.

#### Example

```
class SampleThread extends Thread{  
  
    public void run() {  
        System.out.println("Thread is under Running...");  
        for(int i= 1; i<=10; i++) {  
            System.out.println("i = " + i);  
        }  
    }  
}
```

```

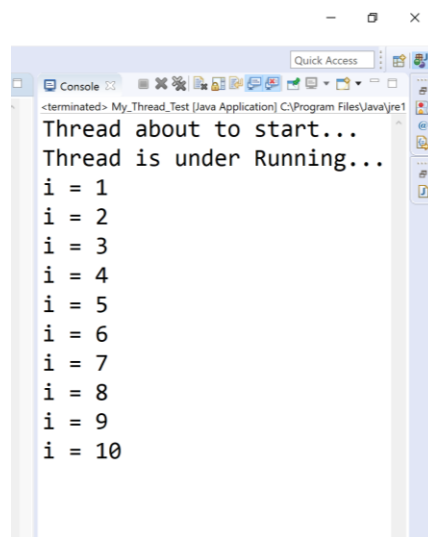
    }
}

public class My_Thread_Test {

    public static void main(String[] args) {
        SampleThread t1 = new SampleThread();
        System.out.println("Thread about to start...");
        t1.start();
    }
}

```

When we run this code, it produce the following output.



```

<terminated> My_Thread_Test [Java Application] C:\Program Files\Java\jre1
Thread about to start...
Thread is under Running...
i = 1
i = 2
i = 3
i = 4
i = 5
i = 6
i = 7
i = 8
i = 9
i = 10

```

## Implementng Runnable interface

The java contains a built-in interface Runnable inside the java.lang package. The Runnable interface implemented by the Thread class that contains all the methods that are related to the threads.

To create a thread using Runnable interface, follow the step given below.



- **Step-1:** Create a class that implements Runnable interface.
- **Step-2:** Override the run( ) method with the code that is to be executed by the thread. The run( ) method must be public while overriding.
- **Step-3:** Create the object of the newly created class in the main( ) method.
- **Step-4:** Create the Thread class object by passing above created object as parameter to the Thread class constructor.
- **Step-5:** Call the start( ) method on the Thread class object created in the above step.

Look at the following example program.

### Example

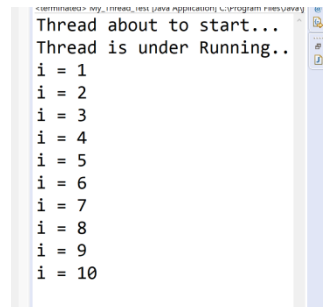
```
class SampleThread implements Runnable{

    public void run() {
        System.out.println("Thread is under Running...");
        for(int i= 1; i<=10; i++) {
            System.out.println("i = " + i);
        }
    }
}

public class My_Thread_Test {

    public static void main(String[] args) {
        SampleThread threadObject = new SampleThread();
        Thread thread = new Thread(threadObject);
        System.out.println("Thread about to start...");
        thread.start();
    }
}
```

When we run this code, it produce the following output.

A screenshot of a Java IDE's console window. The window title is "<terminated> - org.eclipse.jdt.internal.ui.console.ConsoleViewer". The output text is: "Thread about to start...", "Thread is under Running..", followed by a list of values for 'i' from 1 to 10, each on a new line: "i = 1", "i = 2", "i = 3", "i = 4", "i = 5", "i = 6", "i = 7", "i = 8", "i = 9", and "i = 10".

```
<terminated> - org.eclipse.jdt.internal.ui.console.ConsoleViewer
Thread about to start...
Thread is under Running..
i = 1
i = 2
i = 3
i = 4
i = 5
i = 6
i = 7
i = 8
i = 9
i = 10
```

## More about Thread class

The Thread class in java is a subclass of Object class and it implements Runnable interface. The Thread class is available inside the java.lang package. The Thread class has the following syntax.

```
class Thread extends Object implements Runnable{

...

}
```

The Thread class has the following constructors.

- **Thread( )**
- **Thread( String threadName )**
- **Thread( Runnable objectName )**
- **Thread( Runnable objectName, String threadName )**

The Thread class contains the following methods.

Method	Description	Return Value
run( )	Defines actual task of the thread.	void
start( )	It moves the thread from Ready state to Running state by calling run( ) method.	void
setName(String)	Assigns a name to the thread.	void
getName( )	Returns the name of the thread.	String
setPriority(int)	Assigns priority to the thread.	void
getPriority( )	Returns the priority of the thread.	int
getId( )	Returns the ID of the thread.	long
activeCount()	Returns total number of thread under active.	int
currentThread( )	Returns the reference of the thread that currently in running state.	void
sleep( long )	moves the thread to blocked state till the specified number of milliseconds.	void
isAlive( )	Tests if the thread is alive.	boolean

yield( )	Tells to the scheduler that the current thread is willing to yield its current use of a processor.	void
join( )	Waits for the thread to end.	void

□ The Thread class in java also contains methods like **stop( )**, **destroy( )**, **suspend( )**, and **resume( )**. But they are deprecated.

## Java Thread Priority

In a java programming language, every thread has a property called priority. Most of the scheduling algorithms use the thread priority to schedule the execution sequence. In java, the thread priority range from 1 to 10. Priority 1 is considered as the lowest priority, and priority 10 is considered as the highest priority. The thread with more priority allocates the processor first.

The java programming language Thread class provides two methods **setPriority(int)**, and **getPriority( )** to handle thread priorities.

The Thread class also contains three constants that are used to set the thread priority, and they are listed below.

**MAX\_PRIORITY** - It has the value 10 and indicates highest priority.

**NORM\_PRIORITY** - It has the value 5 and indicates normal priority.

**MIN\_PRIORITY** - It has the value 1 and indicates lowest priority.

□ The default priority of any thread is 5 (i.e. NORM\_PRIORITY).

### setPriority( ) method

The setPriority( ) method of Thread class used to set the priority of a thread. It takes an integer range from 1 to 10 as an argument and returns nothing (void).

The regular use of the setPriority( ) method is as follows.

### Example

```
threadObject.setPriority(4);
```

or

```
threadObject.setPriority(MAX_PRIORITY);
```

### **getPriority( ) method**

The getPriority( ) method of Thread class used to access the priority of a thread. It does not takes any argument and returns name of the thread as String.

The regular use of the getPriority( ) method is as follows.

### **Example**

```
String threadName = threadObject.getPriority();
```

Look at the following example program.

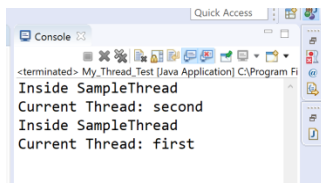
### **Example**

```
class SampleThread extends Thread{  
    public void run() {  
        System.out.println("Inside SampleThread");  
        System.out.println("Current Thread: " +  
Thread.currentThread().getName());  
    }  
}
```

```
public class My_Thread_Test {
```

```
public static void main(String[] args) {  
  
    SampleThread threadObject1 = new SampleThread();  
  
    SampleThread threadObject2 = new SampleThread();  
  
    threadObject1.setName("first");  
  
    threadObject2.setName("second");  
  
  
    threadObject1.setPriority(4);  
  
    threadObject2.setPriority(Thread.MAX_PRIORITY);  
  
  
    threadObject1.start();  
  
    threadObject2.start();  
  
    }  
}
```

When we run this code, it produce the following output.



□ In java, it is not guaranteed that threads execute according to their priority because it depends on JVM specification that which scheduling it chooses.

## Synchronization in Java

Synchronization in Java is the capability to control the access of multiple threads to any shared resource.

Java Synchronization is better option where we want to allow only one thread to access the shared resource.

### Why use Synchronization?

The synchronization is mainly used to

1. To prevent thread interference.
2. To prevent consistency problem.

### Types of Synchronization

There are two types of synchronization

1. Process Synchronization
2. Thread Synchronization

Here, we will discuss only thread synchronization.

### Thread Synchronization

There are two types of thread synchronization mutual exclusive and inter-thread communication.

1. Mutual Exclusive
  1. Synchronized method.

2. Synchronized block.
  3. Static synchronization.
2. Cooperation (Inter-thread communication in java)

## Mutual Exclusive

Mutual Exclusive helps keep threads from interfering with one another while sharing data. It can be achieved by using the following three ways:

1. By Using Synchronized Method
2. By Using Synchronized Block
3. By Using Static Synchronization

## Concept of Lock in Java

Synchronization is built around an internal entity known as the lock or monitor. Every object has a lock associated with it. By convention, a thread that needs consistent access to an object's fields has to acquire the object's lock before accessing them, and then release the lock when it's done with them.

From Java 5 the package `java.util.concurrent.locks` contains several lock implementations.

## Understanding the problem without Synchronization

In this example, there is no synchronization, so output is inconsistent. Let's see the example:

### TestSynchronization1.java

1. `class Table{`
2. `void printTable(int n){ //method not synchronized`
3. `for(int i=1;i<=5;i++){`
4. `System.out.println(n*i);`



```
5.    try{
6.        Thread.sleep(400);
7.    } catch (Exception e){ System.out.println(e);}
8.    }
9.
10. }
11.}
12.
13.class MyThread1 extends Thread{
14.Table t;
15.MyThread1(Table t){
16.this.t=t;
17.}
18.public void run(){
19.t.printTable(5);
20.}
21.
22.}
23.class MyThread2 extends Thread{
24.Table t;
25.MyThread2(Table t){
26.this.t=t;
27.}
28.public void run(){
29.t.printTable(100);
30.}
31.}
32.
33.class TestSynchronization1 {
34.public static void main(String args[]){
35.Table obj = new Table();//only one object
36.MyThread1 t1=new MyThread1(obj);
37.MyThread2 t2=new MyThread2(obj);
38.t1.start();
```

```
39.t2.start();  
40.}  
41.}
```

**Output:**



```
5  
100  
10  
200  
15  
300  
20  
400  
25  
500
```

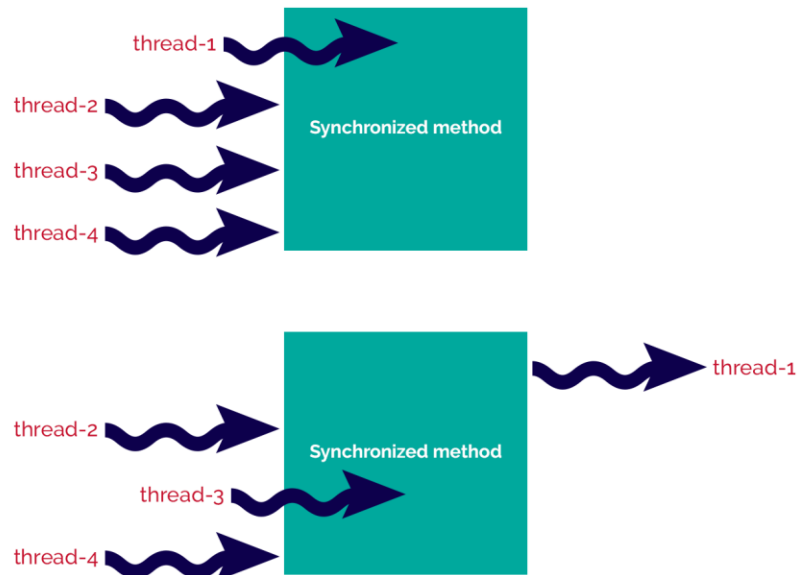
### Java Synchronized Method

If you declare any method as synchronized, it is known as synchronized method.

Synchronized method is used to lock an object for any shared resource.

When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

●●● Java thread execution with synchronized method



### TestSynchronization2.java

```
1. //example of java synchronized method
2. class Table{
3.     synchronized void printTable(int n){//synchronized method
4.         for(int i=1;i<=5;i++){
5.             System.out.println(n*i);
6.             try{
7.                 Thread.sleep(400);
8.             }catch(Exception e){System.out.println(e);}
9.         }
10.
11.     }
12. }
13.
14. class MyThread1 extends Thread{
15.     Table t;
```

```
16.MyThread1(Table t){
17.    this.t=t;
18.}
19.    public void run(){
20.        t.printTable(5);
21.    }
22.
23.}
24.    class MyThread2 extends Thread{
25.        Table t;
26.        MyThread2(Table t){
27.            this.t=t;
28.        }
29.        public void run(){
30.            t.printTable(100);
31.        }
32.    }
33.
34.    public class TestSynchronization2{
35.        public static void main(String args[]){
36.            Table obj = new Table();//only one object
37.            MyThread1 t1=new MyThread1(obj);
38.            MyThread2 t2=new MyThread2(obj);
39.            t1.start();
40.            t2.start();
41.        }
42.    }
```

### Output:

```
5
10
15
20
25
100
```

```
200
300
400
500
```

### Example of synchronized method by using anonymous class

In this program, we have created the two threads by using the anonymous class, so less coding is required.

#### TestSynchronization3.java

```
1. //Program of synchronized method by using anonymous class
2. class Table{
3.     synchronized void printTable(int n){//synchronized method
4.         for(int i=1;i<=5;i++){
5.             System.out.println(n*i);
6.             try{
7.                 Thread.sleep(400);
8.             }catch(Exception e){System.out.println(e);}
9.         }
10.
11.     }
12. }
13.
14. public class TestSynchronization3{
15.     public static void main(String args[]){
16.         final Table obj = new Table();//only one object
17.
18.         Thread t1=new Thread(){
19.             public void run(){
20.                 obj.printTable(5);
```

```
21.}  
22.};  
23.Thread t2=new Thread(){  
24.    public void run(){  
25.        obj.printTable(100);  
26.    }  
27.};  
28.  
29.t1.start();  
30.t2.start();  
31.}  
32.}
```

### Output:



### Synchronized Block in Java

Synchronized block can be used to perform synchronization on any specific resource of the method.

Suppose we have 50 lines of code in our method, but we want to synchronize only 5 lines, in such cases, we can use synchronized block.

If we put all the codes of the method in the synchronized block, it will work same as the synchronized method.

### Points to Remember

- Synchronized block is used to lock an object for any shared resource.
- Scope of synchronized block is smaller than the method.
- A Java synchronized block doesn't allow more than one JVM, to provide access control to a shared resource.
- The system performance may degrade because of the slower working of synchronized keyword.
- Java synchronized block is more efficient than Java synchronized method.

## Syntax

1. **synchronized** (object reference expression) {
2. **//code block**
3. }

## Example of Synchronized Block

Let's see the simple example of synchronized block.

### TestSynchronizedBlock1.java

1. **class** Table
2. {
3. **void** printTable(**int** n){
4. **synchronized(this){//synchronized block**
5. **for(int** i=**1**;i<=**5**;i++){
6. **System.out.println(n\*i);**
7. **try{**
8. **Thread.sleep(400);**
9. **}catch**(Exception e){**System.out.println(e);**}
10. }
11. }
12. **//end of the method**
13. }
- 14.

```
15.class MyThread1 extends Thread{
16.Table t;
17.MyThread1(Table t){
18.this.t=t;
19.}
20.public void run(){
21.t.printTable(5);
22.}
23.
24.}
25.class MyThread2 extends Thread{
26.Table t;
27.MyThread2(Table t){
28.this.t=t;
29.}
30.public void run(){
31.t.printTable(100);
32.}
33.
34.
35.public class TestSynchronizedBlock1{
36.public static void main(String args[]){
37.Table obj = new Table();//only one object
38.MyThread1 t1=new MyThread1(obj);
39.MyThread2 t2=new MyThread2(obj);
40.t1.start();
41.t2.start();
42.}
43.}
```

**Output:**

```
5
10
15
```



```
20
25
100
200
300
400
500
```

## Synchronized Block Example Using Anonymous Class

### TestSynchronizedBlock2.java

```
1. // A Sender class
2. class Sender
3. {
4.     public void SenderMsg(String msg)
5.     {
6.         System.out.println("\nSending a Message: " + msg);
7.         try
8.         {
9.             Thread.sleep(800);
10.        }
11.        catch (Exception e)
12.        {
13.            System.out.println("Thread interrupted.");
14.        }
15.        System.out.println("\n" + msg + "Sent");
16.    }
17.}
18.// A Sender class for sending a message using Threads
19.class SenderWThreads extends Thread
20.{
21.    private String msg;
22.    Sender sd;
23.
24.    // Receiver method to receive a message object and a message to be sent
```

```
25. SenderWThreads(String m, Sender obj)
26. {
27.     msg = m;
28.     sd = obj;
29. }
30.
31. public void run()
32. {
33.     // Checks that only one thread sends a message at a time.
34.     synchronized(sd)
35.     {
36.         // synchronizing the sender object
37.         sd.SenderMsg(msg);
38.     }
39. }
40.}
41.// Driver Code
42.public class ShynchronizedMultithreading
43.{
44.    public static void main(String args[])
45.    {
46.        Sender sender = new Sender();
47.        SenderWThreads sender1 = new SenderWThreads( "Hola " , sender);
48.        SenderWThreads sender2 = new SenderWThreads( "Welcome to DS " , sender);
49.
50.        // Start two threads of SenderWThreads type
51.        sender1.start();
52.        sender2.start();
53.
54.        // wait for threads to end
55.        try
56.        {
57.            sender1.join();
58.            sender2.join();
```

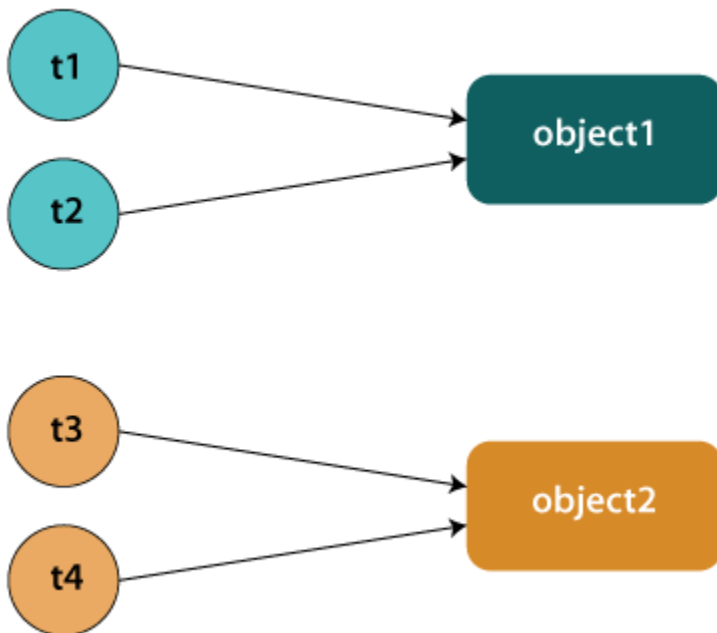
```
59. }  
60. catch(Exception e)  
61. {  
62.     System.out.println("Interrupted");  
63. }  
64. }  
65. }
```

### Output:

```
Sending a Message: Hola  
Hola Sent  
Sending a Message: Welcome to DS  
Welcome to DS Sent
```

### Static Synchronization

If you make any static method as synchronized, the lock will be on the class not on object.



### Problem without static synchronization

Suppose there are two objects of a shared class (e.g. Table) named object1 and object2. In case of synchronized method and synchronized block there cannot be interference between t1 and t2 or t3 and t4 because t1 and t2 both refers to a common object that have a single lock. But there can be interference between t1 and t3 or t2 and t4 because t1 acquires another lock and t3 acquires another lock. We don't want interference between t1 and t3 or t2 and t4. Static synchronization solves this problem.

### Example of Static Synchronization

In this example we have used **synchronized** keyword on the static method to perform static synchronization.

#### TestSynchronization4.java

```
1. class Table
2. {
3.     synchronized static void printTable(int n){
4.         for(int i=1;i<=10;i++){
5.             System.out.println(n*i);
6.             try{
7.                 Thread.sleep(400);
8.             }catch(Exception e){}
9.         }
10.    }
11.}
12.class MyThread1 extends Thread{
13.    public void run(){
14.        Table.printTable(1);
15.    }
16.}
17.class MyThread2 extends Thread{
```

```
18. public void run(){
19. Table.printTable(10);
20. }
21. }
22. class MyThread3 extends Thread{
23. public void run(){
24. Table.printTable(100);
25. }
26. }
27. class MyThread4 extends Thread{
28. public void run(){
29. Table.printTable(1000);
30. }
31. }
32. public class TestSynchronization4{
33. public static void main(String t[]){
34. MyThread1 t1=new MyThread1();
35. MyThread2 t2=new MyThread2();
36. MyThread3 t3=new MyThread3();
37. MyThread4 t4=new MyThread4();
38. t1.start();
39. t2.start();
40. t3.start();
41. t4.start();
42. }
43. }
```

**Output:**

```
1
2
3
4
```

```
5
6
7
8
9
10
10
20
30
40
50
60
70
80
90
100
100
200
300
400
500
600
700
800
900
1000
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
```

## Inter-thread Communication in Java

**Inter-thread communication** or **Co-operation** is all about allowing synchronized threads to communicate with each other.

Cooperation (Inter-thread communication) is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed. It is implemented by following methods of **Object class**:

- wait()
- notify()
- notifyAll()

### 1) wait() method

The wait() method causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.

The current thread must own this object's monitor, so it must be called from the synchronized method only otherwise it will throw exception.

Method	Description
public final void wait()throws InterruptedException	It waits until object is notified.
public final void wait(long timeout)throws InterruptedException	It waits for the specified amount of time.

### 2) notify() method

The notify() method wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation.

## Syntax:

1. **public final void** notify()

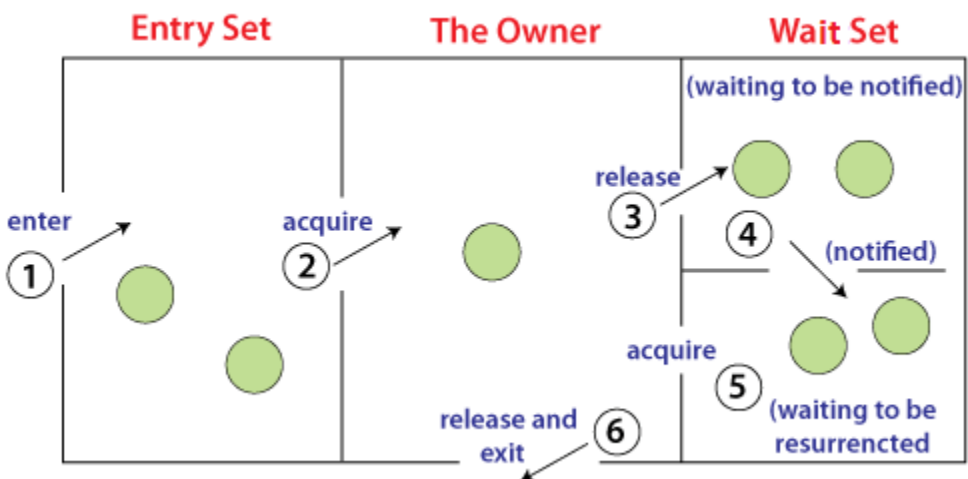
### 3) notifyAll() method

Wakes up all threads that are waiting on this object's monitor.

## Syntax:

1. **public final void** notifyAll()

Understanding the process of inter-thread communication





The point to point explanation of the above diagram is as follows:

1. Threads enter to acquire lock.
2. Lock is acquired by on thread.
3. Now thread goes to waiting state if you call wait() method on the object. Otherwise it releases the lock and exits.
4. If you call notify() or notifyAll() method, thread moves to the notified state (runnable state).
5. Now thread is available to acquire lock.
6. After completion of the task, thread releases the lock and exits the monitor state of the object.

Why wait(), notify() and notifyAll() methods are defined in Object class not Thread class?

It is because they are related to lock and object has a lock.

Difference between wait and sleep?

Let's see the important differences between wait and sleep methods.

wait()	sleep()
The wait() method releases the lock.	The sleep() method doesn't release the lock.
It is a method of Object class	It is a method of Thread class
It is the non-static method	It is the static method
It should be notified by notify() or notifyAll() methods	After the specified amount of time, sleep is completed.

Example of Inter Thread Communication in Java

Let's see the simple example of inter thread communication.

### Test.java

```
1. class Customer{
2.   int amount=10000;
3.
4.   synchronized void withdraw(int amount){
5.     System.out.println("going to withdraw...");
6.
7.     if(this.amount<amount){
8.       System.out.println("Less balance; waiting for deposit...");
9.       try{wait();}catch(Exception e){}
10.    }
11.    this.amount-=amount;
12.    System.out.println("withdraw completed...");
13.  }
14.
15.  synchronized void deposit(int amount){
16.    System.out.println("going to deposit...");
17.    this.amount+=amount;
18.    System.out.println("deposit completed... ");
19.    notify();
20.  }
21.  }
22.
23.  class Test{
24.    public static void main(String args[]){
25.      final Customer c=new Customer();
```

```
26.new Thread(){
27. public void run(){c.withdraw(15000);}
28. }.start();
29.new Thread(){
30. public void run(){c.deposit(10000);}
31. }.start();
32.
33.}}
```

**Output:**

```
going to withdraw...
Less balance; waiting for deposit...
going to deposit...
deposit completed...
withdraw completed
```