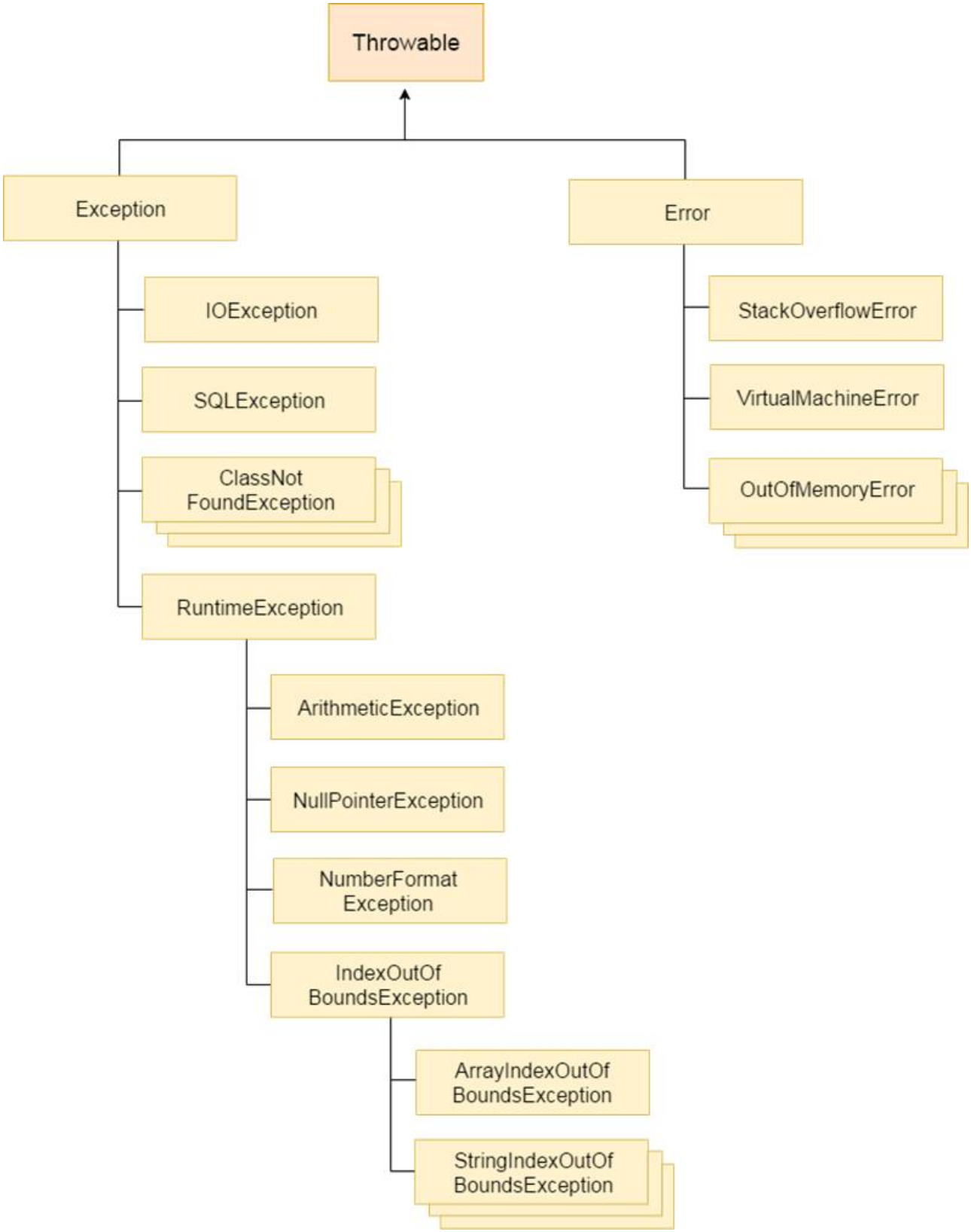
Exception Handling in Java

The **Exception Handling in Java** is one of the powerful *mechanism to handle the runtime errors* so that normal flow of the application can be maintained.

What is Exception in Java

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


Throwable
Exception
Error
IOException
SQLException
ClassNot FoundException
RuntimeException
StackOverflowError
VirtualMachineError
OutOfMemoryError
ArithmeticException
NullPointerException
NumberFormat Exception
IndexOutOf BoundsException
ArrayIndexOutOf BoundsException
StringIndexOutOf BoundsException

What is Exception Handling

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

Advantage of Exception Handling

The core advantage of exception handling is **to maintain the normal flow of the application**. An exception normally disrupts the normal flow of the application that is why we use exception handling. Let's take a scenario:

1. statement 1;
2. statement 2;
3. statement 3;
4. statement 4;
5. statement 5;//exception occurs
6. statement 6;
7. statement 7;
8. statement 8;
9. statement 9;
10. statement 10;

Suppose there are 10 statements in your program and there occurs an exception at statement 5, the rest of the code will not be executed i.e. statement 6 to 10 will not be executed. If we perform exception handling, the rest of the statement will be executed. That is why we use exception handling in Java.

Hierarchy of Java Exception classes

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

Types of Java Exceptions

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

1. Checked Exception
2. Unchecked Exception
3. Error

Difference between Checked and Unchecked Exceptions

1) Checked Exception

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

2) Unchecked Exception

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

3) Error

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

Java Exception Keywords

There are 5 keywords which are used in handling exceptions in Java.

Keyword	Description
try	The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. It means, we can't use try block alone.
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 important 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 doesn't throw an exception. It specifies that there may occur an exception in the method. It is always used with method signature.

Java Exception Handling Example

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

```
1.  public class JavaExceptionExample{
2.    public static void main(String args[]){
3.     try{
4.        //code that may raise exception
5.        int data=100/0;
6.     }catch(ArithmeticException e){System.out.println(e);}
7.     //rest code of the program
8.     System.out.println("rest of the code...");
9.    }
10. }
```

Output:

```
Exception in thread main java.lang.ArithmeticException:/ by zero
rest of the code...
```

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

Common Scenarios of Java Exceptions

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

1) A scenario where ArithmeticException occurs

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

```
1.  int a=50/0;//ArithmeticException
```

2) A scenario where NullPointerException occurs

If we have a null value in any variable, performing any operation on the variable throws a NullPointerException.

```
1. String s=null;
2. System.out.println(s.length());//NullPointerException
```

---

3) A scenario where NumberFormatException occurs

The wrong formatting of any value may occur NumberFormatException. Suppose I have a string variable that has characters, converting this variable into digit will occur NumberFormatException.

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

---

4) A scenario where ArrayIndexOutOfBoundsException occurs

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

```
1. int a[]=new int[5];
2. a[10]=50; //ArrayIndexOutOfBoundsException
```

Java try-catch block

Java try block

Java **try** block is used to enclose the code that might throw an exception. It must be used within the method.

If an exception occurs at the particular statement of try block, the rest of the block code will not execute. So, it is recommended not to keeping the code in try block that will not throw an exception.

Java try block must be followed by either catch or finally block.

Syntax of Java try-catch

```
1. try{
2. //code that may throw an exception
3. }catch(Exception_class_Name ref){}
```

Syntax of try-finally block

```
1. try{
2. //code that may throw an exception
3. }finally{}
```

Java catch block

Java catch block is used to handle the Exception by declaring the type of exception within the parameter. The declared exception must be the parent class exception (i.e., Exception) or the generated exception type. However, the good approach is to declare the generated type of exception.

The catch block must be used after the try block only. You can use multiple catch block with a single try block.

Problem without exception handling

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

Example 1

```
public class TryCatchExample1 {
     public static void main(String[] args) {
          int data=50/0; //may throw exception
              System.out.println("rest of the code");

    }

}
```

Test it Now

Output:

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

As displayed in the above example, the **rest of the code** is not executed (in such case, the **rest of the code** statement is not printed).

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

Solution by exception handling

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

Example 2

```
public class TryCatchExample2 {

     public static void main(String[] args) {

       try

       {

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

       }

         //handling the exception

       catch(ArithmeticException e)

       {

          System.out.println(e);

       }

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

    }
```

```
}
```

Output:

```
java.lang.ArithmeticException: / by zero
rest of the code
```

Now, as displayed in the above example, the **rest of the code** is executed, i.e., the **rest of the code** statement is printed.

Example 3

In this example, we also kept the code in a try block that will not throw an exception.

```
public class TryCatchExample3 {
    public static void main(String[] args) {
      try
      {
      int data=50/0; //may throw exception
                  // if exception occurs, the remaining statement will not exceute
      System.out.println("rest of the code");
      }
          // handling the exception
      catch(ArithmeticException e)
      {
         System.out.println(e);
      }

   }

}
```

Output:

```
java.lang.ArithmeticException: / by zero
```

Here, we can see that if an exception occurs in the try block, the rest of the block code will not execute.

Example 4

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

```
public class TryCatchExample4 {

   public static void main(String[] args) {
      try
      {
      int data=50/0; //may throw exception
      }
         // handling the exception by using Exception class
      catch(Exception e)
      {
         System.out.println(e);
      }
      System.out.println("rest of the code");
   }

}
```

Output:

```
java.lang.ArithmeticException: / by zero
rest of the code
```

Example 5

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

```
public class TryCatchExample5 {

   public static void main(String[] args) {
      try
      {
      int data=50/0; //may throw exception
      }
          // handling the exception
      catch(Exception e)
      {
              // displaying the custom message
         System.out.println("Can't divided by zero");
      }
   }

}
```

Output:

```
Can't divided by zero
```

Example 6

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

```
public class TryCatchExample6 {

   public static void main(String[] args) {
      int i=50;
      int j=0;
      int data;
      try
      {
      data=i/j; //may throw exception
      }
         // handling the exception
      catch(Exception e)
      {
          // resolving the exception in catch block
         System.out.println(i/(j+2));
      }
   }
}
```

Output:

```
25
```

Example 7

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

```
public class TryCatchExample7 {

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

```
        try
        {
        int data1=50/0; //may throw exception

        }
            // handling the exception
        catch(Exception e)
        {
           // generating the exception in catch block
        int data2=50/0; //may throw exception

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

Output:

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

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

Example 8

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

```
  public class TryCatchExample8 {

      public static void main(String[] args) {
        try
        {
        int data=50/0; //may throw exception

        }
           // try to handle the ArithmeticException using ArrayIndexOutOfBoundsException
        catch(ArrayIndexOutOfBoundsException e)
        {
           System.out.println(e);
        }
        System.out.println("rest of the code");
      }

  }
```

Output:

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

Example 9

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

```
  public class TryCatchExample9 {

      public static void main(String[] args) {
        try
        {
        int arr[]= {1,3,5,7};
        System.out.println(arr[10]); //may throw exception
        }
```

```
            // handling the array exception
        catch(ArrayIndexOutOfBoundsException e)
        {
            System.out.println(e);
        }
        System.out.println("rest of the code");
    }

}
```

Output:

```
java.lang.ArrayIndexOutOfBoundsException: 10
rest of the code
```

Example 10

Let's see an example to handle checked exception.

```
import java.io.FileNotFoundException;
import java.io.PrintWriter;

public class TryCatchExample10 {

    public static void main(String[] args) {

        PrintWriter pw;
        try {
            pw = new PrintWriter("jtp.txt"); //may throw exception
            pw.println("saved");
        }
// providing the checked exception handler
 catch (FileNotFoundException e) {

            System.out.println(e);
        }
    System.out.println("File saved successfully");
    }
}
```

Output:

```
File saved successfully
```

Internal working of java try-catch block

The JVM firstly checks whether the exception is handled or not. If exception is not handled, JVM provides a default exception handler that performs the following tasks:

- Prints out exception description.
- Prints the stack trace (Hierarchy of methods where the exception occurred).
- Causes the program to terminate.

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

Java catch multiple exceptions

Java Multi-catch block

A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block.

Points to remember

- At a time only one exception occurs and at a time only one catch block is executed.
- All catch blocks must be ordered from most specific to most general, i.e. catch for ArithmeticException must come before catch for Exception.

Example 1

Let's see a simple example of java multi-catch block.

```
public class MultipleCatchBlock1 {

   public static void main(String[] args) {

        try{
            int a[]=new int[5];
            a[5]=30/0;
           }
           catch(ArithmeticException e)
              {
               System.out.println("Arithmetic Exception occurs");
              }
           catch(ArrayIndexOutOfBoundsException e)
              {
               System.out.println("ArrayIndexOutOfBounds Exception occurs");
              }
           catch(Exception e)
              {
               System.out.println("Parent Exception occurs");
              }
           System.out.println("rest of the code");
   }
}
```

Output:

```
Arithmetic Exception occurs
rest of the code
```

Example 2

```
public class MultipleCatchBlock2 {

   public static void main(String[] args) {

        try{
            int a[]=new int[5];

            System.out.println(a[10]);
           }
           catch(ArithmeticException e)
              {
               System.out.println("Arithmetic Exception occurs");
```

```
            }
          catch(ArrayIndexOutOfBoundsException e)
            {
             System.out.println("ArrayIndexOutOfBounds Exception occurs");
            }
          catch(Exception e)
            {
             System.out.println("Parent Exception occurs");
            }
          System.out.println("rest of the code");
    }
}
```

Output:

```
ArrayIndexOutOfBounds Exception occurs
rest of the code
```

Example 3

In this example, try block contains two exceptions. But at a time only one exception occurs and its corresponding catch block is invoked.

```
public class MultipleCatchBlock3 {

   public static void main(String[] args) {

       try{
           int a[]=new int[5];
           a[5]=30/0;
           System.out.println(a[10]);
          }
          catch(ArithmeticException e)
            {
             System.out.println("Arithmetic Exception occurs");
            }
          catch(ArrayIndexOutOfBoundsException e)
            {
             System.out.println("ArrayIndexOutOfBounds Exception occurs");
            }
          catch(Exception e)
            {
             System.out.println("Parent Exception occurs");
            }
          System.out.println("rest of the code");
    }
}
```

Output:

```
Arithmetic Exception occurs
rest of the code
```

Example 4

In this example, we generate NullPointerException, but didn't provide the corresponding exception type. In such case, the catch block containing the parent exception class **Exception** will invoked.

```
public class MultipleCatchBlock4 {
```

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

        try{
            String s=null;
            System.out.println(s.length());
           }
           catch(ArithmeticException e)
              {
               System.out.println("Arithmetic Exception occurs");
              }
           catch(ArrayIndexOutOfBoundsException e)
              {
               System.out.println("ArrayIndexOutOfBounds Exception occurs");
              }
           catch(Exception e)
              {
               System.out.println("Parent Exception occurs");
              }
           System.out.println("rest of the code");
   }
}
```

Output:

```
Parent Exception occurs
rest of the code
```

Example 5

Let's see an example, to handle the exception without maintaining the order of exceptions (i.e. from most specific to most general).

```
class MultipleCatchBlock5{
  public static void main(String args[]){
   try{
    int a[]=new int[5];
    a[5]=30/0;
   }
   catch(Exception e){System.out.println("common task completed");}
   catch(ArithmeticException e){System.out.println("task1 is completed");}
   catch(ArrayIndexOutOfBoundsException e){System.out.println("task 2 completed");}
   System.out.println("rest of the code...");
 }
}
```

Output:

```
Compile-time error
```

Java Nested try block

The try block within a try block is known as nested try block in java.

Why use nested try block

Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested.

Syntax:

```
1.  ....

try
{
   statement 1;
   statement 2;
   try
   {
      statement 1;
      statement 2;
   }
   catch(Exception e)
   {
   }
}
catch(Exception e)
{
}
```

Java nested try example

Let's see a simple example of java nested try block.

```
class Excep6{

 public static void main(String args[]){

  try{

    try{

     System.out.println("going to divide");

     int b =39/0;

    }catch(ArithmeticException e){System.out.println(e);}

    try{

    int a[]=new int[5];

    a[5]=4;

    }catch(ArrayIndexOutOfBoundsException e){System.out.println(e);}



    System.out.println("other statement);

  }catch(Exception e){System.out.println("handeled");}
```

```
 System.out.println("normal flow..");

 }

}
```

Java finally block

Java finally block is a block that is used *to execute important code* such as closing connection, stream etc.

Java finally block is always executed whether exception is handled or not.

Java finally block follows try or catch block.

Note: If you don't handle exception, before terminating the program, JVM executes finally block(if any).

Why use java finally

- Finally block in java can be used to put "cleanup" code such as closing a file, closing connection etc.

Usage of Java finally

Let's see the different cases where java finally block can be used.

Case 1

Let's see the java finally example where **exception doesn't occur**.

```
1.  class TestFinallyBlock{
2.    public static void main(String args[]){
3.    try{
4.     int data=25/5;
5.     System.out.println(data);
6.    }
7.    catch(NullPointerException e){System.out.println(e);}
8.    finally{System.out.println("finally block is always executed");}
9.    System.out.println("rest of the code...");
10.   }
11. }
```

```
Output:5
       finally block is always executed
       rest of the code...
```

Case 2

Let's see the java finally example where **exception occurs and not handled**.

```
class TestFinallyBlock1{
  public static void main(String args[]){
  try{
   int data=25/0;
   System.out.println(data);
  }
  catch(NullPointerException e){System.out.println(e);}
  finally{System.out.println("finally block is always executed");}
```

```
 System.out.println("rest of the code...");
 }
}
Output:finally block is always executed
    Exception in thread main java.lang.ArithmeticException:/ by zero
```

Case 3

Let's see the java finally example where **exception occurs and handled**.

```
public class TestFinallyBlock2{

 public static void main(String args[]){

 try{

  int data=25/0;

  System.out.println(data);

 }

 catch(ArithmeticException e){System.out.println(e);}

 finally{System.out.println("finally block is always executed");}

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

 }

}
```

```
Output:Exception in thread main java.lang.ArithmeticException:/ by zero
       finally block is always executed
       rest of the code...
```

Rule: For each try block there can be zero or more catch blocks, but only one finally block.

Note: The finally block will not be executed if program exits(either by calling System.exit() or by causing a fatal error that causes the process to abort).

Java throw exception

Java throw keyword

The Java throw keyword is used to explicitly throw an exception.

We can throw either checked or uncheked exception in java by throw keyword. The throw keyword is mainly used to throw custom exception. We will see custom exceptions later.

The syntax of java throw keyword is given below.

1. throw exception;

Let's see the example of throw IOException.

1. throw new IOException("sorry device error);

java throw keyword example

In this example, we have created the validate method that takes integer value as a parameter. If the age is less than 18, we are throwing the ArithmeticException otherwise print a message welcome to vote.

```
public class TestThrow1{
  static void validate(int age){
    if(age<18)
     throw new ArithmeticException("not valid");
    else
     System.out.println("welcome to vote");
  }
  public static void main(String args[]){
    validate(13);
    System.out.println("rest of the code...");
  }
}
```

Output:

```
Exception in thread main java.lang.ArithmeticException:not valid
```

Java throws keyword

The **Java throws keyword** is used to declare an exception. It gives information to the programmer that there may occur an exception so it is better for the programmer to provide the exception handling code so that normal flow can be maintained.

Exception Handling is mainly used to handle the checked exceptions. If there occurs any unchecked exception such as NullPointerException, it is programmers fault that he is not performing check up before the code being used.

Syntax of java throws

```
1. return_type method_name() throws exception_class_name{
2. //method code
3. }
```

Which exception should be declared

Ans) checked exception only, because:

- **unchecked Exception:** under your control so correct your code.
- **error:** beyond your control e.g. you are unable to do anything if there occurs VirtualMachineError or StackOverflowError.

Advantage of Java throws keyword

Now Checked Exception can be propagated (forwarded in call stack).

It provides information to the caller of the method about the exception.

Java throws example

Let's see the example of java throws clause which describes that checked exceptions can be propagated by throws keyword.

```
import java.io.IOException;

class Testthrows1{
```

```
  void m()throws IOException{
    throw new IOException("device error");//checked exception
  }
  void n()throws IOException{
    m();
  }
  void p(){
   try{
    n();
   }catch(Exception e){System.out.println("exception handled");}
  }
  public static void main(String args[]){
   Testthrows1 obj=new Testthrows1();
   obj.p();
   System.out.println("normal flow...");
  }
}
```

Output:

```
exception handled
normal flow...
```

Rule: If you are calling a method that declares an exception, you must either caught or declare the exception.

There are two cases:

1. **Case1:**You caught the exception i.e. handle the exception using try/catch.
2. **Case2:**You declare the exception i.e. specifying throws with the method.

Case1: You handle the exception

- In case you handle the exception, the code will be executed fine whether exception occurs during the program or not.

```
import java.io.*;
class M{
 void method()throws IOException{
```

```
 throw new IOException("device error");
 }
}
public class Testthrows2{
  public static void main(String args[]){
   try{
    M m=new M();
    m.method();
   }catch(Exception e){System.out.println("exception handled");}

   System.out.println("normal flow...");
 }
}
```

Test it Now

```
Output:exception handled
       normal flow...
```

Case2: You declare the exception

- A)In case you declare the exception, if exception does not occur, the code will be executed fine.
- B)In case you declare the exception if exception occurs, an exception will be thrown at runtime because throws does not handle the exception.

A)Program if exception does not occur

```
1.  import java.io.*;
2.  class M{
3.   void method()throws IOException{
4.    System.out.println("device operation performed");
5.   }
6.  }
7.  class Testthrows3{
8.    public static void main(String args[])throws IOException{//declare exception
9.      M m=new M();
10.     m.method();
11.
12.    System.out.println("normal flow...");
13.  }
14. }
```

Test it Now

```
Output:device operation performed
       normal flow...
```

B)Program if exception occurs

```
import java.io.*;
class M{
 void method()throws IOException{
  throw new IOException("device error");
 }
}
class Testthrows4{
  public static void main(String args[])throws IOException{//declare exception
    M m=new M();
    m.method();
     System.out.println("normal flow...");
 }
}
```

```
Output:Runtime Exception
```

Que) Can we rethrow an exception?

Yes, by throwing same exception in catch block.

Difference between throw and throws in Java

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

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

Java throw example

```
1. void m(){
2. throw new ArithmeticException("sorry");
3. }
```

Java throws example

```
1. void m()throws ArithmeticException{
2. //method code
3. }
```

Java throw and throws example

```
1. void m()throws ArithmeticException{
2. throw new ArithmeticException("sorry");
3. }
```

Difference between final, finally and finalize

There are many differences between final, finally and finalize. A list of differences between final, finally and finalize are given below:

No.	final	finally	finalize
1)	Final is used to apply restrictions on class, method and variable. Final class can't be inherited, final method can't be overridden and final variable value can't be changed.	Finally is used to place important code, it will be executed whether exception is handled or not.	Finalize is used to perform clean up processing just before object is garbage collected.
2)	Final is a keyword.	Finally is a block.	Finalize is a method.

Java final example

```
class FinalExample{
public static void main(String[] args){
final int x=100;
x=200;//Compile Time Error
}}
```

Java finally example

```
1. class FinallyExample{
2. public static void main(String[] args){
3. try{
4. int x=300;
5. }catch(Exception e){System.out.println(e);}
6. finally{System.out.println("finally block is executed");}
7. }}
```

Java finalize example

```
class FinalizeExample{
public void finalize(){System.out.println("finalize called");}
public static void main(String[] args){
FinalizeExample f1=new FinalizeExample();
FinalizeExample f2=new FinalizeExample();
f1=null;
f2=null;
System.gc();
}}
```

ExceptionHandling with MethodOverriding in Java

There are many rules if we talk about methodoverriding with exception handling. The Rules are as follows:

- **If the superclass method does not declare an exception**
  - If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but it can declare unchecked exception.
- **If the superclass method declares an exception**
  - If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception.

If the superclass method does not declare an exception

1) Rule: If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception.

```
import java.io.*;
class Parent{
  void msg(){System.out.println("parent");}
}

class TestExceptionChild extends Parent{
  void msg()throws IOException{
    System.out.println("TestExceptionChild");
  }
  public static void main(String args[]){
   Parent p=new TestExceptionChild();
   p.msg();
  }
}
```

Test it Now

```
Output:Compile Time Error
```

2) Rule: If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but can declare unchecked exception.

```
import java.io.*;
class Parent{
  void msg(){System.out.println("parent");}
}

class TestExceptionChild1 extends Parent{
  void msg()throws ArithmeticException{
    System.out.println("child");
  }
  public static void main(String args[]){
   Parent p=new TestExceptionChild1();
   p.msg();
  }
```

```
    }
Output:child
```

If the superclass method declares an exception

1) Rule: If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception.

Example in case subclass overridden method declares parent exception

```
    import java.io.*;
    class Parent{
      void msg()throws ArithmeticException{System.out.println("parent");}
    }

    class TestExceptionChild2 extends Parent{
      void msg()throws Exception{System.out.println("child");}

      public static void main(String args[]){
       Parent p=new TestExceptionChild2();
       try{
       p.msg();
       }catch(Exception e){}
      }
    }
Output:Compile Time Error
```

Example in case subclass overridden method declares same exception

```
    import java.io.*;
    class Parent{
      void msg()throws Exception{System.out.println("parent");}
    }

    class TestExceptionChild3 extends Parent{
      void msg()throws Exception{System.out.println("child");}

      public static void main(String args[]){
       Parent p=new TestExceptionChild3();
       try{
       p.msg();
       }catch(Exception e){}
      }
    }
Output:child
```

Example in case subclass overridden method declares subclass exception

```
    import java.io.*;
    class Parent{
      void msg()throws Exception{System.out.println("parent");}
    }

    class TestExceptionChild4 extends Parent{
      void msg()throws ArithmeticException{System.out.println("child");}

      public static void main(String args[]){
       Parent p=new TestExceptionChild4();
       try{
       p.msg();
```

```
      }catch(Exception e){}
     }
    }
Output:child
```

Example in case subclass overridden method declares no exception

```
    import java.io.*;
    class Parent{
     void msg()throws Exception{System.out.println("parent");}
    }

    class TestExceptionChild5 extends Parent{
     void msg(){System.out.println("child");}

     public static void main(String args[]){
      Parent p=new TestExceptionChild5();
      try{
      p.msg();
      }catch(Exception e){}
     }
    }
Output:child
```

Java Custom Exception

If you are creating your own Exception that is known as custom exception or user-defined exception. Java custom exceptions are used to customize the exception according to user need.

By the help of custom exception, you can have your own exception and message.

Let's see a simple example of java custom exception.

```
    class InvalidAgeException extends Exception{
     InvalidAgeException(String s){
      super(s);
     }
    }
    class TestCustomException1{

      static void validate(int age)throws InvalidAgeException{
       if(age<18)
        throw new InvalidAgeException("not valid");
       else
        System.out.println("welcome to vote");
      }

      public static void main(String args[]){
        try{
        validate(13);
        }catch(Exception m){System.out.println("Exception occured: "+m);}

        System.out.println("rest of the code...");
      }
    }
Output:Exception occured: InvalidAgeException:not valid
    rest of the code...
```

Additional Notes

Java Exception propagation

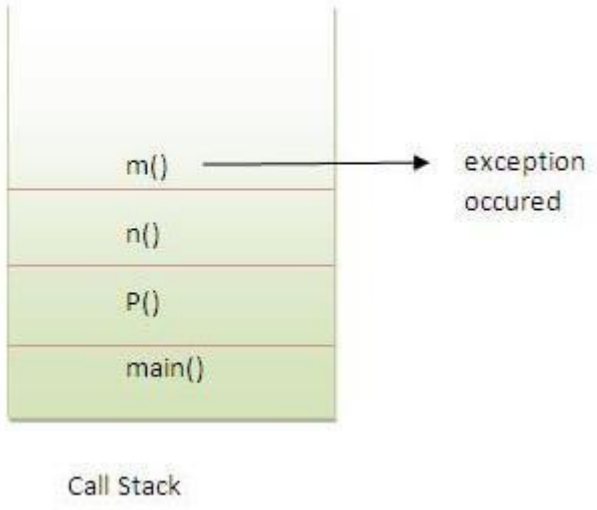
An exception is first thrown from the top of the stack and if it is not caught, it drops down the call stack to the previous method,If not caught there, the exception again drops down to the previous method, and so on until they are caught or until they reach the very bottom of the call stack.This is called exception propagation.

Rule: By default Unchecked Exceptions are forwarded in calling chain (propagated).

Program of Exception Propagation

```
class TestExceptionPropagation1{
  void m(){
    int data=50/0;
  }
  void n(){
    m();
  }
  void p(){
   try{
    n();
   }catch(Exception e){System.out.println("exception handled");}
  }
  public static void main(String args[]){
   TestExceptionPropagation1 obj=new TestExceptionPropagation1();
   obj.p();
   System.out.println("normal flow...");
  }
}
```

```
Output:exception handled
       normal flow...
```



In the above example exception occurs in m() method where it is not handled,so it is propagated to previous n() method where it is not handled, again it is propagated to p() method where exception is handled.

Exception can be handled in any method in call stack either in main() method,p() method,n() method or m() method.

Rule: By default, Checked Exceptions are not forwarded in calling chain (propagated).

Program which describes that checked exceptions are not propagated

```
class TestExceptionPropagation2{
  void m(){
    throw new java.io.IOException("device error");//checked exception
  }
  void n(){
```

```
  m();
 }
 void p(){
  try{
   n();
  }catch(Exception e){System.out.println("exception handeled");}
 }
 public static void main(String args[]){
  TestExceptionPropagation2 obj=new TestExceptionPropagation2();
  obj.p();
  System.out.println("normal flow");
 }
}
Output:Compile Time Error
```

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

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

---

2) 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:

public final void notify()

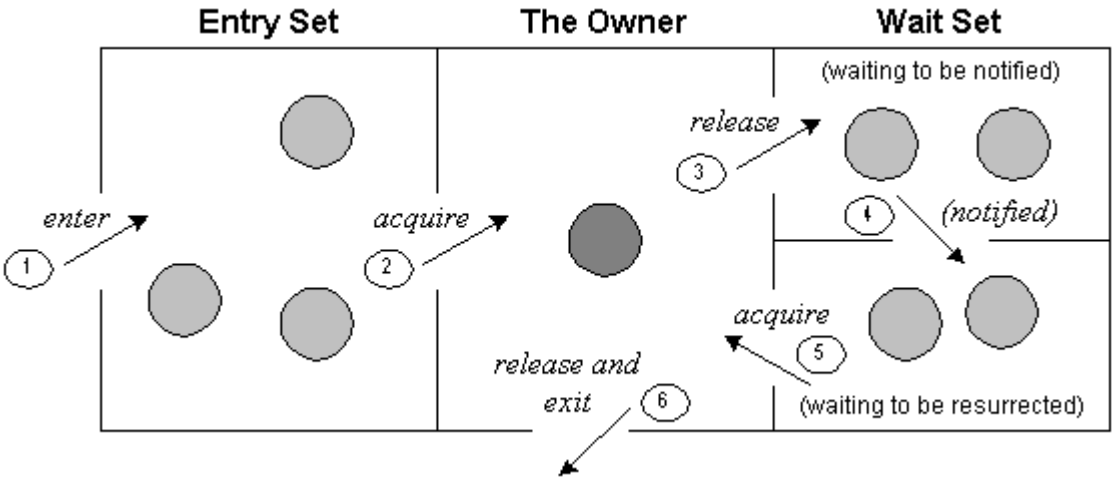
---

3) notifyAll() method

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

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()
wait() method releases the lock	sleep() method doesn't release the lock.
is the method of Object class	is the method of Thread class
is the non-static method.	is the static method
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.

```
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
```

Interrupting a Thread:

If any thread is in sleeping or waiting state (i.e. sleep() or wait() is invoked), calling the interrupt() method on the thread, breaks out the sleeping or waiting state throwing InterruptedException. If the thread is not in the sleeping or waiting state, calling the interrupt() method performs normal behaviour and doesn't interrupt the thread but sets the interrupt flag to true. Let's first see the methods provided by the Thread class for thread interruption.

The 3 methods provided by the Thread class for interrupting a thread

- **public void interrupt()**
- **public static boolean interrupted()**
- **public boolean isInterrupted()**

Example of interrupting a thread that stops working

In this example, after interrupting the thread, we are propagating it, so it will stop working. If we don't want to stop the thread, we can handle it where sleep() or wait() method is invoked. Let's first see the example where we are propagating the exception.

```
1.  class TestInterruptingThread1 extends Thread{
2.  public void run(){
3.  try{
4.  Thread.sleep(1000);
5.  System.out.println("task");
6.  }catch(InterruptedException e){
7.  throw new RuntimeException("Thread interrupted..."+e);
8.  }
```

```
9.
10. }
11.
12. public static void main(String args[]){
13. TestInterruptingThread1 t1=new TestInterruptingThread1();
14. t1.start();
15. try{
16. t1.interrupt();
17. }catch(Exception e){System.out.println("Exception handled "+e);}
18.
19. }
20. }
```

Test it Now
download this example

```
Output:Exception in thread-0
       java.lang.RuntimeException: Thread interrupted...
       java.lang.InterruptedException: sleep interrupted
       at A.run(A.java:7)
```

Example of interrupting a thread that doesn't stop working

In this example, after interrupting the thread, we handle the exception, so it will break out the sleeping but will not stop working.

```
1. class TestInterruptingThread2 extends Thread{
2. public void run(){
3. try{
4. Thread.sleep(1000);
5. System.out.println("task");
6. }catch(InterruptedException e){
7. System.out.println("Exception handled "+e);
8. }
9. System.out.println("thread is running...");
10. }
11.
12. public static void main(String args[]){
13. TestInterruptingThread2 t1=new TestInterruptingThread2();
14. t1.start();
15.
16. t1.interrupt();
17.
18. }
19. }
```

Test it Now
download this example

```
Output:Exception handled
       java.lang.InterruptedException: sleep interrupted
       thread is running...
```

Example of interrupting thread that behaves normally

If thread is not in sleeping or waiting state, calling the interrupt() method sets the interrupted flag to true that can be used to stop the thread by the java programmer later.

```
1. class TestInterruptingThread3 extends Thread{
2.
3. public void run(){
4. for(int i=1;i<=5;i++)
5. System.out.println(i);
6. }
```

```
7.
8.  public static void main(String args[]){
9.  TestInterruptingThread3 t1=new TestInterruptingThread3();
10. t1.start();
11.
12. t1.interrupt();
13.
14. }
15. }
```

Test it Now

```
Output:1
       2
       3
       4
       5
```

What about isInterrupted and interrupted method?

The isInterrupted() method returns the interrupted flag either true or false. The static interrupted() method returns the interrupted flag afterthat it sets the flag to false if it is true.

```
1.  public class TestInterruptingThread4 extends Thread{
2.  public void run(){
3.  for(int i=1;i<=2;i++){
4.  if(Thread.interrupted()){
5.  System.out.println("code for interrupted thread");
6.  }
7.  else{
8.  System.out.println("code for normal thread");
9.  }
10.
11. }//end of for loop
12. }
13.
14. public static void main(String args[]){
15.
16. TestInterruptingThread4 t1=new TestInterruptingThread4();
17. TestInterruptingThread4 t2=new TestInterruptingThread4();
18.
19. t1.start();
20. t1.interrupt();
21.
22. t2.start();
23.
24. }
25. }
```

Test it Now

```
Output:Code for interrupted thread
       code for normal thread
       code for normal thread
       code for normal thread
```

Multithreading in Java

Multithreading in java is a process of executing multiple threads simultaneously.

A thread is a lightweight sub-process, the smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.

However, we use multithreading than multiprocessing because threads use a shared memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.

Java Multithreading is mostly used in games, animation, etc.

Advantages of Java Multithreading

1) It **doesn't block the user** because threads are independent and you can perform multiple operations at the same time.

2) You **can perform many operations together, so it saves time**.

3) Threads are **independent**, so it doesn't affect other threads if an exception occurs in a single thread.

Multitasking

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved in two ways:

- Process-based Multitasking (Multiprocessing)
- Thread-based Multitasking (Multithreading)

1) Process-based Multitasking (Multiprocessing)

- Each process has an address in memory. In other words, each process allocates a separate memory area.
- A process is heavyweight.
- Cost of communication between the process is high.
- Switching from one process to another requires some time for saving and loading registers, memory maps, updating lists, etc.

2) Thread-based Multitasking (Multithreading)

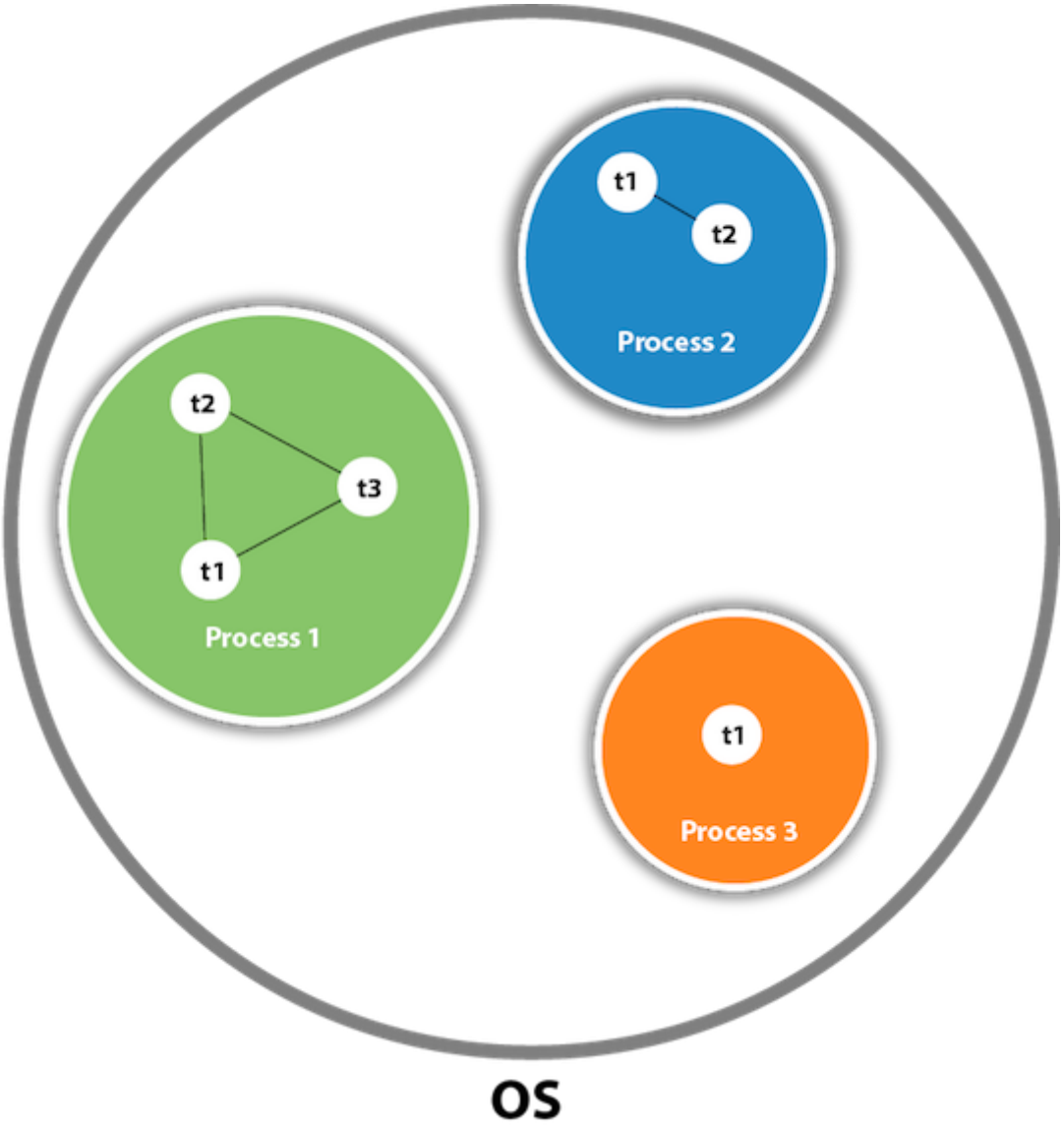
- Threads share the same address space.
- A thread is lightweight.
- Cost of communication between the thread is low.

Note: At least one process is required for each thread.

What is Thread in java

A thread is a lightweight subprocess, the smallest unit of processing. It is a separate path of execution.

Threads are independent. If there occurs exception in one thread, it doesn't affect other threads. It uses a shared memory area.



As shown in the above figure, a thread is executed inside the process. There is context-switching between the threads. There can be multiple processes inside the OS, and one process can have multiple threads.

Note: At a time one thread is executed only.

Java Thread class

Java provides **Thread class** to achieve thread programming. Thread class provides constructors and methods to create and perform operations on a thread. Thread class extends Object class and implements Runnable interface.

Java Thread Methods

S.N.	Modifier and Type	Method	Description
1)	void	start()	It is used to start the execution of the thread.
2)	void	run()	It is used to do an action for a thread.
3)	static void	sleep()	It sleeps a thread for the specified amount of time.
4)	static Thread	currentThread()	It returns a reference to the currently executing thread object.
5)	void	join()	It waits for a thread to die.

6)	int	getPriority()	It returns the priority of the thread.
7)	void	setPriority()	It changes the priority of the thread.
8)	String	getName()	It returns the name of the thread.
9)	void	setName()	It changes the name of the thread.
10)	long	getId()	It returns the id of the thread.
11)	boolean	isAlive()	It tests if the thread is alive.
12)	static void	yield()	It causes the currently executing thread object to pause and allow other threads to execute temporarily.
13)	void	suspend()	It is used to suspend the thread.
14)	void	resume()	It is used to resume the suspended thread.
15)	void	stop()	It is used to stop the thread.
16)	void	destroy()	It is used to destroy the thread group and all of its subgroups.
17)	Boolean	isDaemon()	It tests if the thread is a daemon thread.
18)	void	setDaemon()	It marks the thread as daemon or user thread.
19)	void	interrupt()	It interrupts the thread.
20)	Boolean	isinterrupted()	It tests whether the thread has been interrupted.
21)	static Boolean	interrupted()	It tests whether the current thread has been interrupted.
22)	static int	activeCount()	It returns the number of active threads in the current thread's thread group.
23)	void	checkAccess()	It determines if the currently running thread has permission to modify the thread.
24)	static Boolean	holdLock()	It returns true if and only if the current thread holds the monitor lock on the specified object.
25)	static void	dumpStack()	It is used to print a stack trace of the current thread to the standard error stream.
26)	StackTraceElement[]	getStackTrace()	It returns an array of stack trace elements representing the stack dump of the thread.
27)	static int	enumerate()	It is used to copy every active thread's thread group and its subgroup into the specified array.
28)	Thread.State	getState()	It is used to return the state of the thread.
29)	ThreadGroup	getThreadGroup()	It is used to return the thread group to which this thread belongs
30)	String	toString()	It is used to return a string representation of this thread, including the thread's name, priority, and thread group.
31)	void	notify()	It is used to give the notification for only one thread which is waiting for a particular object.
32)	void	notifyAll()	It is used to give the notification to all waiting threads of a particular object.
33)	void	setContextClassLoader()	It sets the context ClassLoader for the Thread.
34)	ClassLoader	getContextClassLoader()	It returns the context ClassLoader for the thread.
35)	static Thread.UncaughtExceptionHandler	getDefaultUncaughtExceptionHandler()	It returns the default handler invoked when a thread abruptly terminates due to an uncaught exception.
36)	static void	setDefaultUncaughtExceptionHandler()	It sets the default handler invoked when a thread abruptly terminates due to an uncaught exception.

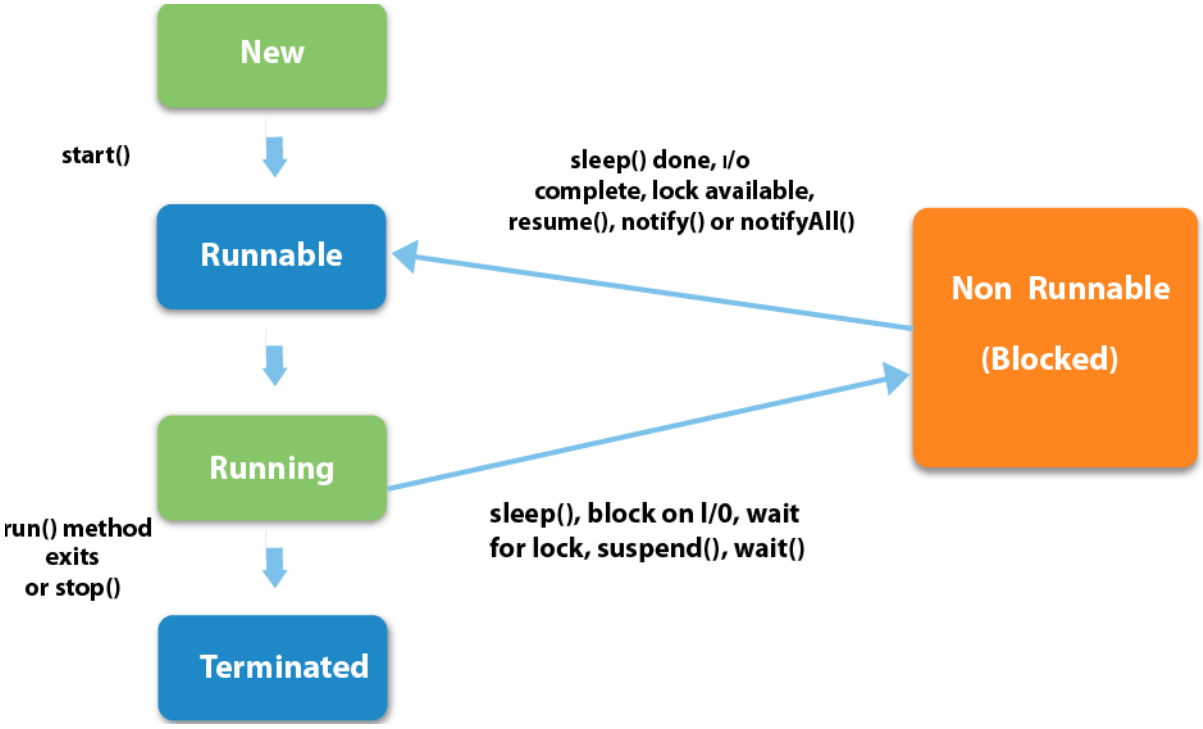
Life cycle of a Thread (Thread States)

A thread can be in one of the five states. According to sun, there is only 4 states in **thread life cycle in java** new, runnable, non-runnable and terminated. There is no running state.

But for better understanding the threads, we are explaining it in the 5 states.

The life cycle of the thread in java is controlled by JVM. The java thread states are as follows:

1. New
2. Runnable
3. Running
4. Non-Runnable (Blocked)
5. Terminated



1) New

The thread is in new state if you create an instance of Thread class but before the invocation of start() method.

2) Runnable

The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.

3) Running

The thread is in running state if the thread scheduler has selected it.

4) Non-Runnable (Blocked)

This is the state when the thread is still alive, but is currently not eligible to run.

5) Terminated

A thread is in terminated or dead state when its run() method exits.

How to create thread

There are two ways to create a thread:

1. By extending Thread class
2. By implementing Runnable interface.

Thread class:

Thread class provide constructors and methods to create and perform operations on a thread.Thread class extends Object class and implements Runnable interface.

Commonly used Constructors of Thread class:

- Thread()
- Thread(String name)
- Thread(Runnable r)
- Thread(Runnable r,String name)

Commonly used methods of Thread class:

1. **public void run():** is used to perform action for a thread.
2. **public void start():** starts the execution of the thread.JVM calls the run() method on the thread.
3. **public void sleep(long miliseconds):** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
4. **public void join():** waits for a thread to die.
5. **public void join(long miliseconds):** waits for a thread to die for the specified miliseconds.
6. **public int getPriority():** returns the priority of the thread.
7. **public int setPriority(int priority):** changes the priority of the thread.
8. **public String getName():** returns the name of the thread.
9. **public void setName(String name):** changes the name of the thread.
10. **public Thread currentThread():** returns the reference of currently executing thread.
11. **public int getId():** returns the id of the thread.
12. **public Thread.State getState():** returns the state of the thread.
13. **public boolean isAlive():** tests if the thread is alive.
14. **public void yield():** causes the currently executing thread object to temporarily pause and allow other threads to execute.
15. **public void suspend():** is used to suspend the thread(depricated).
16. **public void resume():** is used to resume the suspended thread(depricated).
17. **public void stop():** is used to stop the thread(depricated).
18. **public boolean isDaemon():** tests if the thread is a daemon thread.
19. **public void setDaemon(boolean b):** marks the thread as daemon or user thread.
20. **public void interrupt():** interrupts the thread.

21. **public boolean isInterrupted():** tests if the thread has been interrupted.
22. **public static boolean interrupted():** tests if the current thread has been interrupted.

Runnable interface:

The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. Runnable interface have only one method named run().

1. **public void run():** is used to perform action for a thread.

Starting a thread:

start() method of Thread class is used to start a newly created thread. It performs following tasks:

- A new thread starts(with new callstack).
- The thread moves from New state to the Runnable state.
- When the thread gets a chance to execute, its target run() method will run.

---

1) Java Thread Example by extending Thread class

```
class Multi extends Thread{

public void run(){

System.out.println("thread is running...");

}

public static void main(String args[]){

Multi t1=new Multi();

t1.start();

 }

}
```

```
Output:thread is running...
```

---

2) Java Thread Example by implementing Runnable interface

```
class Multi3 implements Runnable{

public void run(){

System.out.println("thread is running...");

}
```

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

Multi3 m1=new Multi3();

Thread t1 =new Thread(m1);

t1.start();

 }

}
```

```
Output:thread is running...
```

If you are not extending the Thread class,your class object would not be treated as a thread object.So you need to explicitely create Thread class object.We are passing the object of your class that implements Runnable so that your class run() method may execute.

Thread Scheduler in Java

Thread scheduler in java is the part of the JVM that decides which thread should run.

There is no guarantee that which runnable thread will be chosen to run by the thread scheduler.

Only one thread at a time can run in a single process.

The thread scheduler mainly uses preemptive or time slicing scheduling to schedule the threads.

Difference between preemptive scheduling and time slicing

Under preemptive scheduling, the highest priority task executes until it enters the waiting or dead states or a higher priority task comes into existence. Under time slicing, a task executes for a predefined slice of time and then reenters the pool of ready tasks. The scheduler then determines which task should execute next, based on priority and other factors

Sleep method in java

The sleep() method of Thread class is used to sleep a thread for the specified amount of time.

Syntax of sleep() method in java

The Thread class provides two methods for sleeping a thread:

- public static void sleep(long miliseconds)throws InterruptedException
- public static void sleep(long miliseconds, int nanos)throws InterruptedException

Example of sleep method in java

```
class TestSleepMethod1 extends Thread{

 public void run(){

  for(int i=1;i<5;i++){

    try{Thread.sleep(500);}catch(InterruptedException e){System.out.println(e);}
```

```
    System.out.println(i);

  }

 }

 public static void main(String args[]){

  TestSleepMethod1 t1=new TestSleepMethod1();

  TestSleepMethod1 t2=new TestSleepMethod1();



  t1.start();

  t2.start();

 }

}
```

Output:

```
1
1
2
2
3
3
4
4
```

As you know well that at a time only one thread is executed. If you sleep a thread for the specified time,the thread shedular picks up another thread and so on.

Can we start a thread twice

No. After starting a thread, it can never be started again. If you does so, an *IllegalThreadStateException* is thrown. In such case, thread will run once but for second time, it will throw exception.

Let's understand it by the example given below:

```
1.  public class TestThreadTwice1 extends Thread{
2.   public void run(){
3.     System.out.println("running...");
4.   }
5.   public static void main(String args[]){
6.    TestThreadTwice1 t1=new TestThreadTwice1();
7.    t1.start();
8.    t1.start();
9.   }
10.
11. }
```

Test it Now

```
running
Exception in thread "main" java.lang.IllegalThreadStateException
```

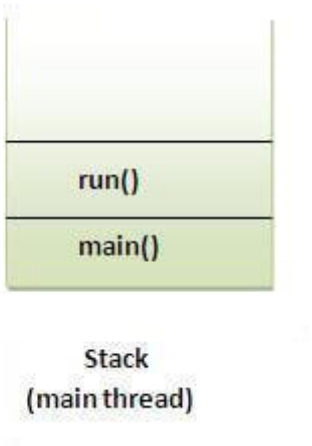
What if we call run() method directly instead start() method?

- Each thread starts in a separate call stack.
- Invoking the run() method from main thread, the run() method goes onto the current call stack rather than at the beginning of a new call stack.

```
1.  class TestCallRun1 extends Thread{
2.   public void run(){
3.     System.out.println("running...");
4.   }
5.   public static void main(String args[]){
6.    TestCallRun1 t1=new TestCallRun1();
7.    t1.run();//fine, but does not start a separate call stack
8.   }
9.  }
```

Test it Now

```
Output:running...
```



Problem if you direct call run() method

```
class TestCallRun2 extends Thread{

 public void run(){

  for(int i=1;i<5;i++){

    try{Thread.sleep(500);}catch(InterruptedException e){System.out.println(e);}

    System.out.println(i);

  }

 }

 public static void main(String args[]){

  TestCallRun2 t1=new TestCallRun2();

  TestCallRun2 t2=new TestCallRun2();



  t1.run();

  t2.run();
```

```
    }

    }

Output:1
       2
       3
       4
       5
       1
       2
       3
       4
       5
```

As you can see in the above program that there is no context-switching because here t1 and t2 will be treated as normal object not thread object.

Priority of a Thread (Thread Priority):

Each thread have a priority. Priorities are represented by a number between 1 and 10. In most cases, thread schedular schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.

3 constants defined in Thread class:

1. public static int MIN_PRIORITY
2. public static int NORM_PRIORITY
3. public static int MAX_PRIORITY

Default priority of a thread is 5 (NORM_PRIORITY). The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.

Example of priority of a Thread:

```
class TestMultiPriority1 extends Thread{

 public void run(){

  System.out.println("running thread name is:"+Thread.currentThread().getName());

  System.out.println("running thread priority is:"+Thread.currentThread().getPriority());



 }

 public static void main(String args[]){

 TestMultiPriority1 m1=new TestMultiPriority1();

 TestMultiPriority1 m2=new TestMultiPriority1();

 m1.setPriority(Thread.MIN_PRIORITY);

 m2.setPriority(Thread.MAX_PRIORITY);

 m1.start();
```

```
        m2.start();



      }

     }
```

```
Output:running thread name is:Thread-0
       running thread priority is:10
       running thread name is:Thread-1
       running thread priority is:1
```

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. This can be done by three ways in java:

1. by synchronized method
2. by synchronized block
3. by static synchronization

Concept of Lock in Java

Synchronization is built around an internal entity known as the lock or monitor. Every object has an 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:

```
class Table{
void printTable(int n){//method not synchronized
  for(int i=1;i<=5;i++){
    System.out.println(n*i);
    try{
     Thread.sleep(400);
    }catch(Exception e){System.out.println(e);}
  }

 }
}

class MyThread1 extends Thread{
Table t;
MyThread1(Table t){
this.t=t;
}
public void run(){
t.printTable(5);
}

}
class MyThread2 extends Thread{
Table t;
MyThread2(Table t){
this.t=t;
}
public void run(){
t.printTable(100);
}
}

class TestSynchronization1{
public static void main(String args[]){
Table obj = new Table();//only one object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
}
}
```

```
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.

```
1.  //example of java synchronized method

class Table{
 synchronized void printTable(int n){//synchronized method
   for(int i=1;i<=5;i++){
     System.out.println(n*i);
     try{
      Thread.sleep(400);
     }catch(Exception e){System.out.println(e);}
   }

 }
}

class MyThread1 extends Thread{
Table t;
MyThread1(Table t){
this.t=t;
}
public void run(){
t.printTable(5);
}

}
class MyThread2 extends Thread{
Table t;
MyThread2(Table t){
this.t=t;
}
public void run(){
t.printTable(100);
}
}

public class TestSynchronization2{
public static void main(String args[]){
Table obj = new Table();//only one object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
}
}
```

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

Example of synchronized method by using annonymous class

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

```
1.  //Program of synchronized method by using annonymous 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: 5
        10
        15
        20
        25
        100
        200
        300
        400
        500
```

Synchronized block in java

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

Suppose you have 50 lines of code in your method, but you want to synchronize only 5 lines, you can use synchronized block.

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

Points to remember for Synchronized block

- Synchronized block is used to lock an object for any shared resource.
- Scope of synchronized block is smaller than the method.

Syntax to use synchronized block

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

Example of synchronized block

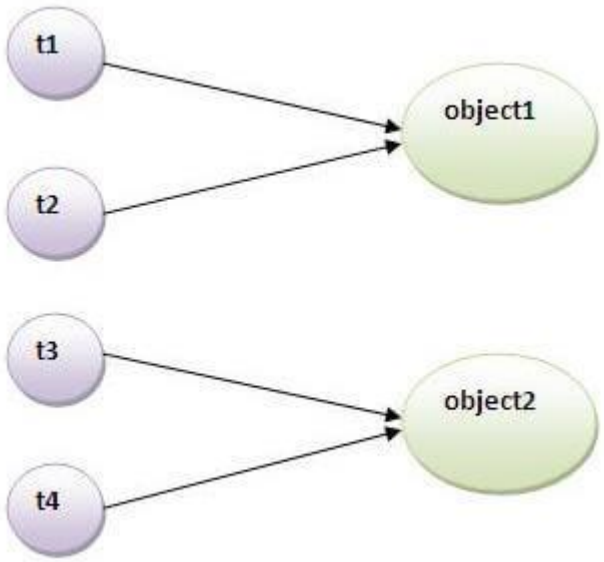
Let's see the simple example of synchronized block.

Program of synchronized block

```
class Table{

 void printTable(int n){
   synchronized(this){//synchronized block
     for(int i=1;i<=5;i++){
      System.out.println(n*i);
      try{
       Thread.sleep(400);
      }catch(Exception e){System.out.println(e);}
     }
   }
 }//end of the method
}

class MyThread1 extends Thread{
Table t;
MyThread1(Table t){
this.t=t;
}
public void run(){
t.printTable(5);
}

}
class MyThread2 extends Thread{
Table t;
MyThread2(Table t){
this.t=t;
}
public void run(){
t.printTable(100);
}
}

public class TestSynchronizedBlock1{
public static void main(String args[]){
Table obj = new Table();//only one object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
}
}
```

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.I want no interference between t1 and t3 or t2 and t4.Static synchronization solves this problem.

Example of static synchronization

In this example we are applying synchronized keyword on the static method to perform static synchronization.

```
class Table{

 synchronized static void printTable(int n){
   for(int i=1;i<=10;i++){
     System.out.println(n*i);
     try{
       Thread.sleep(400);
     }catch(Exception e){}
   }
 }
}

class MyThread1 extends Thread{
public void run(){
Table.printTable(1);
}
}

class MyThread2 extends Thread{
public void run(){
Table.printTable(10);
}
}

class MyThread3 extends Thread{
public void run(){
Table.printTable(100);
}
}
```

```
class MyThread4 extends Thread{
public void run(){
Table.printTable(1000);
}
}

public class TestSynchronization4{
public static void main(String t[]){
MyThread1 t1=new MyThread1();
MyThread2 t2=new MyThread2();
MyThread3 t3=new MyThread3();
MyThread4 t4=new MyThread4();
t1.start();
t2.start();
t3.start();
t4.start();
}
}
```

Test it Now

```
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
```

Same example of static synchronization by annonymous class

In this example, we are using annonymous class to create the threads.

```
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.
13. public class TestSynchronization5 {
14. public static void main(String[] args) {
15.
16.    Thread t1=new Thread(){
17.       public void run(){
18.          Table.printTable(1);
19.       }
20.    };
21.
22.    Thread t2=new Thread(){
23.       public void run(){
24.          Table.printTable(10);
25.       }
26.    };
27.
28.    Thread t3=new Thread(){
29.       public void run(){
30.          Table.printTable(100);
31.       }
32.    };
33.
34.    Thread t4=new Thread(){
35.       public void run(){
36.          Table.printTable(1000);
37.       }
38.    };
39.    t1.start();
40.    t2.start();
41.    t3.start();
42.    t4.start();
43.
44. }
45. }
```

```
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
```

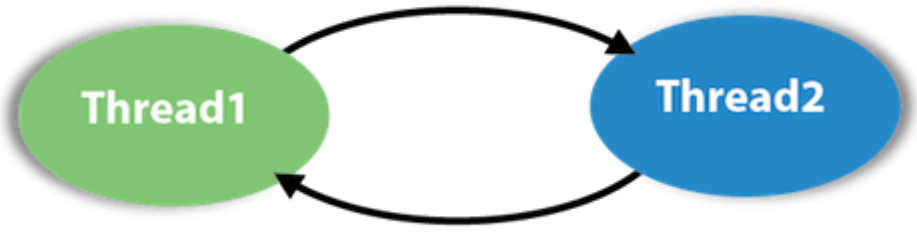
Synchronized block on a class lock:

The block synchronizes on the lock of the object denoted by the reference .class name .class. A static synchronized method printTable(int n) in class Table is equivalent to the following declaration:

```
1.  static void printTable(int n) {
2.      synchronized (Table.class) {       // Synchronized block on class A
3.          // ...
4.      }
5.  }
```

Deadlock in java

Deadlock in java is a part of multithreading. Deadlock can occur in a situation when a thread is waiting for an object lock, that is acquired by another thread and second thread is waiting for an object lock that is acquired by first thread. Since, both threads are waiting for each other to release the lock, the condition is called deadlock.



Example of Deadlock in java

```
public class TestDeadlockExample1 {

  public static void main(String[] args) {

    final String resource1 = "ratan jaiswal";

    final String resource2 = "vimal jaiswal";
```

```
// t1 tries to lock resource1 then resource2
Thread t1 = new Thread() {
  public void run() {
     synchronized (resource1) {
      System.out.println("Thread 1: locked resource 1");

      try { Thread.sleep(100);} catch (Exception e) {}

      synchronized (resource2) {
       System.out.println("Thread 1: locked resource 2");
      }
    }
  }
};

// t2 tries to lock resource2 then resource1
Thread t2 = new Thread() {
  public void run() {
    synchronized (resource2) {
      System.out.println("Thread 2: locked resource 2");

      try { Thread.sleep(100);} catch (Exception e) {}

      synchronized (resource1) {
       System.out.println("Thread 2: locked resource 1");
      }
    }
  }
};
```

```
        t1.start();

        t2.start();

      }

    }
```

```
Output: Thread 1: locked resource 1
        Thread 2: locked resource 2
```

```
Inorder to resolve the deadlock problem , we have to change the order of locks in thread
t2
```

```
        // t2 tries to lock resource1 then resource2

        Thread t2 = new Thread() {

          public void run() {

            synchronized (resource1) {

              System.out.println("Thread 2: locked resource 2");



              try { Thread.sleep(100);} catch (Exception e) {}

              synchronized (resource2) {

                System.out.println("Thread 2: locked resource 1");

              }

            }

          }

    }
```