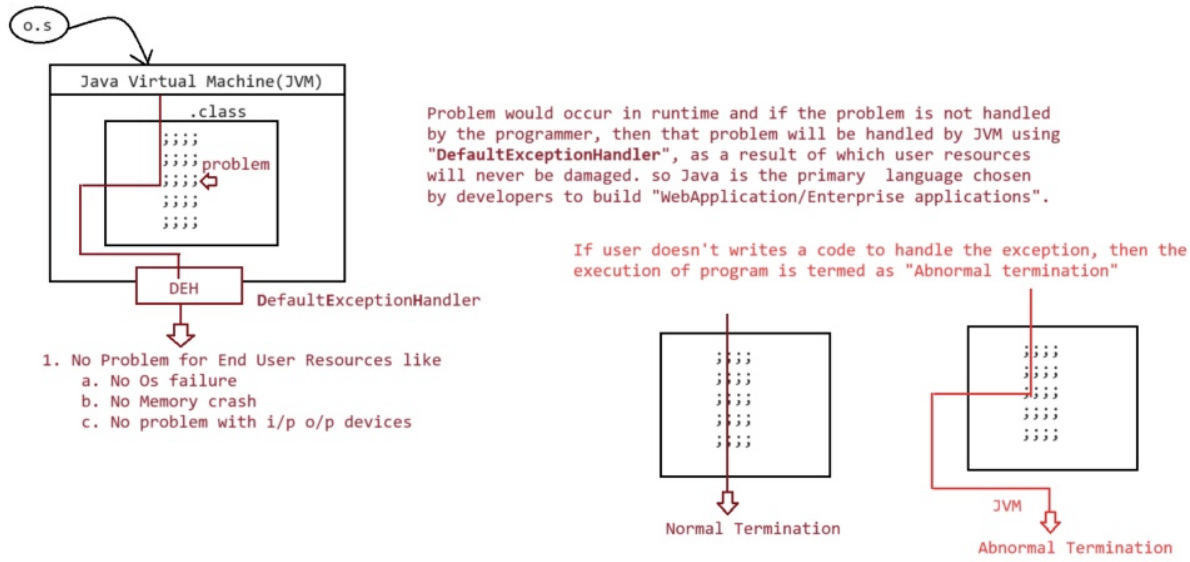
Exception Handling and Multithreading Revision(Part 3)

1)- Developer should have compiler and jvm to develop while end user(who is using application requires only jre).He will download bytecode from network and jre(os dependent) will run the application on his device. Error faced by user at run time is called as exception.

2)- It is important to handle exceptions because user will not able to access the application if a certain exception occurs and it lead to huge business loss



Abnormal Termination --> where we don't have wrote code to handle exception, then jvm will pass the control to DEH and program will end there only(no next set of lines will be executed.)

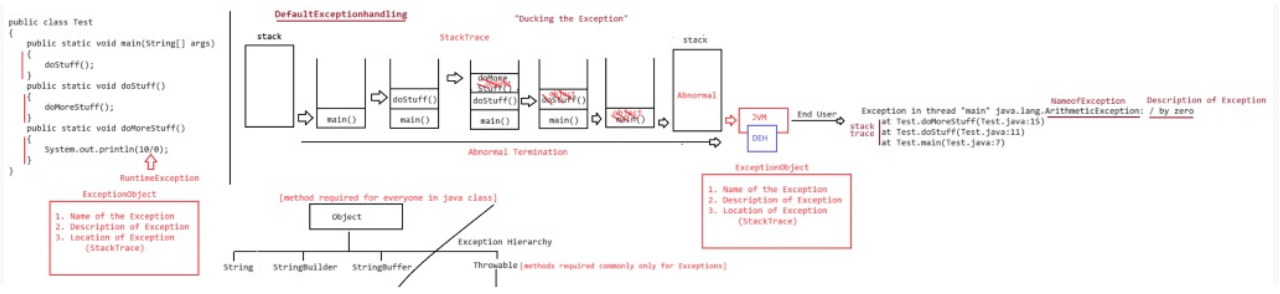
Graceful Termination --> where we have wrote code to handle exception and jvm will execute that part and we will ask him to execute next set of lines. Hence no immediate termination of program.

=> An unwanted/expected event that disturbs the normal flow of execution of program is called "Exception handling".

=> The main objective of Exception handling is to handle the exception.

=> It is available for graceful termination of program.

Exception object + Stack Trace(error explanation as well o n right) → very imp



In above diagram an exception object will be created by the method in which the exception occurred. Now jvm will search current method if any exception handling is written in current method, if not it will move to next method (doStuff)(and also pass exception object to it), will again search for exception logic, if not then will move to main and will perform same operations. Finally the program will terminate abnormally. Now since the methods were removed in abnormal fashion from the stack so exception object will be passed to jvm and jvm will pass the object to DEH and DEH will handle that exception normally and will handle the control to user. ---> exception is handled by jvm hence it is known as default exception handling or ducking(not handling) the exception.

JVM is written in such a way that if any problem or exception occurs then it will not allow the problem to reach the hardware devices, or not the end user, it will not corrupt the os or will not corrupt the hardware devices and I/o devices, will not crash the memory. It will pass that exception object to Default exception handler. If JVM was not handled properly than sometime end users device might shut down automatically(crash) or memory might be crashed.

RunTimeStackMechansim

For every thread in java language, jvm create a seperate stack at the time of Thread creation.

All method calls performed by this thread will be stored in the stack. Every entry in the stack is called **"StackFrame/Activation Record".**

When exception is raised inside any method, that method is responsible for creating the Exception object will the following details

location/stacktrace::

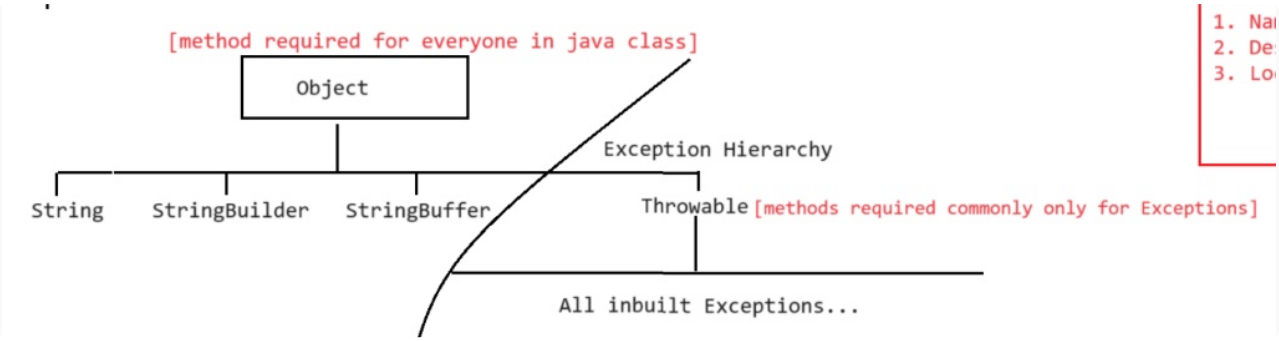
1. This Exception object will be handed over to jvm, now jvm will check whether the method has the handling code or not,if it is not available then that method will be abnormally terminated. since it is a method, it will propogate the exception object to caller method.
2. Now jvm will check whether the caller method is having the code of caller method or not if it is not available, then that method will be abnormally terminated.
3. Simiar way if the exception object is propogated to main(), jvm will check whether the main() is having a code for handling or not, if not then the exception object will be propagated to JVM by terminating the main().
4. JVM now will handover the exception object to "Default exception handler", the duty of "default exception handler" is to just print the exception object details in the following way

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

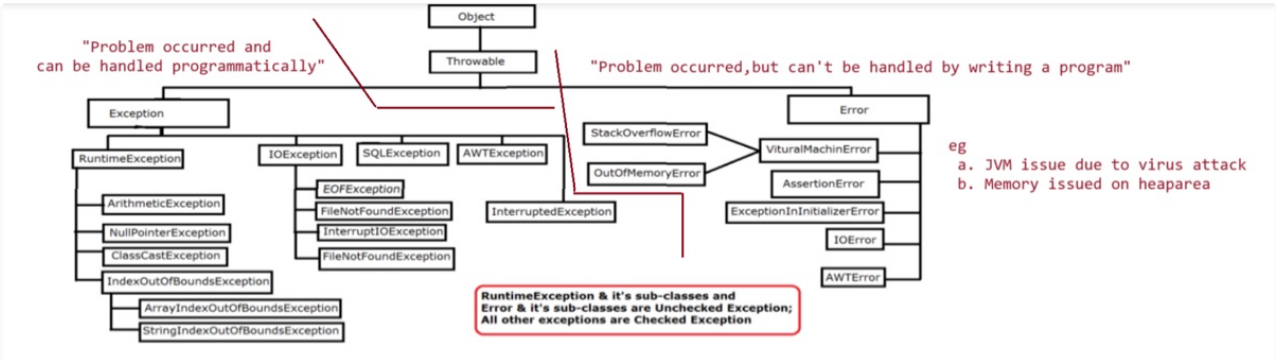
at TestApp.doMoreStuff

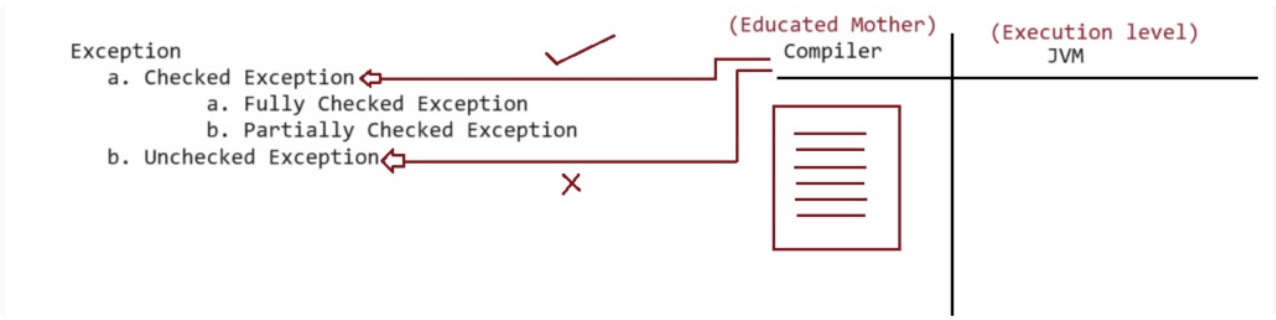
at TestApp.doStuff

at TestApp.main



Instead of including common methods for Exception in Object a separate class is made known as Throwable whose parent is Object. Throwable will include all the exceptions





Exception:: Most of the cases exceptions are caused by our program and these are recoverable

ex:: If FileNotFoundException occurs then we can use local file and we can continue rest of the program execution normally.

Error:: Most of the cases errors are not caused by our program these are due to lack of system resources and these are non-recoverable.

ex:: If OutOfMemoryError occurs being a programmer we can't do anything the program will be terminated abnormally.

Checked vs UnCheckedExceptions

=> The exceptions which are checked by the compiler whether programmer handling or not, for smooth execution of the program at the runtime are called CheckedException.

eg::FileNotFoundException, IOException, SQLException...

=> The exceptions which are not checked by the compiler whether programmer is handling or not such type of exceptions are called as "UnCheckedExceptions".

eg::NullPointerException, ArithmeticException

Note:: RunTimeException and its child classes, Error and its child classes are called as "UncheckedException",remaining all exceptions are considered as "CheckedExceptions".

Note:: Whether the exception is checked or unchecked, compulsorily it should occurs at runtime only and there is no chance of occuring any exception at compile time.

A checked exception is said to be fully checked exception if and only if all its child classes are also checked.

1. IOException
2. InterruptedException

A checked exception is said to be partially checked if and only if some of its child classes are unchecked.
eg:: Throwable, Exception

Customized Exception handling

1)- with try catch

2)- throw(manually throw an exception. **it cant handle exceptions**)(**mainly used to throw an customized exception not for predefined exception.**)

3)- throws("throws" keyword required **only to convince compiler so that it doesnt throw error at compile time for checked exceptions. Usage of throws keyword does not prevent abnormal termination of the program. Hence recommended to use try-catch over throws keyword.**)

4)- try with resources**(once the control reaches the end of try automatically br will be closed)(untill 1.6 version try should compulsorily be followed by either catch or finally, but from 1.7 version we can take only take try with resources without catch or finally.)**

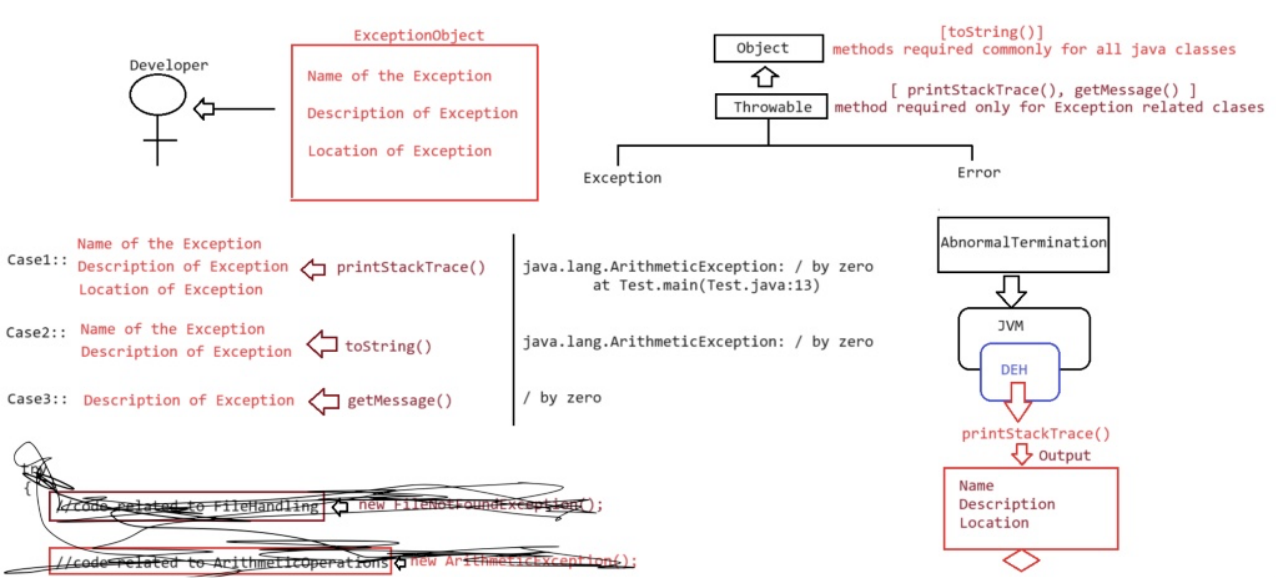
5)- **MultiCatchBlock**(**Till jdk1.6, eventhough we have multiple exception having same handling code we have to write a seperate catch block for every exceptions, it increases the length of the code and disturbs readability.)(In multicatch block,there should not be any relation b/w exception types(either child to parent or parent to child or same type) it would result in compile time error.)**

Note::

1. Within the try block if anywhere an exception raised then rest of the try block won't be executed even though we handled that exception. Hence we have to place/take only risk code inside try block and length of the try block should be as less as possible.
2. If any statement which raises an exception and it is not part of any try block then it is always abnormal termination of the program.
3. There may be a chance of raising an exception inside catch and finally blocks also in addition to try block.

Throwable class defines the following methods to print exception information to the console
printStackTrace()⇒ **in diagram**

Default exception handler internally uses printStatckTrace() method to print exception information to the console.



Right side diagram is really important as it shows difference between the methods inside of Object and Throwble

Try with mulitple catch Blocks

The way of handling the exception is varied from exception to exception, hence for every exception type it is recommended to take a separate catch block. That is try with multiple catch blocks is possible and recommended to use.(**not recommended → for any type of Exception we are using same catch block.**)

```
try
{
    //code related to FileHandling  ⇦ new FileNotFoundException();

    //code related to ArithmeticOperations ⇦ new ArithmeticException();

    //code related to SQLOperations  ⇦ new SQLException();
}catch(IOException e){ //IOException e = new FileNotFoundException();
    //handling logic w.r.t FileHandling
}catch(ArithmeticException e){ //ArithmeticException e = new ArithmeticException();
    //handling logic w.r.t ArithmeticOperations
}catch(SQLException e){ //SQLException e = new SQLException();
    //handling logic w.r.t SQLOperations
}catch (Exception e){
    //handling logic w.r.t Generic statements
}
```

This approach is highly recomended because for any exception raise we are defining a seperate catch block.

If try with multiple catch blocks present then order of catch blocks is very important, it should be from child to parent by mistake if we are taking from parent to child then we will get "CompileTimeError" saying

finally

1)- It is not recommended to take clean up code inside try block becoz there is no gurantee for the execution of every statement inside a try block.

2)- It is not recommended to place clean up code inside catch block becoz if there is no exception then catch block wont be executed.

3)- we require some place to maintain clean up code which should be executed always irrespective of whether exception raised or not raised and whether or not handled. such type of best place is nothing but finally block.

4)- Hence the main objective of finally block is to maintain cleanup code.

return vs finally

Even though return statement present in try or catch blocks first finally will be executed and after that only return statement will be considered.ie **finally block dominates return statement.**

Example:

```
class Test{
    public static void main(String... args){
        try{
            System.out.println("try block executed");
            return;
        }catch(ArithmeticException e){
            System.out.println("catch block executed");
        }finally{
            System.out.println("finally block executed");
        }
    }
}
```

Output

try block executed

finally block executed

finally vs System.exit(0)

There is only one situation where the finally block wont' be executed is whenever we are using System.exit(0) method.

Whenever we are using **System.exit(0) then JVM itself will be shutdown**, in this case finally block won't be executed.

ie,.. System.exit(0) dominates finally block

Note:: System.exit(0);

1. This argument acts as status code, Instead of Zero, we can't take any integer value.
2. Zero means normal termination, non-zero means abnormal termination
3. This status code internally used by JVM, whether it is zero or non-zero there is no change in the result and effect is same w.r.t program.

Difference b/w final, finally and finalize

final

=> final is the modifier applicable for classes, methods and variables

=> If a class declared as the final then child class creation is not possible.

=> If a method declared as the final then overriding of that method is not possible.

=> If a variable declared as the final then reassignement is not possible.

finally

=> It is a final block associated with try-catch to maintain clean up code, which should be executed always irrespective of whethere exception raised or not raised and whether handled or not handled.

finalize

=> It is a method, always invoked by Garbage Collector just before destroying an object to perform cleanup activites.

Note::

1. **finally block meant for cleanup activites related to try block whereas finalize() method for cleanup activites related to object.**
2. **To maintain cleanup code finally block is recomended over finalize() method because we can't expect exact behaviour of GC.**

Note:

1. If we are not entering into try block then finally block wont be executed.
2. If we are entering into try block without executing finally block we can't come out.
3. We can write try inside try, nested try-catch is possible.
4. Specific exceptions can be handled using inner try catch and generalized exceptions can be handled using outer try catch.

Note::

```
public class TestApp{
```

```
    public static void main(String... args){
        try{
            System.out.println(10/0);
        }catch(ArithmeticException ae){
            System.out.println(10/0);
        }finally{
            String s=null;
            System.out.println(s.length());
        }
    }
}
```

Default exception handler handles the most recent exception and it can handle only one exception.

RE: java.lang.NullPointerException.(from try and catch block exception was arithemetic exception and after finally exception object is NullPointerException)

Rules::

1. Whenever we are writing try block compulsorily we should write either catch block or finally try without catch and finally is invalid.
2. Whenever we are writing catch block, compulsorily try block is required.
3. Whenever we are writing finally block, compulsorily try block is required.
4. try catch and finally order is important.
5. With in try catch finally blocks, we can take try catch finally.
6. **For try catch finally blocks curly braces are mandatory.**

throw keyword in java

This keyword is used in java to throw the exception object manually and **informing jvm to handle the exception.**

Eg#1.

```
class Test{
    public static void main(String... args){
        System.out.println(10/0);
    }
}
```

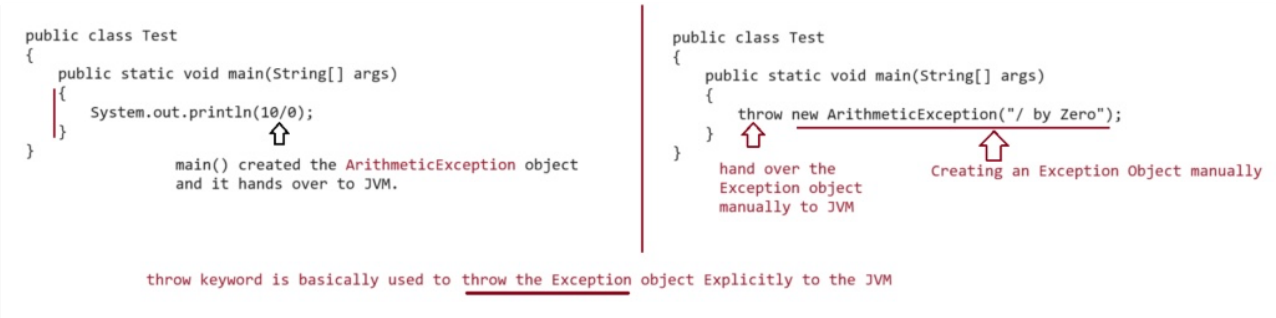
Here the jvm will generate an Exception called "ArithmeticException", since main() is not handling it will hand-over the control to jvm, jvm will handover to DEH to dump the exception object details through printStackTrace().

vs

```
class Test{
    public static void main(String... args){
        throw new ArithmeticException("/by Zero");
    }
}
```

Here the programmer will generate ArithmeticException, and this exception object will be delegated to JVM, jvm will hand-over the control to DEH to dump the excpetion information details through printStackTrace().

Note:: throw keyword is mainly used to throw an customized exception not for predefined exception.



eg → ==========. very important. ==========

```
class Test{
   static ArithmeticException e;
   public static void main(String... args){
      throw e;
   }
}
```

Output

Exception in thread "main" java.lang.NullPointerException. --> just throws the reference of the object to the jvm and then jvm decides about the exception hence NullPointerException(as e is null)

Case2

After throw statement we can't take any statement directly otherwise we will get compile time error saying unreachable statement.

Case3

We can use throw keyword only for Throwable types otherwise we will get compile time error saying incompatible type.

eg#1.

```
class Test3{
    public static void main(String... args){
        throw new Test3();
    }
```

}

Output::

Compile time error.

found::Test3

required:: java.lang.Throwable

Sol →

eg#2.

```
public class Test3 extends RunTimeException{
    public static void main(String... args){
        throw new Test3();
    }
}
```

Output::

RunTimeError: Exception in thread "main" Test3

If there is any checked exception in the code and we haven't handled it using try catch then compiler will give error as we need to handle it. throws keyword is another way in which we can say to compiler that exception is handled and you need not to throw any error at compile time. It is used at method signature level and we don't use it for unchecked exception as they don't give any error at compile time.

throws statement

In our program if there is a chance of raising checked exception then compulsory we should handle either by try catch or by throws keyword otherwise the code won't compile.

We can handle this compile time error by using the following 2 ways

1. using try catch
2. using throws keyword

1. **using try catch**

```
class Test3{
    public static void main(String... args){
        try{
            Thread.sleep(5000);
        }catch(InterruptedException ie){}
        }
}
```

ouput:: compiles and succesfully running

2. **using throws keyword**

```
class Test{
    pubilc static void main(String... args) throws InterruptedException{
        Thread.sleep(5000);
    }
}
```

output:: compiles and successfully running.

Note::

1)- Hence the main objective of "throws" keyword is to **delegate** the responsibility of exception handling to the caller method.

2)- throws keyword required only for checked exception. usage of throws keyword for unchecked exception there is no use.

3)- "throws" keyword required **only to convince compiler.** **Usage of throws keyword does not prevent abnormal termination of the program. Hence recommended to use try-catch over throws keyword.**

eg#1.

```
class Test{
    public static void main(String... args) throws InteruptedException{
        doWork();
    }
    pubilc static void doWork() throws InteruptedException{
        doMoreWork();
    }
    public static void doMoreWork() throws InteruptedException{
        Thread.sleep(5000);
    }
}
```

In the above code, if we remove any of the throws keyword it would result in "CompileTimeError".(if throws is added in only doMoreWork then compiler will raise exception that doMoreWork doesn't include code to handle exception and simarlily if we remove it from main and add it in doWork then it will raise that exception is not handled for main method as the method which includes exception is called from main method. So it should be only used when we just require compiler to not throw any exception.)

Case studies of Throwable

Case 1::

we can use throws keyword only for Throwable types otherwise we will get compile time error.

```
class Test3{
    public static void main(String... args) throws Test3{
    }
}
```

output:Compile Time Error,Test3 cannot be Throwable

```
class Test3 extends RuntimeException{
    public static void main(String... args) throws Test3{
    }
}
```

output:Compiles and run successfully

Case2::

Will give error during compilation -->

```
public class Test3 {
    public static void main(String... args) {
        throw new Exception();
    }
}
```

Output:Compile Time Error unreported Exception must be caught or declared to be thrown

Handling logic is required in case of checked exception

Will compile successfully -->

```
public class Test3 {
    public static void main(String... args) throws Exception {
        throw new Exception();
    }
}
```

```
public class Test3 {
    public static void main(String... args) {
        throw new Error();
    }
}
```

No handling logic is required in case of unchecked exceptions(no throws)

Output:RunTimeException Exception in thread "main" java.lang.Error at Test3.main(Test3.java:4)

Case3::

In our program with in try block, if there is no chance of rising an exception then we can't write catch block for that exception, otherwise we will get Compile Time Error saying "exception XXX is never thrown in the body of corresponding try statement", **but this rule is applicable only for fully checked exceptions only.(similar to unreachable code)**

eg#1.

```
public class Test3
{
    public static void main(String... args) {
        try
        {
            System.out.println("hiee");
        }
        catch (Exception e)
        {
        }
    }
}
```

Output:hiee(same if i use ArithmeticException e or Error e inside catch brackets**)**

eg#2.

```
public class Test3
{
    public static void main(String... args) {
        try
        {
            System.out.println("hiee");
        }
        catch (java.io.FileNotFoundException e)
        {
        }
    }
}
```

Ouput::Compile time error(fully checked Exception)

eg#3.

```
public class Test3
{
    public static void main(String... args) {
        try
```

```
        {
            System.out.println("hiee");
        }
        catch (InteruptedException e)
        {
        }
    }
}
```

Ouput::Compile time error(fully checked Exception) exception InterruptedException is never thrown in the body of corresponding try statement.

Summery -->

throw =>

1)- it can be used inside the method body.

2)- it is used to throw the Exception object manually.

3)- it is used for throwing the "Userdefined" Exception object manually.

throws =>

1)- it can be used at method signature level.

2)- it is used to indicate the compiler about the exception handling code.

3)- it is mainly used for "CheckedExeption" object.

Case4: we can use throws keyword only for constructors and methods but not for classes.

eg#1.

```
class Test throws Exception. //invalid
{
    Test() throws Exception{.  //valid
    }
    methodOne() throws Exception{.   //valid
    }
}
```

Exception handling keywords summary

1. **try => Maintain risky code**
2. **catch => Maintain handling code**
3. **finally => Maintain cleanup code**
4. **throw => To handover the created exception object to JVM manually**
5. **throws => To delegate the Exception object from called method to caller method.**

Various compile time errors in ExceptionHandling

1. Exception XXXX is already caught [try with multiple catch blocks]
2. Unreported Exception XXX must be caught or declared to be thrown. [CheckedException]
3. Exception XXXX is never thrown in the body of corresponding try statement. [FullyCheckedException]

SyntacticallyErrors

4. try without catch, finally
5. catch without try
6. finally without try
7. incompatible types :found xxx required:Throwable [using throws/throw Keyword]
8. unreachable code**.[normal coding error]**

CustomException in java

eg#1.

```
//DrivingLicense Generator App :: ageFactor
/*
    age>60 :: TooOldAgeException
    age<18 :: TooYoungAgeException
    otherwise :: issueDL by checking the skill of driving.
*/
//InbuiltException -> CheckedException(partial,fully),UnCheckedException
//CustomException -> UncheckedException[RuntimeException]
import java.util.Scanner;
class TooOldAgeException extends RuntimeException
{
    TooOldAgeException(String msg)
    {
        super(msg);
    }
}
class TooYoungAgeException extends RuntimeException
    {
        TooYoungAgeException(String msg)
        {
            super(msg);
    }
```

```
}
public class Test
{
    public static void main(String[] args)
    {
        Scanner scanner =new Scanner(System.in);
        System.out.print("Enter the age of the candidate:: ");
        int age = scanner.nextInt();
        if (age>60)
        {
            throw new TooOldAgeException("Sorry DL can't be issued for senior citizen people");
        }
        else if(age<18)
        {
            throw new TooYoungAgeException("Sorry DL can't be issued for minor candidates");
        }
        else
        {
            System.out.println("You will get DL soon in registered email...");
        }
    }
}
```

Output

Enter the age of the candidate:: 65

Exception in thread "main" TooOldAgeException: Sorry DL can't be issued for senior citizen people at Test.main(Test.java:38)

Enter the age of the candidate:: 15

Exception in thread "main" TooYoungAgeException: Sorry DL can't be issued for minor candidates at Test.main(Test.java:42)

Enter the age of the candidate:: 19

You will get DL soon in registered email...

Exceptions which are normally occured in java coding

1. Based on the events occured exceptions are classified into 2 types
   a. JVM Exceptions
   b. Programtic Exceptions

JVM Exceptions(Errors)

=> The exceptions which are raised automatically by the jvm whenever a particular event occurs are called JVM Exceptions.

These exceptions originate from the Java Virtual Machine (JVM) itself, typically indicating severe errors that halt program execution.

Handling: These exceptions are generally not intended to be handled within your program as they often signify critical problems. Logging the error and terminating the program gracefully is a common approach.

eg:: OutOfMemoryError

StackOverflowError

ProgramaticExceptions

=> The exceptions which are raised explicitly by the programmer or by API developers is called as "Programatic Exceptions".

eg:: IllegalArgumentException,

NumberFormatException

ArrayIndexOutOfBoundsException =>This exception is raised automatically whenever we are trying to access array elements which is out of the range.

Handling: try-catch

- JVM Exceptions (Errors) are typically unrecoverable and indicate severe issues within the JVM or the program itself.
- Programmatic Exceptions are thrown by your code and can be handled gracefully using `try-catch` blocks to potentially recover and continue program execution.

Top10JavaExceptions

1. ArithmeticException => while working with division operand on int type data.
2. NullPointerException => While working with data which is actually holding null.
3. StackOverFlowError => while working with calling methods in recursive style.
4. IllegalArgumentException

   eg:: Thread t=new Thread();

   t.setPriority(10);

   t.setPriority(100); **//invalid**
5. NumberFormatException => while working with Wrapper classes.
6. ExceptionInInitializerError => if exception occurs during the initialization of static variables or static blocks.

   eg#1.

   ```
   public class Test
   {
       static int i = 10/0;
   }
   ```

eg#2.

```
public class Test
{
    static
    {
        String s= null;
        System.out.println(s.length());
    }
}
```

7. ArrayIndexOutOfBoundsException
8. NoClassDefFoundError
9. ClassCastException
10. IllegalStateException(learn in servlet programming :: Session)
11. AssertionError(learn in Junit)

JVMException

a. ArithmeticException

b. NullPointerException

c. ArrayIndexOutOfBoundsException

d. StackOverFlowError

e. ClassCastException

f. ExceptionInInitalizerError

ProgrammaticException[We use API Code given by other developers]

a. IllegalArgumentException

b. NumberFormatException

c. IllegalStateException

d. AssertionError

Key Differences:

- **Origin:** JVM Exceptions originate from the JVM, while Programmatic Exceptions are thrown by your code.
- **Recoverability:** JVM Exceptions are often unrecoverable, while Programmatic Exceptions can be handled within your program to potentially recover from the error.
- **Inheritance:** **JVM Exceptions typically extend the `Error` class, while Programmatic Exceptions extend the `Exception` class (or a custom subclass).**

1.7 version Enhancements

1. try with resource
2. try with multicatch block

untill jdk1.6, it is compulsorily required to write finally block to close all the resources which are open as a part of try block.

eg::

```
BufferReader br=null
try{
    br=new BufferedReader(new FileReader("abc.txt"));
}catch(IOException ie){
    ie.printStackTrace();
}finally{
    try{
        if(br!=null){
            br.close();
    }
    }catch(IOException ie){
        ie.printStackTrace();
    }
}
```

Problems in the approach

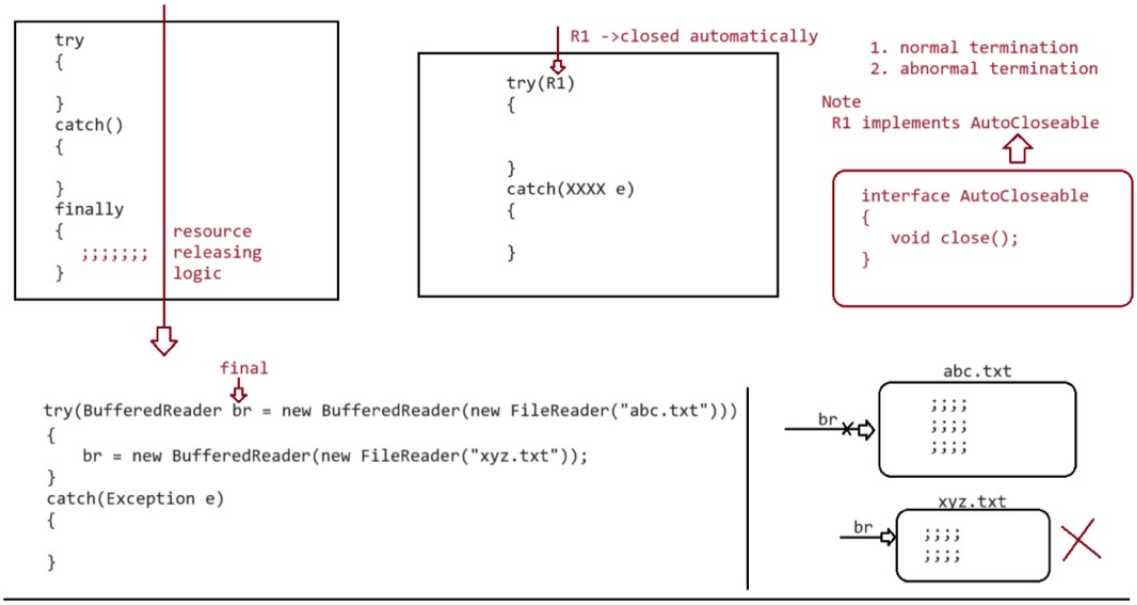
1. **Compulsorily the programmer is required to close all opened resources which increases the complexity of the program**
2. **Compulsorily we should write finally block explicitly, which increases the length of the code and reviews readeablity. To Overcome this problem SUN MS introduced try with resources in "1.7" version of jdk.**

try with resources

In this approach, the resources which are opened as a part of try block will be closed automatically once the control reaches to the **end of try block normally or abnormally**, so it is not required to close explicitly so the complexity of the program would be reduced.

It is not required to write finally block explicitly,so length of the code would be reduced and readability is improved.

```
try(BufferedReader br=new BufferedReader(new FileReader("abc.txt")){
    //use br and perform the necessary operation
    //once the control reaches the end of try automatically br will be closed
}catch(IOException ie){
    //handling code
}
```



Rules of using try with resource

1. we can declare any no of resources, but all these resources should be seperated with ;

   eg#1.

   try(R1;R2;R3;){

   //use the resources

   }

2. All resources are said to be AutoCloseable resources iff the class implements an interface called "**java.lang.AutoCloseable**" either directly or indirectly

   eg:: java.io package classes, java.sql.package classes

3. **All resource reference by default are treated as implicitly final and hence we can't perform reassignment with in try block.**

   try(BufferedReader br=new BufferedReader(new FileWriter("abc.txt")){

   br=new BufferedReader(new FileWriter("abc.txt"));

   }

output::CE: can't reassign a value

4. **untill 1.6 version try should compulsorily be followed by either catch or finally, but from 1.7 version we can take only take try with resources without catch or finally.**

try(R){

//valid

}

5. Advantage of try with resources concept is finally block will become dummy because we are not required to close resources explicitly.

MultiCatchBlock

Till jdk1.6, eventhough we have multiple exception having same handling code we have to write a seperate catch block for every exceptions, it increases the length of the code and disturbs readability.

```
try{
    ....
    ....
    ....
    ....
}catch(ArithmeticException ae){
    ae.printStackTrace();
}catch(NullPointerExcepion ne){
    ne.printStackTrace();
}catch(ClassCastException ce){
    System.out.println(ce.getMessage());
}catch(IOException ie){
    System.out.println(ie.getMessage());
}
```

To overcome this problem SUNMS has introduced "Multi catch block" concept in 1.7 version

```
try{
    ....
    ....
    ....
    ....
}catch(ArithmeticException | NullPointerException e){
    e.printStackTrace();
}catch(ClassCastException | IOException e){
    e.printStackTrace();
}
```

In multicatch block,there should not be any relation b/w exception types(either child to parent or parent to child or same type) it would result in compile time error.

eg::

```
try{
}catch( ArithmeticExeption | Exception e){
    e.printStackTrace();
}
```

Output:CompileTime Error

eg::

```
try
{
    int a = 10/0;
}
catch (ArithmeticException | NullPointerException | ClassCastException e)
{
    //handling logic
    e.printStackTrace();
}
```

Exception Propogation

With in a method, if an exception is raised and if that method does not handle that exception then Exception object will be propogated to the caller method then caller method is responsible to handle that exceptions, This process is called as "**Exception Propogation/Ducking**".

ReThrowing an Exception

To convert one exception type to another exception type, we can use rethrowing exception concept.

eg::

```
public class TestApp
{
    public static void main(String[] args)
    {
        try{
            System.out.println(10/0);
        }catch(ArithmeticException e){
            throw new NullPointerException();   //ReThrowing an Exception
        }
    }
}
```

Output::

```
Exception in thread "main" java.lang.NullPointerException
    at TestApp.main(TestApp.java:10)
```

Rules w.r.t Overriding

1)- parent: public void methodOne() throws Exception{}

child : public void methodOne()

Output:: valid[parent throwing an Exception, Child need not throw any Exception]

2)- parent: public void methodOne(){}

child : public void methodOne() throws Exception{}

Output:: invalid[Child throwing CheckedException,Compulsorily parent should throw the SameCheckedException or its Parent]

3)- parent: public void methodOne()throws Exception{}

child : public void methodOne()throws Exception{}

Output:: valid[Child throwing CheckedException,Compulsorily parent should throw the SameCheckedException or its Parent]

4)- parent: public void methodOne()throws IOException{}

child : public void methodOne()throws IOException{}

Output:: valid[Child throwing CheckedException,Compulsorily parent should throw the SameCheckedException or its Parent]

5)- parent: public void methodOne()throws IOException{}

child : public void methodOne()throws FileNotFoundException,EOFException{}

Output:: Valid[Child throwing CheckedException,Compulsorily parent should throw the SameCheckedException or its Parent]

6)- parent: public void methodOne()throws IOException{}

child : public void methodOne()throws FileNotFoundException,InterruptedException{}

Output:: invalid[Child throwing CheckedException,Compulsorily parent should throw the SameCheckedException or its Parent]

7)- parent: public void methodOne()throws IOException{}

child : public void methodOne()throws FileNotFoundException,ArithmeticException{}

Output:: Valid[Child throwing CheckedException,Compulsorily parent should throw the SameCheckedException or its Parent]

The rule is not applicable for UnCheckedException.(There are no restrictions on UncheckedException.)

8- parent: public void methodOne()

child : public void methodOne()throws ArithmeticException,NullPointerException,RuntimeException{}

Output:: Valid[Rule is not applicable for UncheckedException].(There are no restrictions on UncheckedException.)

Constructor level

=> Constructor level Exceptions should be of Sametype otherwise the code won't compiler(if child throws file not found exception than compulsorily parent should throw file not found exception as well)

=> Rule is applicable only for CheckedExceptions not for UnCheckedExceptions.

=> Rules are separate for a method and constructor, because constructor won't participate in "inheritance" so we can't override the constructor.

```
class Parent
{
    Parent() throws java.io.FileNotFoundException{
    }
}
class Child extends Parent
{
    Child() throws java.io.FileNotFoundException{
    }
}
public class Test
{
     //(very imp) as child throws so corresponding method should also be handled.
    public static void main(String[] args)throws Exception{
        Parent p = new Child();
        System.out.println(p);
    }
}
```

Output:: Compilation succesfully.

```
class Parent
{
    Parent() throws ArithmeticException{ //Its unchecked hence rule is not applicable for it
    }
```

```
}
class Child extends Parent
{
    Child() throws RuntimeException{
    }
}
public class Test
{
    public static void main(String[] args){
        Parent p = new Child();
        System.out.println(p);
    }
}
```

Output:: Compilation succesfully.

Note: In RealTime project we work with "CustomExceptions" and we use "try with resource" to avoid resourceleak.

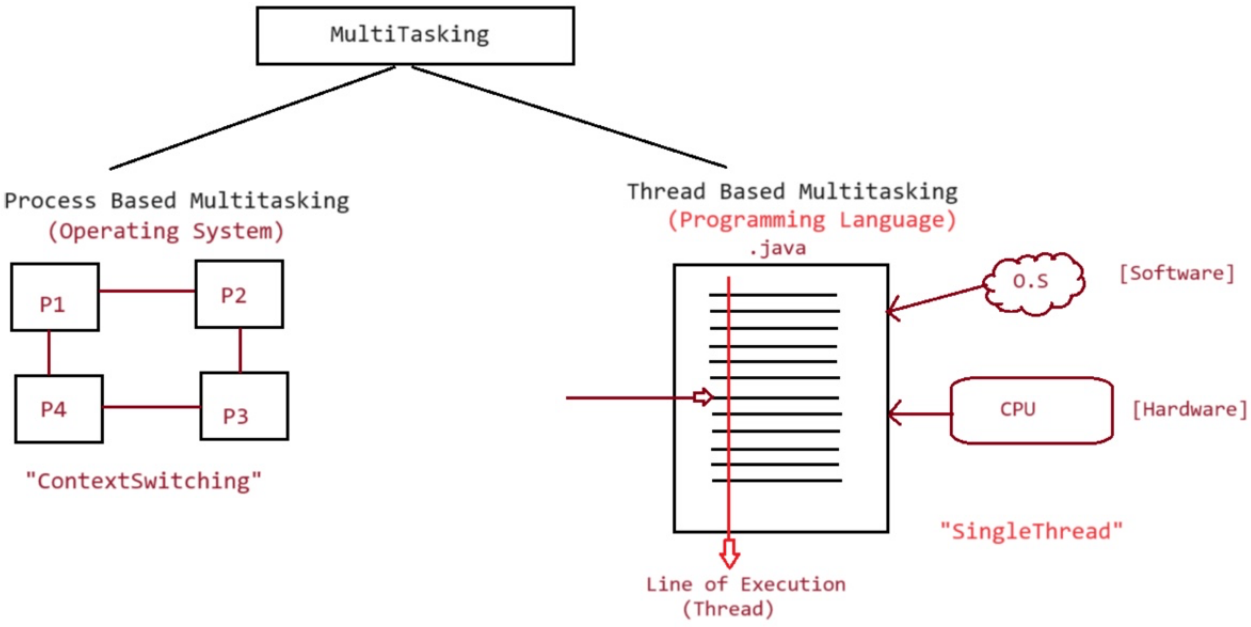
What is MultiTasking?

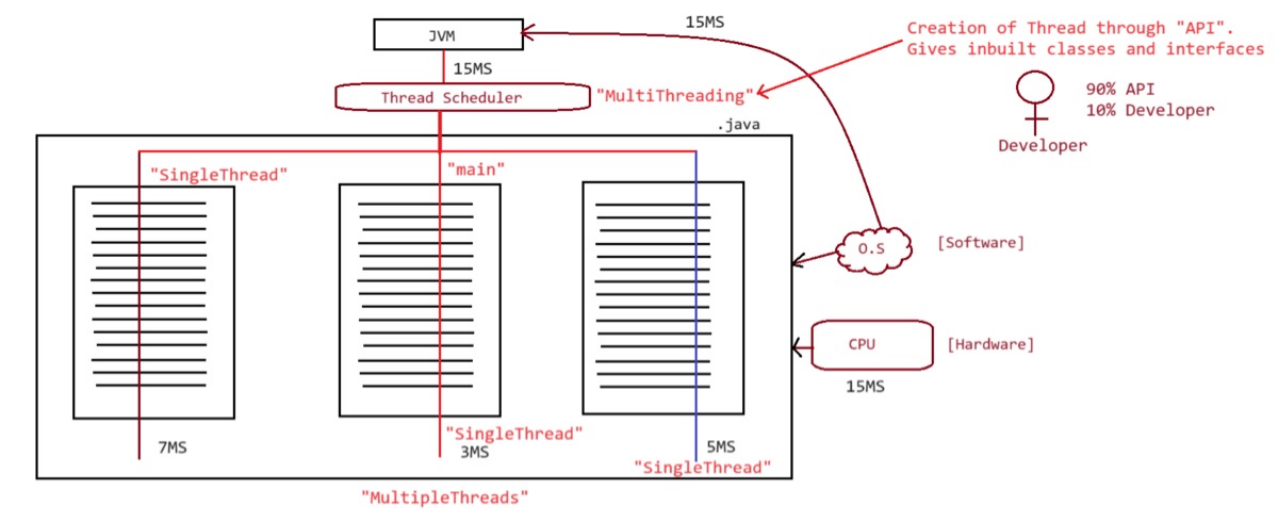
Any program which is under execution is called "Task/Process".

There are 2 types of Multitasking.

a. Process based multitasking

b. Thread based multitasking.





Suppose os decides that 15ms are required to completely execute the thread and then this time is passed to thread schedular and then cpu is communicated the total time for which the resource is required.

Process based multitasking

Executing several tasks simultaneously where each task is a seperate independent process such type of multitasking is called "process based multitasking".
eg:: typing a java pgm

listening to a song
downloading the file from internet

Process based multitasking is best suited at "os level".

Thread based multitasking

=>Executing several tasks simulatenously where each task is a seperate independent part of the same Program, is called "Thread based MultiTasking".

Each independent part is called "Thread".

1. This type of multitasking is best suited at "Programatic level".

   The main advantages of multitasking is to reduce the response time of the system and to improve the performance.

2. The main important application areas of multithreading are

   a. To implement mulitmedia graphics

   b. To develop web application servers

   c. To develop video games

3. Java provides inbuilt support to work with threads through API called Thread,Runnable,ThreadGroup,ThreadLocal,...
4. To work with multithreading, java developers will code only for 10% remaining 90% java API will take care..

What is thread?

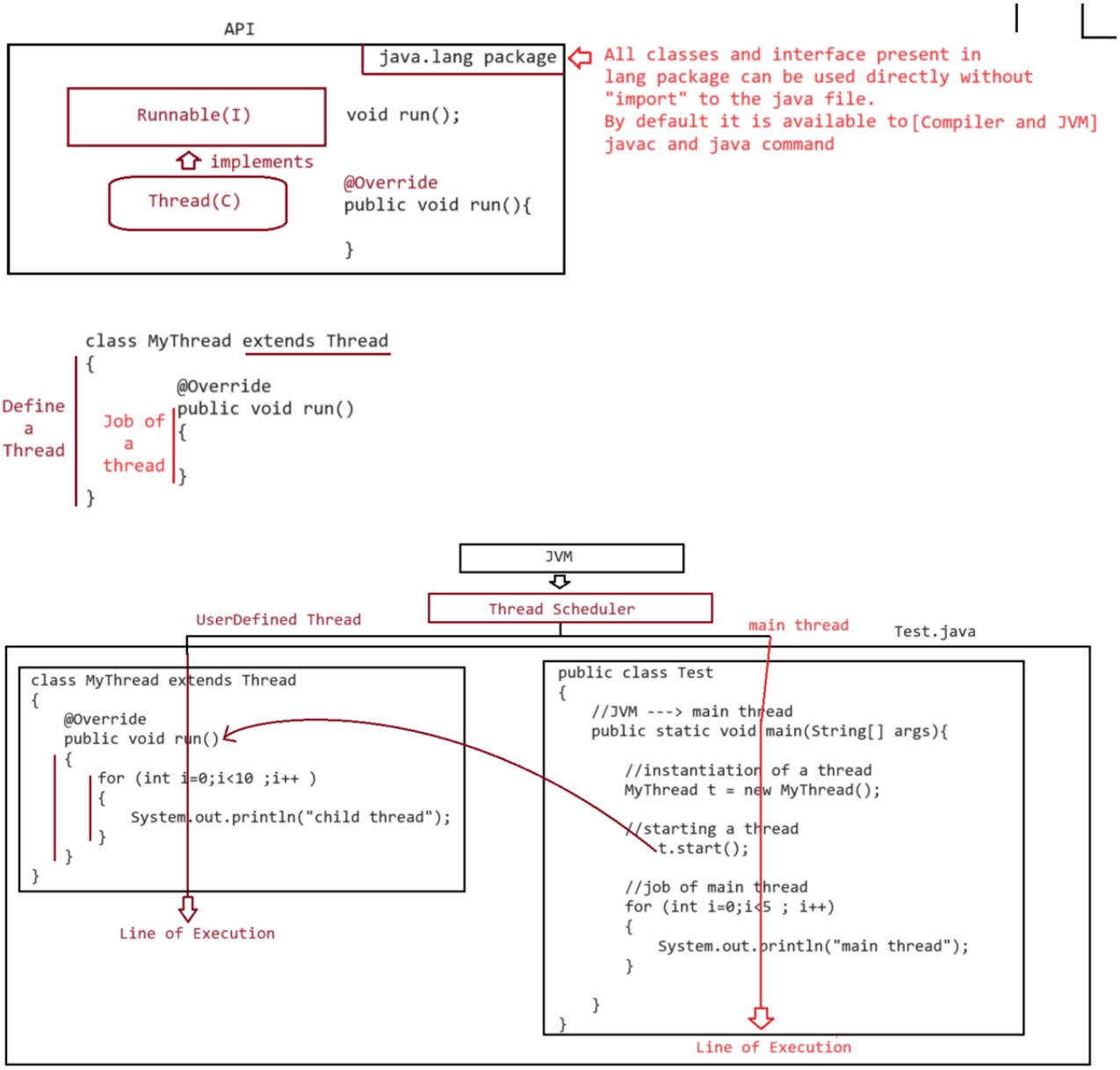
A. Seperate flow of execution is called "Thread". if there is only one flow then it is called "SingleThread" programming.

For every thread there would be a seperate job.

B. In java we can define a thread in 2 ways

a. implementing Runnable interface

b. extending Thread class



The line after t.start() in above diagram we have two threads and we dont know which thread will be executed first as it is decided by Thread Scheduler

By default first the main thread will be executed by jvm and from main other threads will start. Thread scheduler will decide which thread to run and how much time required for each thread to execute.Output of the program cannot be predicted as execution of thread depends on thread scheduler.

In threads our main focus is on cpu utilization(not on the the output of execution(order of execution.))

Behind the scenes

1. Main thread is created automatically by JVM.
2. Main thread creates child thread and starts the child thread.

Case1:ThreadScheduler

If multiple threads are waiting to execute,then which thread will execute 1st is decided by ThreadScheduler which is part of JVM.

In case of MultiThreading we can't predict the exact output only possible output we can expect.

Since jobs of threads are important,we are not interested in the order of execution it should just execute such that performance should be improved.

case2: diff b/w t.start() and t.run()

if we call t.start() and seperate thread will be created which is responsible to execute run() method. if we call t.run(), no seperate thread will be created rather the method will be called just like normal(**that is why we don't directly call t.run() in main method.)** method by main thread.if we replace t.start() with t.run() then the output of the program would be

child thread

child thread

....

main thread

main thread

...

case3:: Importance of Thread class start() method

For every thread, required mandatory activities like registering the thread with Threadscheduler will be taken care by Thread class start() method and programmer is responsible of just doing the job of the Thread inside run() method.

start() acts like an assistance to programmer.

start()

{

register thread with ThreadScheduler

All other mandatory low level activities

invoke or calling run() method.

}

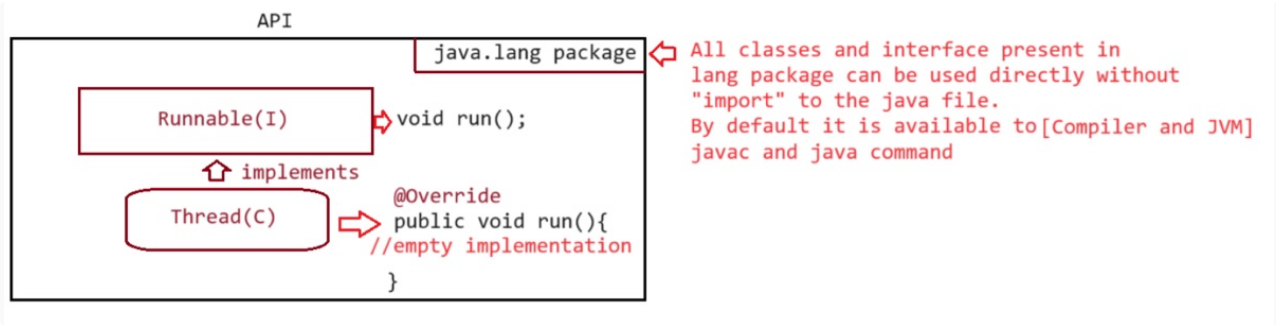
We can conclude that without executing Thread class start() method there is no chance of starting a new Thread in java.

Due to this start() is considered as heart of MultiThreading.

case4:: If we are not overriding run() method

If we are not Overriding run() method then Thread class run() method will be executed which has empty implementation and hence we wont get any output.

It is highly recomended to override run() method, otherwise don't go for MultiThreading concept.



case5:Overloading of run() method

we can overload run() method **but Thread class start() will always call run() with zero argument**. if we overload run method with arguments, then we need to explicitly call argument based run method and it will be executed just like normal method.

If we override start() then our start() method will be executed just like normal method,but no new Thread will be created and no new Thread will be started.

eg#1.

```
class MyThread extends Thread{
    public void run(){
        System.out.println("no arg method");
    }
    public void start(){
        System.out.println("start arg method");
    }
}
class ThreadDemo{
    public static void main(String... args){
        MyThread t=new MyThread();
        t.start();
    }
}
```

Output:: start arg method

It is never recomended to override start() method.

case7::

If i try to override start method than run method will not be called and also the thread will not be created as we are overriding the existing functionality.

If we want it to work in the first line we need to call super.start(), then we can write our own logic. This way we can incorporate existing and our functionality

eg#2.

```
class MyThread extends Thread{
    public void start(){
        super.start();
        System.out.println("start method");
    }
    public void run(){
        System.out.println("run method");
    }
}
class ThreadDemo{
    public static void main(String... args){
        MyThread t=new MyThread();
        t.start();
        System.out.println("Main method");
    }
}
```

Output::

MainThread

a. Main method

b. start method

UserDefinedThread

a. run method

case8:: Life cycle of a Thread

MyThread t=new MyThread(); **// Thread is in born state**

t.start(); **//Thread is in ready/runnable state**

if Thread scheduler allocates CPU time then we say thread entered into Running state.

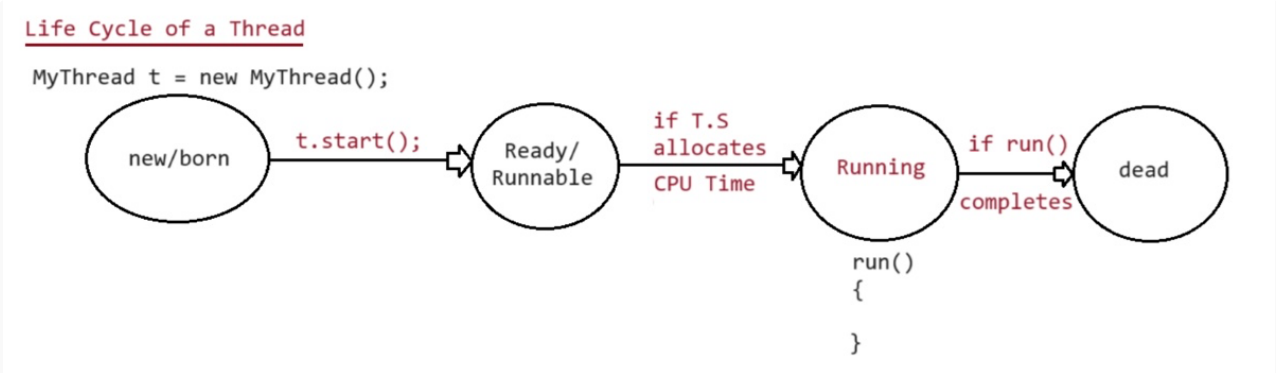
if run() is completed by thread then we say thread entered into dead state.

=> Once we created a Thread object then the Thread is said to be in new state or born state.

=> Once we call start() method then the Thread will be entered into Ready or Runnable state.

=> If Thread Scheduler allocates CPU then the Thread will be entered into running state.

=> Once run() method completes then the Thread will entered into dead state.



case9::

After starting the Thread, we are not supposed to start the same Thread again,then we say Thread is in "**IllegalThreadStateException**".

MyThread t=new MyThread(); **// Thread is in born state**

t.start(); **//Thread is in ready state**

....

....

t.start(); **//IllegalThreadStateException**

Defining a Thread by implementing Runnable Interface

eg::1

```
class MyRunnable implements Runnable{
    @Override
    public void run(){
        for(int i=1;i<=10;i++)
            System.out.println("child thread");
        }
    }
public class ThreadDemo{
    public static void main(String... args){
        MyRunnable r=new MyRunnable();
        Thread t=new Thread(r);  //call MyRunnable run()
        t.start();
        for(int i=1;i<=10;i++)
            System.out.println("main thread");
        }
```

}

Case study

MyRunnable r=new MyRunnable();

Thread t1=new Thread();

Thread t2=new Thread(r);

case1:: t1.start();

A new thread will be created and it will execute Thread class run() which has empty implementation.

case2:: t1.run()

No new Thread will be created, rather run() of Thread class will be executed just like normal method

case3:: t2.start()

A new thread will be created and it will execute MyRunnable class run() which has specific job.

case4:: t2.run()

No new Thread will be created, rather run() of MyRunnable class will be executed just like normal method

case5:: r.start()

It results in compile time error at MyRunnable class***(start method comes from Thread class)***

symbol:method start()

location:MyRunnable

case6:: r.run()

No new Thread will be created, rather run() of MyRunnable class will be executed just like normal method

1)- In which of the above cases a new Thread will be created which is responsible for the execution of MyRunnable run() method ?

Ans.t2.start()

2)- In which of the above cases a new Thread will be created ?

Ans.t1.start();

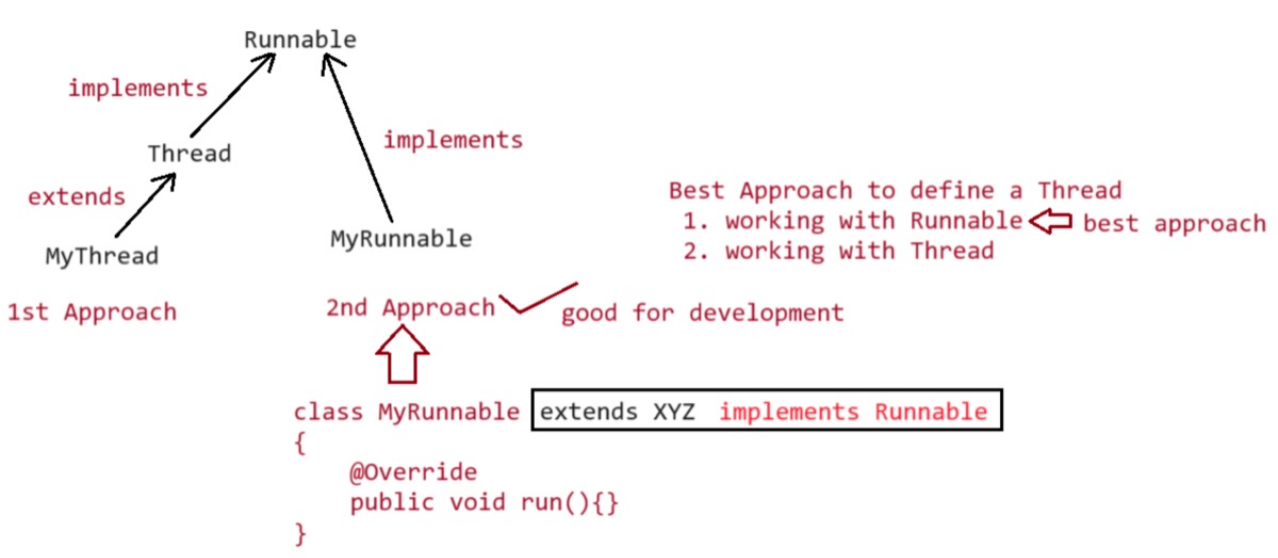
t2.start();

3)- In which of the above cases MyRunnable class run() will be executed ?

Ans.t2.start(); // In multithreading environment

t2.run();

r.run();



(start method comes from Thread class → that's why MyRunnable require Thread class)

Best approach is working with Runnable because we can even extends some properties from some xyz class while in working with thread since thread is a class we cannot extend another class as multiple inheritance is invalid in java.

Which approach is the best approach?

a. implements Runnable interface is recomended becoz our class can extend other class through which inheritance benift can brought in to our class. Internally performance and memory level is also good when we work with interface.
b. if we work with extends feature then we will miss out inheritance benefit becoz already our class has inherited the feature from "Thread class", so we normally we don't prefere extends approach rather implements approach is used in real time for working with "MultiThreading".

Alternate approach to define a Thread(not recomended as it is confusing)

```
class MyThread extends Thread{
    public void run(){
        System.out.println("child thread");
    }
}
class ThreadDemo {
    public static void main(String... args){
        MyThread t=new MyThread();
        Thread t1=new Thread(t);
        t1.start();
        System.out.println("main thread");
    }
```

}

internally related -->

Runnable

^

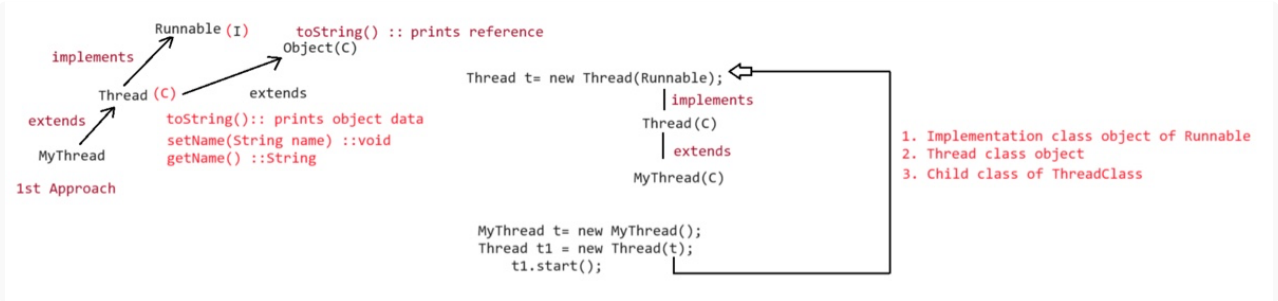
|

Thread

^

|

MyThread



In the above diagram → toString for object prints reference while when it is inherited and overriden in Thread class it prints object data. Right side image shows shows the above approach which is not recommended.

Printing Names, priority, data and setting the name of the Thread

System.out.println(t.getName()); //Thread-0

Thread.currentThread().setName("Yash"); //Yash

System.out.println(Thread.currentThread().getName()); //Yash

System.out.println(10/0);

//Exception in thread "yash" java.lang.ArithmeticException:/by zero TestApp.main()

ThreadPriorities

User defined thread has same priority as that of main thread because that thread is created by main thread only so same priority is replicated to user defined thread as well**(default priority)**.

For every Thread in java has some priority. valid range of priority is 1 to 10, **it is not 0 to 10**. if we try to give a differnt value the it would result in "**IllegalArgumentException**".
Thread.MIN_PRIORITY = 1
Thread.MAX_PRIORITY = 10
Thread.NORM_PRIORITY = 5
Thread scheduler allocates cpu time based on "Priority".
If both the threads have the same priority then which thread will get a chance as a pgm we can't predict becoz it is vendor dependent.
We can set and get priority values of the thread using the following methods

a. public final void setPriority(int priorityNumber)

b. public final int getPriority()

The allowed priorityNumber is from 1 to 10,if we try to give other values it would result in "**IllegalArgumentException**".
System.out.println(Thread.currentThread().setPriority(100); **//IllegalArgumentException.**

DefaultPriority

The default priority for only main thread is "5",where as for other threads priority will be inherited from parent to child.

Parent Thread priority will be given as Child Thread Priority.

eg#1.

```
System.out.println(Thread.currentThread().getPriority());  //5
Thread.currentThread().setPriority(7);
MyThread t= new MyThread();
System.out.println(Thread.currentThread().getPriority()); //7
```

MyThread is creating by "mainThread", so priority of "mainThread" will be shared as a priority for "MyThread".

eg#2.

```
class MyThread extends Thread{
    @Override
    public void run(){
        for (int i=1;i<=5 ;i++ ){
            System.out.println("child thread");
        }
    }
}
public class TestApp{
    public static void main(String... args){
        MyThread t= new MyThread();
        t.setPriority(7); //line -1
        t.start();
```

```
        for (int i=1; i<=5; i++){
            System.out.println("main thread");
        }
    }
}
```

Since priority of child thread is more than main thread, jvm will execute child thread first whereas for the parent thread priority is 5 so it will get last chance. if we comment line-1, then we can't predict the order of execution becoz both the threads have same priority.

Some platform won't provide proper support for Thread priorities.

eg:: windows7,windows10,...(or pirated versions)(check once if it works in mac)

1. when we have multiple threads, which threads will get a chance for execution?

Ans. It will be decided based on the priority. If multiple threads have the same priority then ThreadScheduler will decide which thread to execute.

2. When we create a thread(user-defined), what is the default name given to the thread by JVM?

Ans. Thread-0

We can prevent Threads from Execution

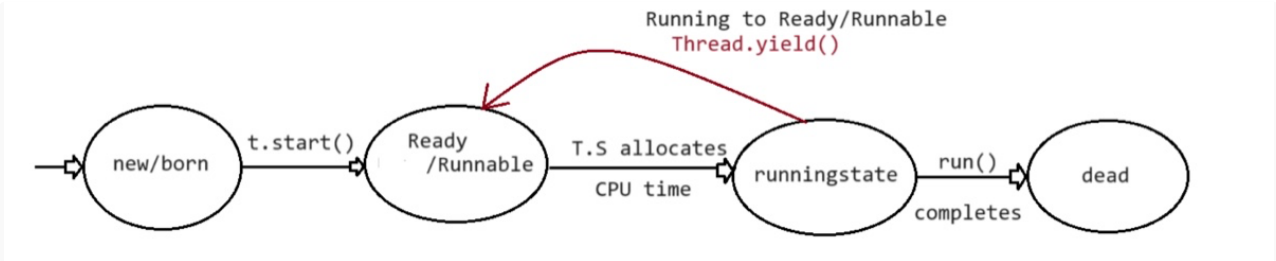
a. yield()

b. sleep()

c. join()

yield()

1. It causes to pause current executing Thread for giving chance for waiting Threads of same priority.
2. **If there is no waiting Threads or all waiting Threads have low priority then same Thread can continue its execution.**
3. If all the threads have same priority and if they are waiting then which thread will get chance we can't expect, it depends on ThreadScheduler.
4. The Thread which is yielded, when it will get the chance once again depends on the mercy on "ThreadScheduler" and we can't expect exactly.
5. public static native void yield()**[native means that the code of yeild will not come from java language, it will come from other languages like c,c++ etc.]**



eg#1.

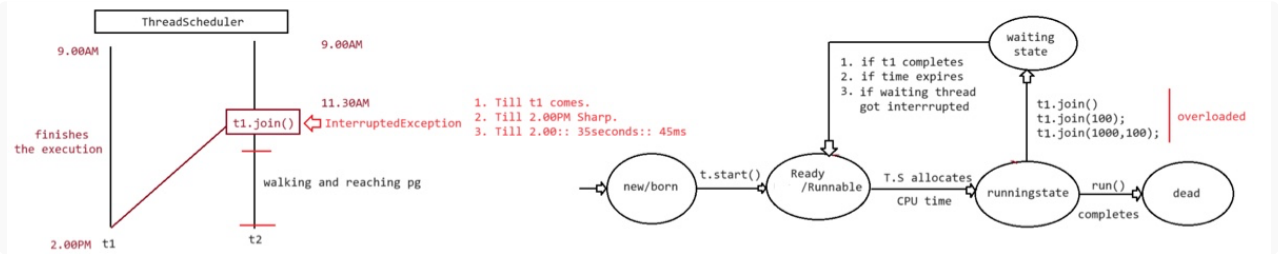
```
class MyThread extends Thread{
    @Override
    public void run(){
        for (int i=1;i<=5 ;i++ ){
            System.out.println("child thread");
            Thread.yield();//line-1
        }
    }
}
public class TestApp{
    public static void main(String... args){
        MyThread t= new MyThread();
        t.start();
        for (int i=1;i<=5 ;i++ ){
            System.out.println("Parent Thread");
        }
    }
}
```

Note:: If we comment line-1, then we can't expect the output becoz both the threads have same priority then which thread the ThreadScheduler will schedule is not in the hands of programmer but if we don't comment line-1, then there is a possibility of main thread getting more no of times, so main thread execution is faster then child thread will get chance.

Note: Some platforms wont provide proper support for yield(),becuase it is getting the execution code from other language prefereably from 'C'.

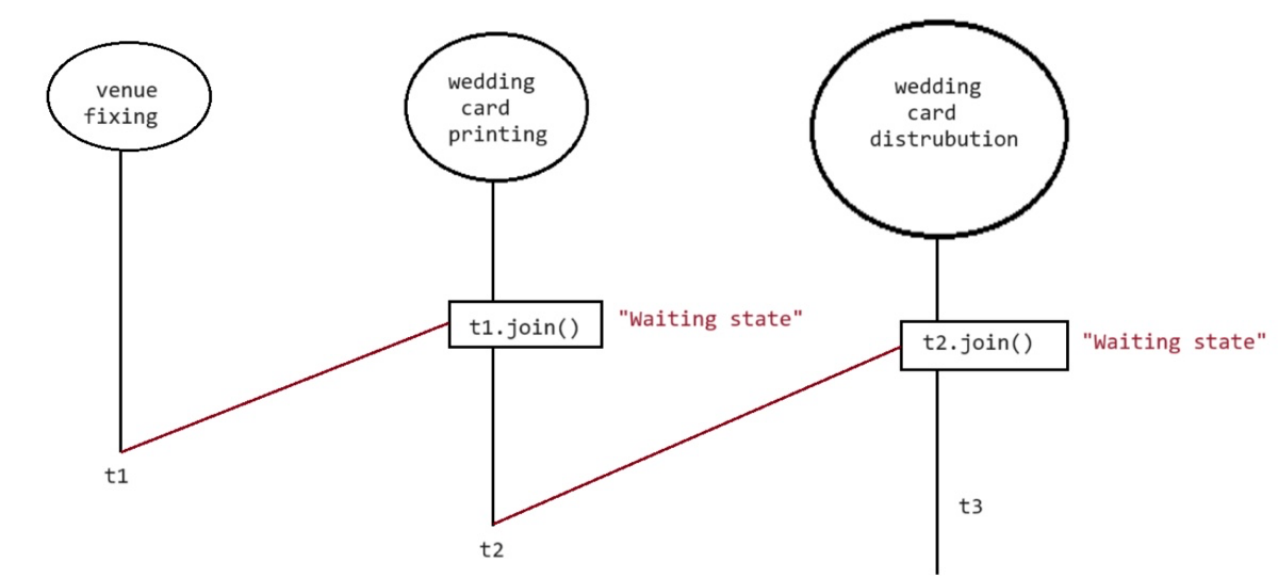
b. join()

If the thread has to wait untill the other thread finishes its execution then we need to go for join().
if t1 executes t2.join() then t1 should wait till t2 finishes its execution.
t1 will be entered into waiting state untill t2 completes, once t2 completes then t1 can continue with its execution.



In the above image t1 and t2 are two threads and t2 needs to wait untill t1 completes its execution(so that both can go to pg together)(so there is a dependency)

Similarly for 3 tasks -->(all three depends on each other as when venue will be fixed then only card will be printed and the card will only be distributed when card is printed.)



eg#1.
venue fixing => t1.start()
wedding card printing => t2.start()=> t1.join()
wedding card distrubution => t3.start()=> t2.join()

Prototype of join()

public final void join() throws InterruptedException

public final void join(long ms)throws InterruptedException

public final void join(long ms,int ns)throws InterruptedException

Note: While one thread is in waiting state and if one more thread interupts then it would result in "InteruptedException".InteruptedException is checkedException which should always be handled. (that's why we added throws InterruptedException)

- The `join()` methods throw an `InterruptedException` if the calling thread is interrupted while waiting.

- Thread interruption is a mechanism to signal a thread that it should stop its current operation. It's achieved using the `interrupt()` method on the thread object.

Thread t =new Thread(); **//new/born state**

t.start(); **//ready/runnable state**

-> If T.S allocates cpu time then Thread enters into running state

-> If currently executing Thread invokes t.join()/t.join(1000),t.join(1000,100), then it would enter into waiting state.

-> If the thread finishes the execution/time expires/interupted then it would come back to ready state/ runnable state.

-> If run() is completed then it would enter into dead state.

eg#1.

```
class MyThread extends Thread{
    @Override
    public void run(){
        for (int i=1;i<=10 ;i++ ){
            System.out.println("Sita Thread");
            try{
                Thread.sleep(2000);
            }
            catch (InterruptedException e){
            }
        }
    }
}
public class Test3 {
    public static void main(String... args)throws InterruptedException{
        MyThread t=new MyThread();
        t.start();
        t.join(10000);//line-n1
        for (int i=1;i<=10;i++ ){
            System.out.println("rama thread");
        }
    }
}
```

=> If line-n1 is commented then we can't predict the output becoz it is the duty of the T.S to assign C.P.U time

=> If line-n1 is not commented, then rama thread(main thread) will enter into waiting state till sita thread(child thread) finishes its execution.

Output

2 Threads

a. Child Thread

sita thread

sita thread

....

b. Main Thread

rama thread

rama thread

....

Waiting of Child Thread untill Completing Main Thread

we can make main thread to wait for child thread as well as we can make child thread also to wait for main thread.

eg#1.

```
class MyThread extends Thread{
    static Thread mt;
    @Override
    public void run(){
        try{
            mt.join();
        }
        catch (InterruptedException e){
        }
        for (int i=1;i<=10 ;i++ ){
            System.out.println("child thread");
        }
    }
}
public class Test3 {
    public static void main(String... args)throws InterruptedException{
        MyThread.mt=Thread.currentThread();
        MyThread t=new MyThread();
        t.start();
        for (int i=1;i<=10;i++ ){
            System.out.println("main thread");
```

```
            Thread.sleep(2000);//20sec sleep
        }
    }
}
```

Here first we want reference of parent(main) that we can acheive with static variable as we can access static variables in static block. Second we need to add throws exception to run, but we cant do that as its parent method should also throw the same and its parent code is pre defined(api). Hence we wrote it inside of try catch block

Output

2 Threads(MainThread,ChildThread)

MainThread

a. main thread

....

....

ChildThread

a. child thread

....

....

eg#2.

```
class MyThread extends Thread{
    static Thread mt;
    @Override
    public void run(){
        try{
            mt.join();
        }
        catch (InterruptedException e){
        }
        for (int i=1;i<=10 ;i++ ){
            System.out.println("child thread");
        }
    }
}
public class Test3 {
    public static void main(String... args)throws InterruptedException{
        MyThread.mt=Thread.currentThread();
        MyThread t=new MyThread();
```

```
            t.start();
            t.join();
            for (int i=1;i<=10;i++ ){
                System.out.println("main thread");
                Thread.sleep(2000);//20sec sleep
            }
        }
}
```

Note::

If both the threads invoke t.join(),mt.join() then the program would result in "deadlock".

eg#3.

```
public class Test3 {
    public static void main(String... args)throws InterruptedException{
        Thread.currentThread().join();
    }
}
```

Output:: Deadlock, becoz main thread is waiting for the main thread itself.

sleep()

If a thread dont' want to perform any operation for a particular amount of time then we should go for sleep().

Signature

public static native void sleep(long ms) throws InterruptedException

public static void sleep(long ms,int ns) throws InterruptedException

Every sleep method throws InterruptedException, which is a checkedexception so we should compulsorily handle the exception using try catch or by throws keyword otherwise it would result in compile time error.

Thread t=new Thread(); **//new or born state**

t.start() **// ready/runnable state**

=> If T.S allocates cpu time then it would enter into running state.

=> If run() completes then it would enter into dead state.

=> If running thread invokes sleep(1000)/sleep(1000,100) then it would enter into Sleeping state

=> If time expires/ if sleeping thread got interrupted then thread would come back to "ready/runnable state".

eg#1.

```
public class SlideRotator {
    public static void main(String... args)throws InterruptedException{
        for (int i=1;i<=10 ;i++ ){
            System.out.println("Slide: "+i);
            Thread.sleep(5000);
        }
    }
}
```

Note:By default what ever property is associated with Parent thread will be shared for Child thread also.(eg. priority number)

Note: join() and sleep() would generate "InterruptedException" which is a fully checkedexception so we need to keep handling code.(not yeild as it is native)

Interupting a Thread

1)- interrupt() is a waste if the thread on which it is applied is not going in waiting or sleeping state. When the thread goes into waiting or sleeping state then immediately it will throw an exception and catch block will be executed.

```
public void interrupt()
```

=> If thread is in sleeping state or in waiting state we can interupt a thread.**(not in running state as only one thread will be there in running state, so no thread will be there to interrupt it.)**

eg#1.

```
class MyThread extends Thread{
    @Override
    public void run(){
        try{
            for (int i=1;i<=10;i++ ){
                System.out.println("I am lazy thread");
                Thread.sleep(2000);
            }
        }
        catch (InterruptedException e){
            System.out.println("I got interrupted");
        }
    }
}
public class Test3 {
    public static void main(String... args)throws InterruptedException{
```

```
        MyThread t=new MyThread();
        t.start();
        t.interrupt(); //line-n1
        System.out.println("End of Main...");
    }
}
```

Scenario:: If a comment line-n1

2 thread

a. Main Thread

End of Main...

b. Child Thread

I am lazy thread

.....

.....

Scneario:: If t.interrupt() then

2 thread

a. Main Thread

main thread

b. Child Thread

I am lazy thread

I got interrupted //→ catch block executed

eg#2.

```
class MyThread extends Thread{
    @Override
    public void run(){
        for (int i=1;i<=10000 ;i++ ){
            System.out.println("I am lazy thread : "+i);
        }
        System.out.println("I am entering into sleeping state");
        try
        {
            Thread.sleep(2000);
        }
        catch (InterruptedException ie){
            ie.printStackTrace();
```

```
        }
    }
}
public class TestApp {
    public static void main(String[] args)throws InterruptedException {
        MyThread t=new MyThread();
        t.start();
        t.interrupt();//line-n1
        System.out.println("main thread");
    }
}
```

line-n1 is commented then no problem

line-n1 is not commented, then interrupt() will wait till the Thread enters into waiting state/sleeping state.

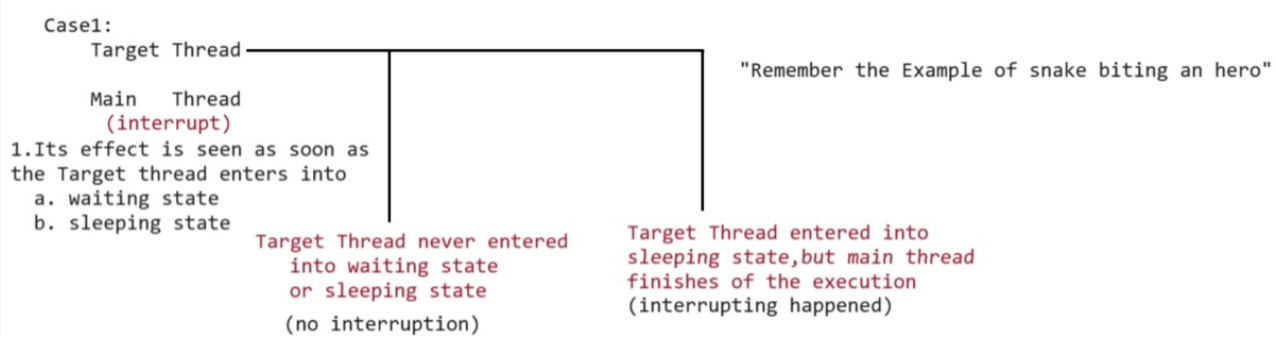
Note::

If thread is interrupting another thread, but target thread is not in waitingstate/sleeping state then there would be no exception.

interrupt() call be waiting till the target thread enters into waiting state/sleeping state so this call wont be wasted.

once the target thread enters into waiting state/sleeping state then interrupt() will interrupt and it causes the exception.

interrupt() call will be wasted only if the Thread does not enters into waiting state/sleeping state.



yield() join() sleep()

1. **Purpose**

   yield()

   To pause current executing Thread for giving the chance of remaining waiting Threads of same priority.

   join()

   If a Thread wants to wait until completing some other Thread then we should go for join.

sleep()

If a Thread don't want to perform any operation for a particular amount of time then we should go for sleep() method.

2. **Is it static**

   yield() yes

   join() no

   sleep() yes

3. **Is it final?**

   yield() no

   join() yes

   sleep() no

4. **Is it overloaded?**

   yield() no

   join() yes

   sleep() yes

5. **Is it throws IE?**

yield() no

join() yes

sleep() yes

6. **Is it native method? //logic of those methods would come from language like 'c'** not from java.

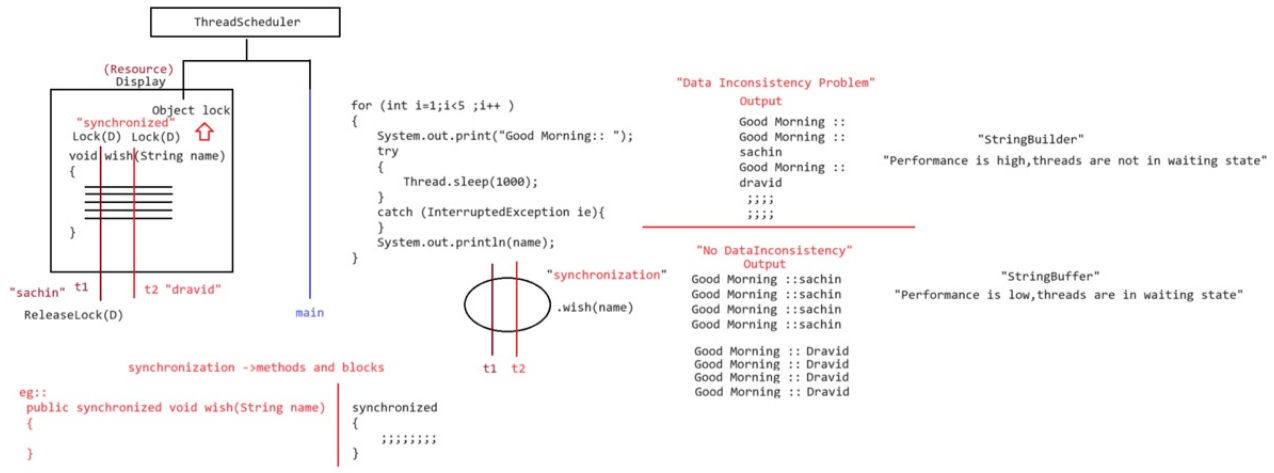
   yield() yes

   join() no

   sleep()

   sleep(long ms) -->native

   sleep(long ms,int ns) -->non-native



Above diagram -→ Multithreading is good for cpu utilization but can create data inconsistencies as if one thread stops immediately another can start without completely finishing the job of last thread.

Sol -→ when one fellow uses a resource, please give that user the resource completely untill he tells the job is done(even does'nt matter if he goes in sleeping or waiting state). Next thread will start only when the last thread completes its work. We can lock the resource("synchronized"). Just add synchronized acess modifier to that method.

Synchronization

1. synchronized is a keyword applicable only for methods and blocks
2. **if we declare a method/block as synchronized then at a time only one thread can execute that method/block on that** object.
3. The main advantage of synchronized keyword is we can resolve data inconsistency problems.
4. **But the main disadvantage of synchronized keyword is it increases waiting time of the Thread and effects performance of the system.**
5. Hence if there is no specific requirement then never recommended to use synchronized keyword.
6. **Internally synchronization concept is implemented by using lock concept.**

```
class X{
    synchronized void m1(){}
    synchronized void m2(){}
    void m3(){}
}
```

KeyPoints

1. if t1 thread invokes m1() then on the Object X lock will applied.
2. if t2 thread invokes m2() then m2() can't be called because lock of X object is with m1.
3. if t3 thread invokes m3() then execution will happen becoz m3() is non-synchronized. **Lock concept is applied at the Object level not at the method level.**
4. Every object in java has a unique lock. Whenever we are using synchronized keyword then only lock concept will come into the picture.
5. If a Thread wants to execute any synchronized method on the given object 1st it has to get the lock of that object. Once a Thread got the lock of that object then it's allow to execute any synchronized method on that object. **If the synchronized method execution completes then automatically Thread releases lock.**
6. **While a Thread executing any synchronized method the remaining Threads are not allowed execute any synchronized method on that object simultaneously. But remaining Threads are allowed to execute any non-synchronized method simultaneously. [lock concept is implemented based on object but not based on method].**

Note:: Every object will have 2 area**[Synchronized area and NonSynchronized area]**

Synchronized Area => write the code only to perform update,insert,delete

NonSynchronized Area => write the code only to perform select operation

```
class ReservationApp{
    checkAvailablity(){
        //perform read operation. // "synchronized" not required
    }
    synchronized bookTicket(){
        //peform update operation
    }
}
```

eg#1.

```
class Display{
    public void wish(String name){
        for (int i=1;i<=10 ;i++ )
        {
            System.out.print("Good Morning: ");
            try{
                Thread.sleep(2000);
                }
            catch (InterruptedException e){
            }
            System.out.println(name);
        }
    }
}
class MyThread extends Thread{
    Display d;
    String name;
    MyThread(Display d,String name){
        this.d=d;
        this.name=name;
    }
    @Override
    public void run(){
        d.wish(name);
    }
}
public class Test3 {
    public static void main(String... args){
```

```
        Display d=new Display();
        MyThread t1= new MyThread(d,"dhoni");
        MyThread t2= new MyThread(d,"yuvi");
        t1.start();
        t2.start();
    }
}
```

Ouput:: As noticed below the output is irregular becoz at a time on a resource called wish() 2 threads are acting simulataneously.

3 Threads

a. Main Thread

b. Child Thread-1

c. Child Thread-2

GoodMorning :GoodMorning : ..

....

....

....

....

....

eg#2.

```
class Display{
    public synchronized void wish(String name){
        for (int i=1;i<=10 ;i++ )
        {
            System.out.print("Good Morning: ");
            try{
                Thread.sleep(2000);
            }
            catch (InterruptedException e){
            }
            System.out.println(name);
        }
    }
}
class MyThread extends Thread{
    Display d;
    String name;
```

```
    MyThread(Display d,String name){
        this.d=d;
        this.name=name;
    }
    @Override
    public void run(){
        d.wish(name);
    }
}
public class Test3 {
    public static void main(String... args)throws InterruptedException{
        Display d=new Display();    //operating on single object
        MyThread t1= new MyThread(d,"dhoni");
        MyThread t2= new MyThread(d,"yuvi");
        t1.start();
        t2.start();
    }
}
```

Ouput::

3 Threads

a. Main Thread

b. Child Thread-1

GoodMorning:dhoni

GoodMorning:dhoni

.....

.....

.....

c. Child Thread-2

GoodMorning:yuvi

GoodMorning:yuvi

....

....

....

Note:: As noticed above there are 2 threads which are trying to operate on single object called "Display" we need synchronization to resolve the problem of "Datainconsistency".

casestudy::

Display d1=new Display();

```
Display d2=new Display();
MyThread t1=new MyThread(d1,"yuvraj");
MyThread t2=new MyThread(d2,"dhoni");
t1.start();
t2.start();
```

In the above case we get irregular output(even after using synchronized keyword in wish method), because two different object and since the method is synchronized lock is applied w.r.t object and both the threads will start simulataneously on different java objects due to which the output is "irregular".

Conclusion :

If multiple threads are operating on multiple objects then there is no impact of Syncronization. If multiple threads are operating on same java objects then syncronized concept is required(applicable).

classlevel lock

1. Every class in java has a unique level lock.
2. **If a thread wants to execute static synchronized method then the thread requires "class level lock".**
3. **While a Thread executing any static synchronized method the remaining Threads are not allowed to execute any static synchronized method of that class simultaneously.**
4. **But remaining Threads are allowed to execute normal synchronized methods, normal static methods, and normal instance methods simultaneously.**
5. **Class level lock and object lock both are different and there is no relationship between these two.**

eg::

```
class X{
    static synchronized m1(){}.   //class level lock
    static synchronized m2(){}
    static m3(){}                 //no lock required
    synchronized m4(){}           //object level lock
    m5(){}                        //no lock required
}
```

t1=> m1() => class level lock applied and chance is given

t2=> m2() => enter into waiting state

t3=> m3() => gets a chance for execution without any lock

t4=> m4() => object level lock applied and chance is given

t5=> m5() => gets a chance for execution without any lock

Program#1

```
class Display
{
    public static synchronized void displayNumbers(){
        for (int i=1;i<=10 ;i++ )
        {
            System.out.print(i);//1 to 10
            try{
                Thread.sleep(2000);
            }
            catch (InterruptedException e){}
        }
    }
    public static synchronized void displayCharacters(){
        for (int i=65;i<=75 ;i++ )
        {
            System.out.print((char)i);//A-K
            try{
                Thread.sleep(2000);
            }
            catch (InterruptedException e){
            }
        }
    }
}
class MyThread1 extends Thread{
    Display d;
    MyThread1(Display d){
        this.d=d;
    }
    @Override
    public void run(){
        d.displayNumbers();
    }
}
class MyThread2 extends Thread{
    Display d;
```

```
    MyThread2(Display d){
        this.d=d;
    }
    @Override
    public void run(){
        d.displayCharacters();
    }
}
public class Test {
    //JVM ---> main thread
    public static void main(String[] args)throws Exception{
        Display d1=new Display();
        MyThread1 t1= new MyThread1(d1);
        MyThread2 t2= new MyThread2(d1);
        t1.start();
        t2.start();
    }
}
```

Output::

3 Threads

a.MainThread

b.userdefinedThread

displayCharacters()

c.userdefinedThread

displayNumbers()

when both the methods are without static and synchronized -> we will get inconsistent output

when both the methods are with static and synchronized -> we will get consistent output

when both the methods are with static(both with synchronized) -> we will get consistent output

when one method is with static and another without static(both with synchronized) -> we will get inconsistent output

synchronized block

If few lines of code is required to get synchronized then it is not recomeneded to make method only as synchronized.
If we do this then for threads performance will be low, to resolve this problem we use "synchronized block", due to synchronized block performance will be improved.

Case Study

If a thread got a lock of current object, then it is allowed to execute that block

a.

```
synchronized(this){
    .....
    .....
    .....
}
```

To get a lock of particular object:: B

b.

```
synchronized(B){
    .....
    .....
    .....
}
```

If a thread got a lock of particular object B, then it is allowed to execute that block.

c. To get class level lock we have to declare synchronized block as follow

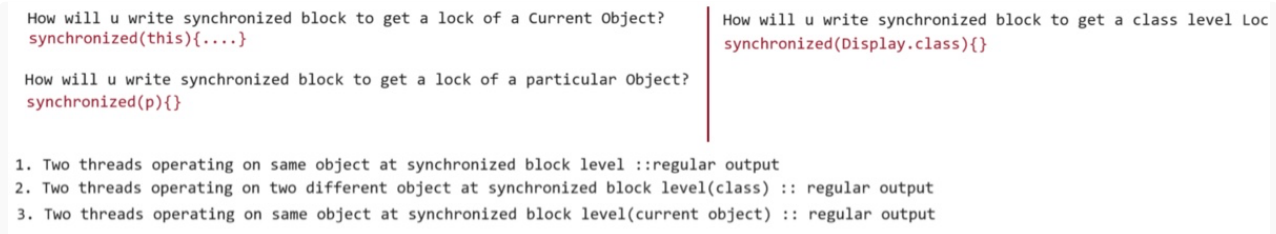
```
synchronized(Display.class){
    ....
    ....
    ....
}
```

If a thread gets class level lock, then it is allowed to execute that block

eg#1.

```
class Display{
    public void wish(String name){
        ;;;;;;;;;;;;; //l-lakh lines of code
        synchronized(this){
            for (int i=1;i<=10;i++ )
            {
                System.out.print("Good morning:");
                try{
                    Thread.sleep(2000);
                }
```

```
                catch (InterruptedException e){}
                System.out.println(name);
            }
        }
        ;;;;;;;;;;;;;;//1-lakh lines of code
    }
}
class MyThread extends Thread{
    Display d;
    String name;
    MyThread(Display d,String name){
        this.d=d;
        this.name=name;
    }
    public void run(){
        d.wish(name);
    }
}
class Test {
    public static void main(String[] args) {
        Display d=new Display();
        MyThread t1=new MyThread(d,"dhoni");
        MyThread t2=new MyThread(d,"yuvi");
        t1.start();
        t2.start();
    }
}
```



1)- two different objects working on same syncronized object level block -→ irregular output -→ solution → class level lock

2)- one object working on same syncronized class level block with method as static -→ regular output

3)- same as first but class level lock -→ regular output

4)- can only be applied of refrence types -->

a)- **int** x = 10;

synchronized(x){} **//compilation error**

b)- **Integer** x = 10;

synchronized(x){} **//will work fine**

eg#2.

```
class Display{
    public void wish(String name){
        ;;;;;;;;;;;;; //l-lakh lines of code
        synchronized(this){
            for (int i=1;i<=10;i++ )
            {
                System.out.print("Good morning:");
            try{
                Thread.sleep(2000);
            }
            catch (InterruptedException e){}
                System.out.println(name);
            }
        }
        ;;;;;;;;;;;;;//1-lakh lines of code
    }
}
class MyThread extends Thread{
    Display d;
    String name;
    MyThread(Display d,String name){
        this.d=d;
        this.name=name;
    }
    public void run(){
        d.wish(name);
    }
}
public class Test {
    public static void main(String[] args) {
```

```
        Display d1=new Display();
        Display d2=new Display();
        MyThread t1=new MyThread(d1,"dhoni");
        MyThread t2=new MyThread(d2,"yuvi");
        t1.start();
        t2.start();
    }
}
```

Output::Irregular output becoz two object and two threads acting on two different objects

eg#3.

```
class Display{
    public void wish(String name){
        ;;;;;;;;;;;;; //l-lakh lines of code
        synchronized(Display.class){
            for (int i=1;i<=10;i++ )
            {
                System.out.print("Good morning:");
                try{
                    Thread.sleep(2000);
                }
                catch (InterruptedException e){}
                System.out.println(name);
            }
        }
        ;;;;;;;;;;;;;//1-lakh lines of code
    }
}
class MyThread extends Thread{
    Display d;
    String name;
    MyThread(Display d,String name){
        this.d=d;
        this.name=name;
    }
    public void run(){
        d.wish(name);
    }
}
```

```
public class Test {
    public static void main(String[] args) {
        Display d1=new Display();
        Display d2=new Display();
        MyThread t1=new MyThread(d1,"dhoni");
        MyThread t2=new MyThread(d2,"yuvi");
        t1.start();
        t2.start();
    }
}
```

Note:: 2 object, 2 thread, but the thread which gets a chance applied class level lock so output is regular.

Note:: lock concept applicable only for objects and class types, but not for primitive types. if we try to do it would result in compile time error saying "unexpected type".

eg:: int x=10;

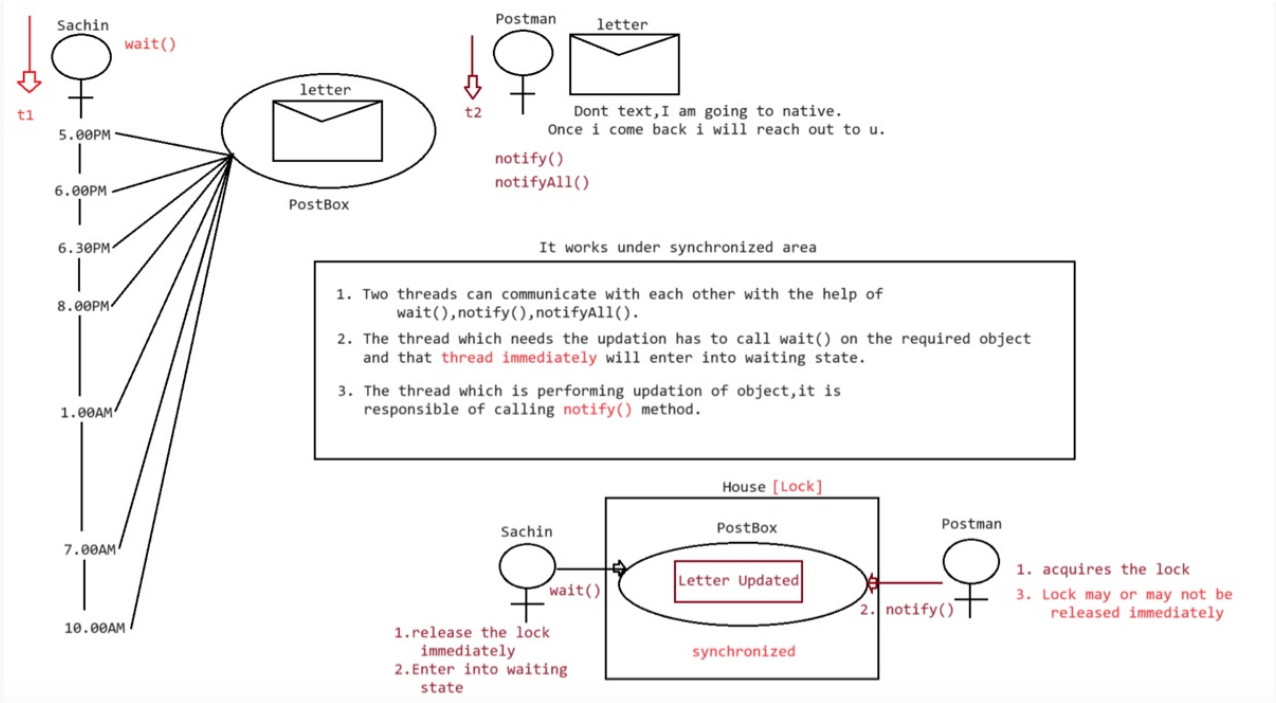
synchronized(x)**{//CE: unexpected type found:int required:reference**

}

Which approach is good in multithreading, writing synchronized methods or synchronized blocks?

Ans. synchronized block is good.

InterThreadCommunication



Story ->

A girl told Sachin that she will send him a letter. Sachin is very excited and waiting for the letter so he go to postbox after every 1hr or half to check weather he receives the letter or not. consider sachin and postman as two different threads, in this way a lot of time is wasted by thread t1.

2nd diagram ->

First sachin is the owner of the house so he has the access to release the lock and enter into waiting state. Since lock is released so postman can access it and first it will acquire the lock and then update the postbox with the letter after that postman may or may not release lock immediately(depends on the nature of postman.). And then will notify to sahin.

InterThreadCommunication(remember postbox example)

Two threads can communicate each other with the help of

a. notify()

b. notifyAll()

c. wait()

notify()=> Thread which is performing updation should call notify(),so the waiting thread will get notification so it will continue with its execution with the updated items.

wait() => Thread which is expecting notification/updation should call wait(), immediately the Thread will enter into waiting state.

wait(),notifyAll(),notify() is present in Object class, but not in Thread class why?

=> Thread will call wait(),notify(),notifyall() on any type of objects like Student,Customer,Engineer.

If a thread wants to call wait(),notify()/notifyall() then compulsorily the thread should be the owner of the object otherwise it would result in "**IllegalMonitorStateException**".

We say thread to be owner of that object if thread has lock of that object.

It means these methods are part of synchronized block or synchronized method, if we try to use outside synchronized area then it would result in RunTimeException called "**IllegalMonitorStateException**".

if a thread calls wait() on any object, then first it immediately releases the lock on that object and it enters into waiting state. if a thread calls notify() on any object,then he may or may not release the lock on that object immediately.

Except wait(),notify(),notifyAll() lock can't be relased by other methods.

Note::

yield(),sleep(),join() => can't release the lock.

wait(),notify(),notifyAll() => will release the lock,otherwise interthread communication can't happen.

Once a Thread calls wait(), notify(), notifyAll() methods on any object then it releases the lock of that particular object but not all locks it has.

Method prototype of wait(),notify(),notifyAll()

1. public final void wait()throws InterruptedException
2. public final native void wait(long ms)throws InterruptedException
3. public final void wait(long ms,int ns)throws InterruptedException
4. public final native void notify()
5. public final void notifyAll()

Program

eg#1.

```
class ThreadB extends Thread{
    int total =0;
    @Override
    public void run(){
        for (int i=0;i<=100 ; i++){
            total+=i;
        }
    }
}
public class Test {
    public static void main(String[] args)throws InterruptedException {
        ThreadB b=new ThreadB();
        b.start();
        stmt-1;
        System.out.println(b.total);
    }
}
```

A. stmt-1 -->

if i replace with Thread.sleep(10000) then thread will enter into waiting state but the updated value could be available at any time, it could be 1 nsec or 10sec, if the updation is not ready, then we should not use Thread.sleep(10000)

B. stmt-2

if i replace with b.join(), then main thread will enter into waiting state,then child will execute for loop,till then main thread has to wait. main thread is waiting for updation result.

```
for (int i=0;i<=100 ; i++){
    total+=i;
}
```

//1cr lines of code is available

main thread has to wait till 1 cr lines of code,y main thread should wait for the complete code to finish.

sol -→ by using wait() and notify()

eg#2.

```
class ThreadB extends Thread{
    int total =0;
    @Override
    public void run(){
        synchronized(this){
            System.out.println("Child thread started calculation");
            for (int i=0;i<=100 ; i++){
                total+=i;
            }
            System.out.println("Child thread trying to give notification");
            this.notify();
        }
    }
}
public class Test {
    public static void main(String[] args)throws InterruptedException {
        ThreadB b=new ThreadB();
        b.start();
        synchronized(b){
            System.out.println("Main thread is calling wait on B object");
            b.wait(10000);//10sec
            System.out.println("Main thread got notification");
            System.out.println(b.total);
        }
    }
}
```

Output

Child thread started calculation

Child thread trying to give notification

Main thread is calling wait on B object for 10sec

Main thread got notification

5050

eg#3. → example of deadlock

```
class ThreadB extends Thread{
    int total =0;
    @Override
```

```
    public void run(){
        synchronized(this){
            System.out.println("Child thread started calculation");
            for (int i=0;i<=100 ; i++){
                total+=i;
            }
            System.out.println("Child thread trying to give notification");
            this.notify();
        }
    }
}
public class Test {
    public static void main(String[] args)throws InterruptedException {
        ThreadB b=new ThreadB();
        b.start();
        Thread.sleep(10000);//10sec
        synchronized(b){
           System.out.println("Main thread is calling wait on B object");
           b.wait();
           System.out.println("Main thread got notification");
           System.out.println(b.total);
        }
    }
}
```

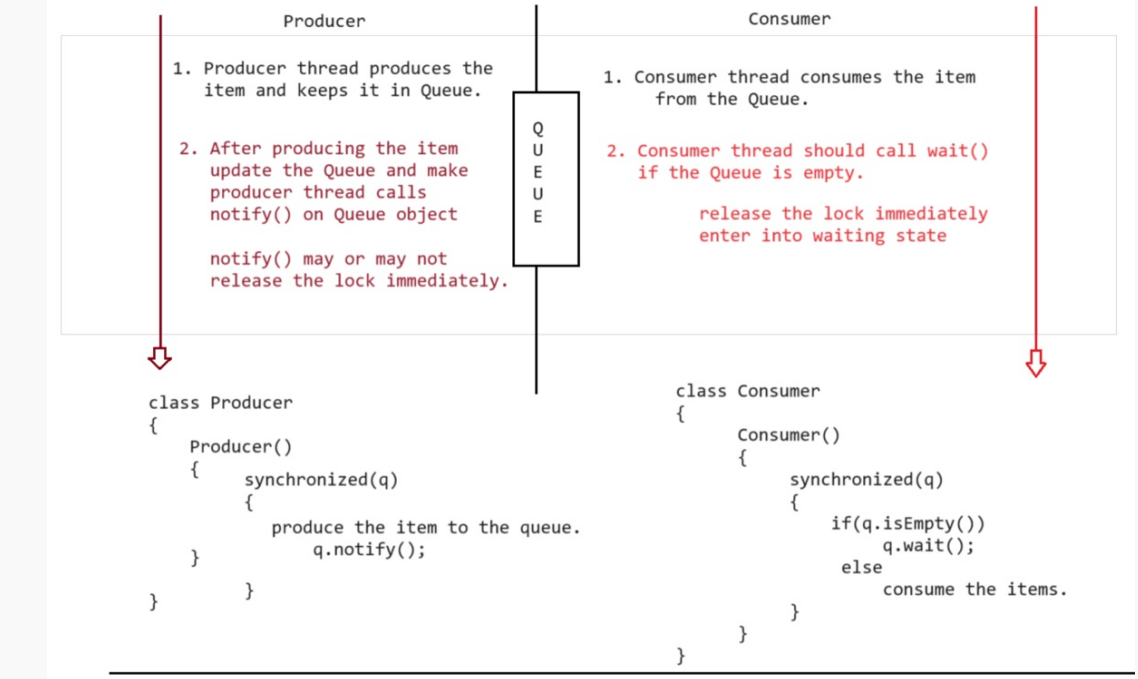
Output

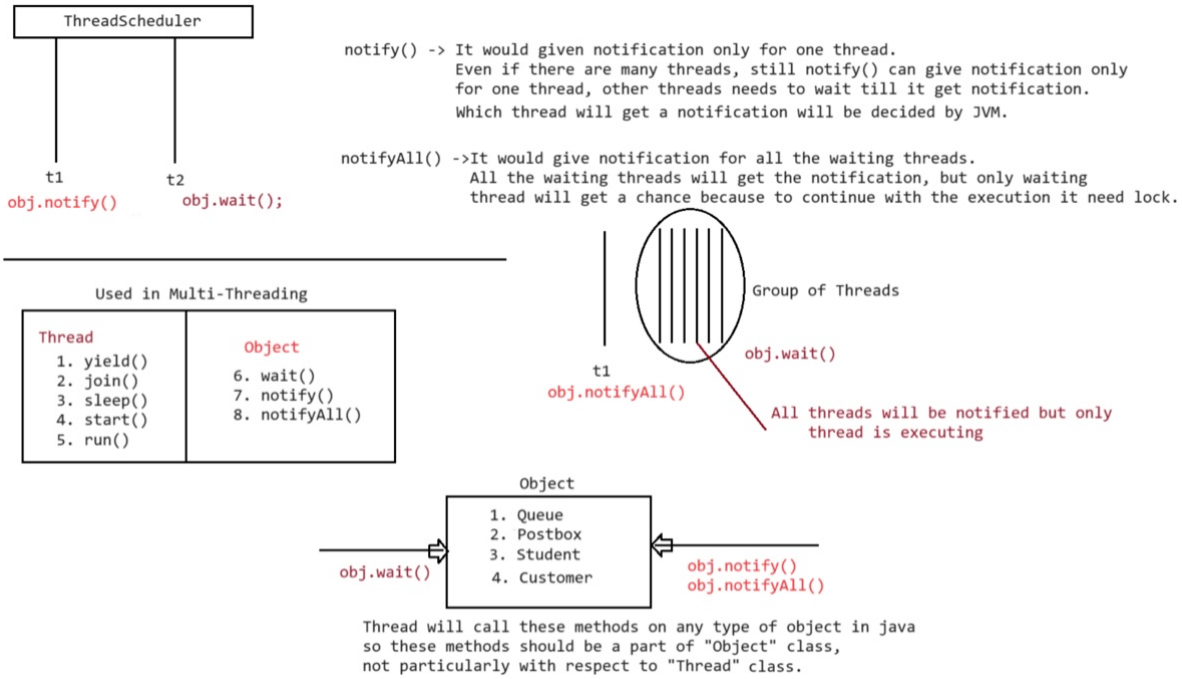
Child thread started calculation

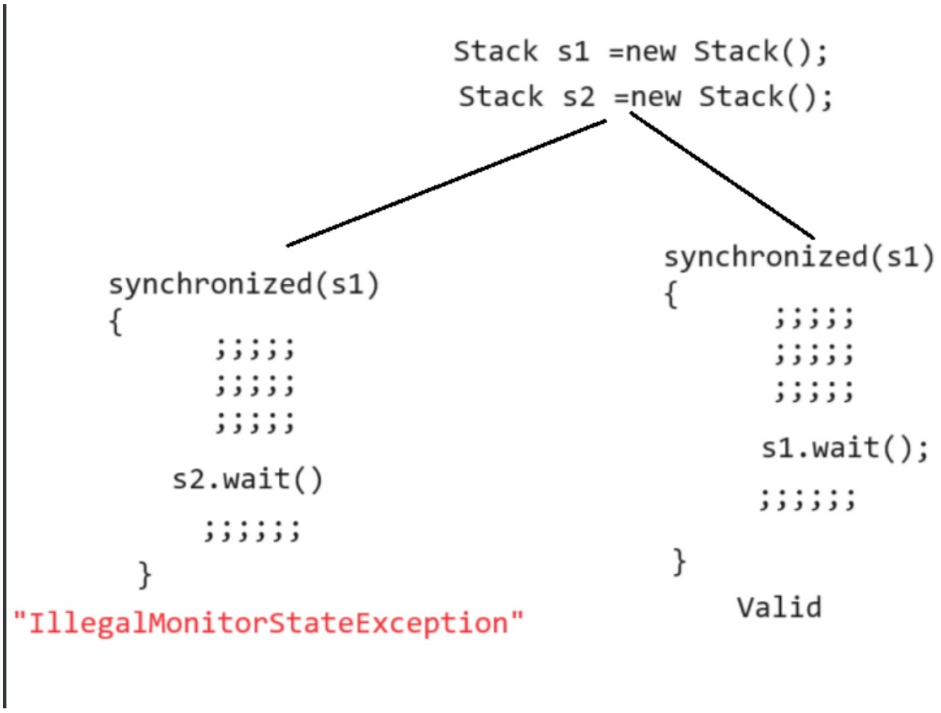
Child thread trying to give notification

Main thread is calling wait on B object becoz of Thread.sleep(10000) main thread will never get notification.

explanation -→ 10000 will put the main thread into sleep state for 10 secs. Till then child code will be executed and it will send the notification. After 10secs since no one has applied the lock so main thread will execute its code and execute b.wait(); and after this it will go to deadlock as it will not any notification.







since the lock is applied on s1 so we can only use wait on s1.

Question based on lock

1. **If a thread calls wait() immediately it will enter into waiting state without releasing any lock.**

   Answer : false(it should be released immediately)

2. **If a thread calls wait() it releases the lock of that object but may not immediately**

   Answer: false(it should be released immediately)

3. **If a thread calls wait() on any object,it releases all locks acquired by that thread and enters into waiting state**

   Answer: false**(it would release the lock of that particular object)**

4. **If a thread calls wait() on any object,it immediately releases the lock of that particular object and entered into waiting state**

   Answer: true

5. **If a thread calls notify() on any object,it immediately releases the lock of that particular object**

   Answer: false(it may or may not release the lock).

6. **If a thread calls notify() on any object,it releases the lock of that object but may not immediately.**

   Answer: true

```
class A
{
    public synchronized void foo(B b)
    {
        System.out.println("Thread1 starts execution of foo() method");
        try
```

```
        {
            Thread.sleep(2000);
        }
        catch (InterruptedException ie)
        {
        }
        System.out.println("Thread1 trying to call b.last()");
        b.last();
    }
    public synchronized void last()
    {
        System.out.println("Inside A, this is the last method");
    }
}
class B
{
    public synchronized void bar(A a)
    {
        System.out.println("Thread2 starts execution of bar() method");
        try
        {
            Thread.sleep(2000);
        }
        catch (InterruptedException ie)
        {
        }
        System.out.println("Thread2 trying to call a.last()");
        a.last();
    }
    public synchronized void last()
    {
        System.out.println("Inside B, this is the last method");
    }
}
public class Test extends Thread{
    //instance area
    A a=new A();
    B b=new B();
```

```
    //instance method
    public void m1(){
        this.start();
        a.foo(b);//executed by main thread
    }

    @Override
    public void run(){
        b.bar(a);//executed by child thread
    }

    //JVM -> main thread
    public static void main(String[] args){
        Test t=new Test();
        t.m1();
    }
}
```

Output

Thread1 starts execution of foo() method

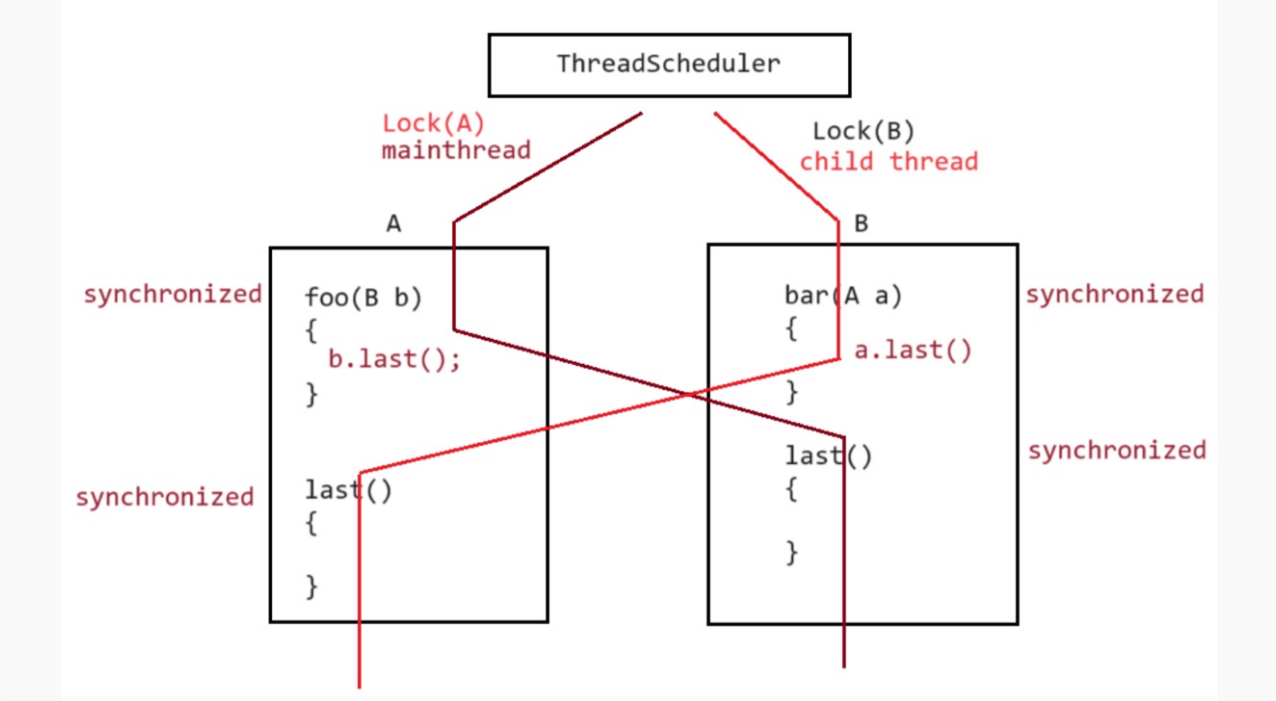
Thread2 starts execution of bar() method

Thread2 trying to call a.last()

Thread1 trying to call b.last()

No deadlock when → only foo method of class A or class B is synchronized, when both foo methods of class A and class B are synchronized, when either one of last method is synchronized(either of class A or B).

Deadlock occurs when → both classes last methods are synchronized.



Explanation

t1(mainthread)

=> starts foo(),since foo() is synchronized and a part of 'A' class so t1 applies lockof(A) and starts the execution, while executing it encounters Thread.sleep().so T.S gives chance for t2 thread.

After getting a chance again by TS, it tries to execute b.last. but lock of b is with t2 thread, so t1 enters into waiting state.

t2=> starts bar(),since bar() is synchronized and a part of 'B' class so t2 applies lockof(B) and starts the execution, while executing it encounter Thread.sleep(),so TS gives chance again for t1 thread.

After getting a chance again by TS, it tries to execute a.last() but lock of a is with t1 thread, so t2 enters into waiting state. Since both the threads are in waiting state and it would be waiting for ever,so we say the above pgm would result in "DeadLock".

Daemon Threads

As soon as main thread is started, main thread starts GarbageCollectorThread as well which can clean garbage object, such types of threads are known as Daemon Thread.

The thread which is executing in the background is called "DaemonThread".

eg: AttachListener,SignalDispatcher,GarbageCollector,....

MainObjective of DaemonThread

The main objective of DaemonThread, to provide support for Non-Daemon threads(main thread).

eg:: if main threads runs with low memory then jvm will call GarbageCollector thread, to destroy the useless objects,so that no of bytes of free memeory will be improved with this free memory main thread can continue its execution.

Usually Daemon threads having low priority,but based on our requirement daemon threads can run with high priority also.

JVM => creates 2 threads

a. Daemon Thread(priority=1,priority=10)

b. main (priority=5)

while executing the main code, if there is a shortage of memory then immediately jvm will change the priority of Daemon thread to 10, so Garbage collector activates Daemon thread and it frees the memory after doing it immediately it changes the priority to 1, so main thread it will continue.

How to check whether the Thread is Daemon or not?

public boolean **isDaemon**() => To check wheter the thread is "Daemon"

public void setDaemon(boolean b) throws **IllegalThreadStateException**

b=> true,means the thread will become Daemaon,**before starting the Thread we need to make the thread as "Daemon" otherwise it would result in "IllegalThreadStateException".**

What is the deafult nature of the Thread?

Ans. By deafult the main thread is "NonDaemon". for all remaining thread Daemon nature is inherited from Parent to child, that is if the parent thread is "Daemon" then child thread will become "Daemon" and if the parent thread is "NonDaemon" then automatically child thread is also "NonDaemon".

Is it possible to change the NonDameon nature of Main Thread?

Ans. Not possible, becoz the main thread starting is not in our hands, it will be started by "JVM". (also at first jvm execute main thread so after execution of a thread we can't make it daemon)

eg::

```
class MyThread extends Thread{}
    public class Test {
        public static void main(String[] args){
            System.out.println(Thread.currentThread().isDaemon());//false
            Thread.currentThread().setDaemon(true);//RE:IllegalThreadStartException
            MyThread t=new MyThread();
            System.out.println(t.isDaemon());//false
            t.setDaemon(true);
            t.start();
            System.out.println(t.isDaemon());//true
        }
```

```
}
```

Note:: Whenever last NonDaemon threads terminates,automatically all Daemon Threads will be terminated irrespective of their position.

eg::

```
class MyThread extends Thread{
    public void run(){
        for (int i=1;i<=10 ;i++ ){
            System.out.println("child thread");
            try{
                Thread.sleep(2000);//2sec
            }
            catch (InterruptedException e){
                System.out.println(e);
            }
        }
    }
}
public class Test {
    public static void main(String[] args){
        MyThread t=new MyThread();
        t.setDaemon(true);//stmt-1
        t.start();
        System.out.println("end of main thread");
    }
}
```

Output: if we comment stmt-1, then both the threads are NonDaemon threads it would continue with its execution.

```
end of main thread
child thread
child thread
...
...
...
```

Output

If we remove comment on stmt-1, then main thread is NonDaemon thread where as userdefined thread is DaemonThread, if the main thread finishes the execution then automatically the DaemonThread also will finish the execution.

Race condition → when multiple threads are in line to use a resource is called race condition.

ThreadGroup

Based on functionality,we can group the threads into single unit is called "ThreadGroup". ThreadGroup contains a group of threads, in addition to threads the thread group can also contain subthread groups

ThreadGroup

t1,t2,t3,t4,.... tn

tx,ty,tz(subthread group)

ta,tb,tc(subthread group)

Advantage

We can perform common operations easily(remeber the example of whatsapp group)

EveryThread in java belongs to some ThreadGroup

eg::

```
public class Test {
    public static void main(String[] args){
        System.out.println(Thread.currentThread().getThreadGroup().getName());//main
        System.out.println(Thread.currentThread().getThreadGroup().getParent().getName());
        //System
    }
}
```

main() is called main thread, main thread belongs to a group called "main". **for every thread group there would be parent group called "System".**

As how for every class there is a parent class called "Object", similarly every thread in java belongs to some Thread group,every Threadgroup in java is the child group of System group directly or indirectly.

System group contains several SystemLevelThreads

a. Finalizer(GarbageCollector)

b. ReferenceHandler

c. SignalDispatchter

d. AttachListener

....

....

z. mainthreadgroup

a. Thread-0

b. Thread-1

....

....

z. subthreadgroup

Thread-0

Thread-1

.....

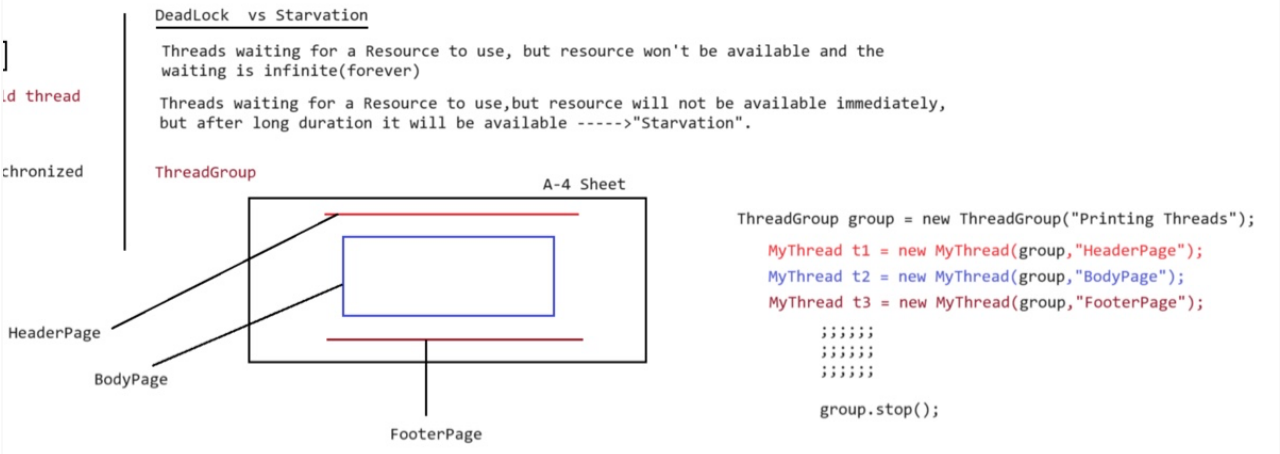
To create ThreadGroup in java, we need use ThreadGroup class constructor

a. ThreadGroup is class present in "java.lang" pacakge and it is the direct child class of "Object".

Constructors

ThreadGroup t=new ThreadGroup(String name);

eg:: ThreadGroup g1=new ThreadGroup("firstGroup");



How to kill a Thread in the middle of the execution?

stop()-> This method is used to kill the thread in the middle of execution. It would enter into dead state immediately. **This method is deperecated in java and it is not recomended to use.**

How to suspend() and resume() of a Thread?

=> remember the example of governement office guy getting suspend and later on getting resumed for his work.

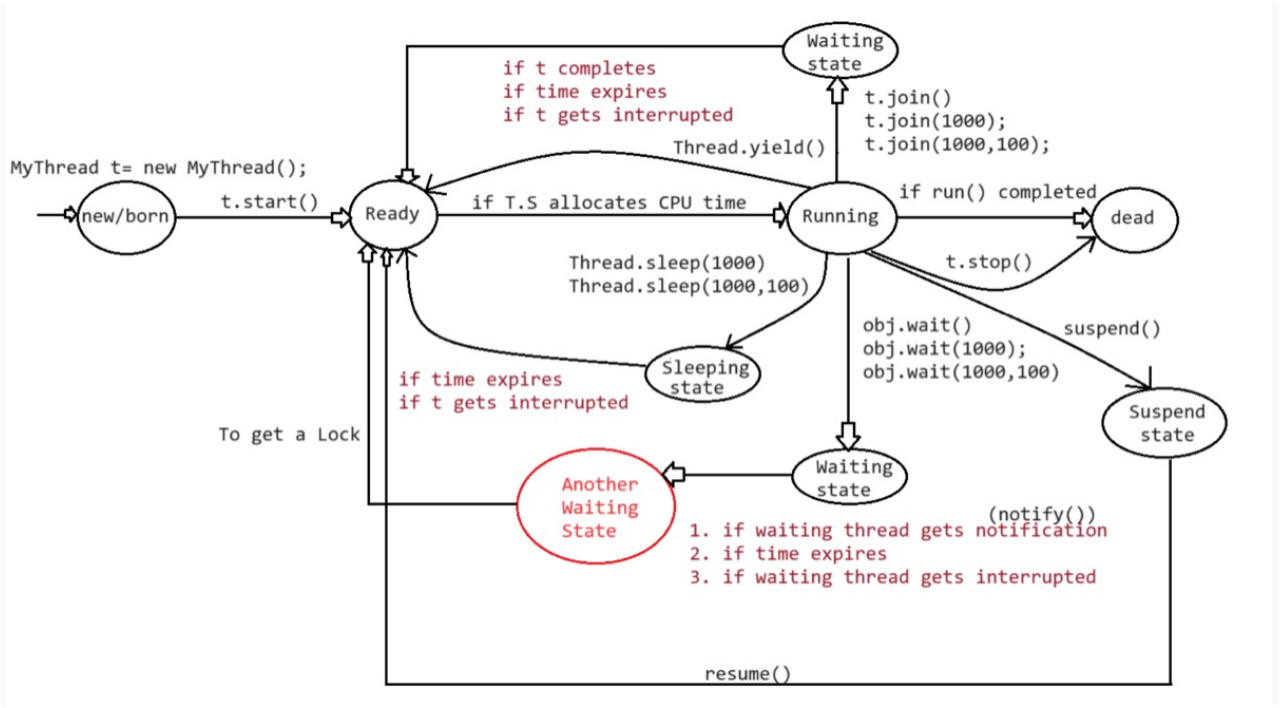
public void suspend()

public void resume()

=> we can suspend a thread by using suspend() method call of Thread class,then thread will enter into suspended state.

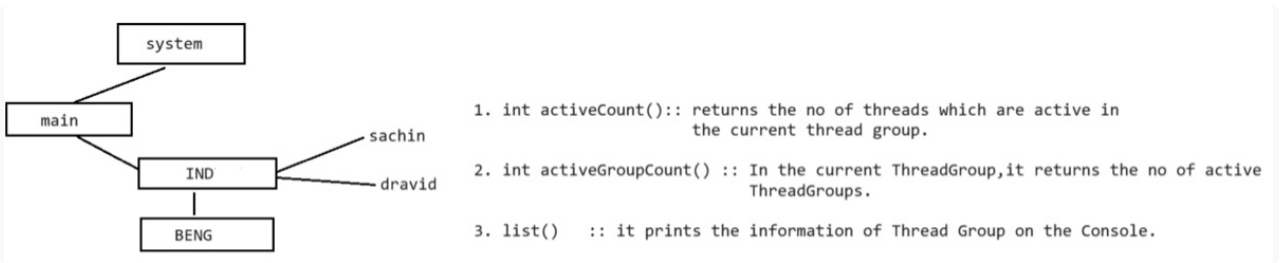
=> we can resume a thread by using resume() method call of Thread class,then suspended thread can contain its execution.

=> **All these methods are deprecated method of Thread class, it is not a good practise to use in our programming.**



For above diagram -→ Thread.yeild() is for below line and three points in red is for line above it(three points is not for Thread.yield()).

ThreadGroup



In above image main parent is System, Beng parent is Ind and Ind parent is.main as a thread group.

```
class MyThread extends Thread
{
    MyThread(ThreadGroup g, String name)
    {
        super(g,name); //passing name and g to Thread class to form a group.
    }
    @Override
    public void run()
    {
        System.out.println("Child Thread");
        try
        {
```

```
                Thread.sleep(2000);
            }
            catch (InterruptedException ie)
            {
            }
        }
}
public class Test{
    //JVM -> main thread created and started
    public static void main(String[] args)throws Exception{
        ThreadGroup indGroup = new ThreadGroup("IND");
        System.out.println(indGroup);
        ThreadGroup benGroup = new ThreadGroup(indGroup,"BENG");
        System.out.println(benGroup);

        MyThread t1 =new MyThread(indGroup,"sachin");
        MyThread t2 =new MyThread(indGroup,"dravid");

        System.out.println(t1);
        System.out.println(t2);
        System.out.println();

        t1.start();
        t2.start();

        System.out.println("Active Threads in a Group:: "+indGroup.activeCount());
        System.out.println("Active ThreadGroup :: "+indGroup.activeGroupCount());
        indGroup.list();

        Thread.sleep(5000);

        System.out.println("Active Threads in a Group:: "+indGroup.activeCount()); //0
        indGroup.list();
        System.out.println("End of main method...");
    }
}
```

Output

java.lang.ThreadGroup[name=IND,maxpri=10]

java.lang.ThreadGroup[name=BENG,maxpri=10]

Thread[sachin,5,IND]

Thread[dravid,5,IND]

Child Thread

Child Thread

Active Threads in a Group:: 2

Active ThreadGroup :: 1

java.lang.ThreadGroup[name=IND,maxpri=10]

Thread[sachin,5,IND]

Thread[dravid,5,IND]

java.lang.ThreadGroup[name=BENG,maxpri=10]

Active Threads in a Group:: 0

java.lang.ThreadGroup[name=IND,maxpri=10]

java.lang.ThreadGroup[name=BENG,maxpri=10]

End of main method...

MyThread t1 =new MyThread("sachin"); -→ By default if it is executed by main then it is part of main thread group.

MyThread t1 =new MyThread(indGroup,"sachin"); → **after specifing the first param it comes under indGroup group.**

All the threads which comes under a thread group will have the same priority as that of that group. If we change the priority of the thread group than all its threads will have same priority.

Since for both the threads sleep time is 4000(2000+2000), so if we put Thread.sleep(5000); for main thread and then see number of active threads it will come as zero as after 5 secs indGroup threads will complete there work.

eg#2.

```
public class Test{
    //JVM -> main thread created and started
    public static void main(String[] args)throws Exception{
        ThreadGroup system=Thread.currentThread().getThreadGroup().getParent();
        System.out.println(system);
        Thread[] t= new Thread[system.activeCount()];
        //Copy all active subThreadGroups into ThreadGroup Array
        system.enumerate(t);
        for ( Thread obj:t)
        {
```

```
            System.out.println(obj);
            System.out.println(obj.getName()+" Is Daemon:: "+obj.isDaemon());
            System.out.println();
        }
    }
}
```

Output

java.lang.ThreadGroup[name=system,maxpri=10]

Thread[Reference Handler,10,system]

Reference Handler Is Daemon:: true

Thread[Finalizer,8,system]

Finalizer Is Daemon:: true

Thread[Signal Dispatcher,9,system]

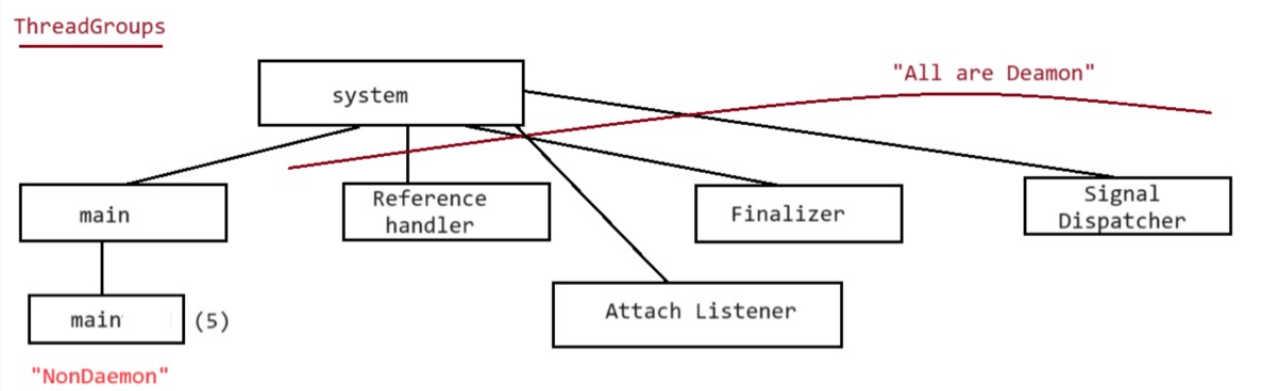
Signal Dispatcher Is Daemon:: true

Thread[Attach Listener,5,system]

Attach Listener Is Daemon:: true

Thread[main,5,main]

main Is Daemon:: false



ThreadLocal

=> It provides ThreadLocal varaibles.

=> It maintains a value on the basis of Threads.

=> Each ThreadLocal varaible maintains a seperate value like userId,transactionId for each thread which can access the object.

=> Thread can access ThreadLocal value, it can manipulate and it can also remove the value from ThreadLocal object.

=> A thread can access its own ThreadLocal varaible value, but not other threads Local varaibles.

=> Once the Thread enters into dead state, the Thread Local variable by default will be eligible for Garabage Collection.

eg#1.

```
public class Test{
    //JVM -> main thread created and started
    public static void main(String[] args)throws Exception{
        ThreadLocal tl = new ThreadLocal(){
            @Override
            protected Object initialValue(){
                System.out.println("Method getting called...");
                return "sachin";
            }
        };
        System.out.println("Getting the TL variable :: "+tl.get());
        System.out.println();
        tl.set("dhoni");
        System.out.println("Getting the TL variable :: "+tl.get());
        System.out.println();
        tl.remove();
        System.out.println("Getting the TL variable :: "+tl.get());
    }
}
```

Output

Method getting called...

Getting the TL variable :: sachin

Getting the TL variable :: dhoni

Method getting called...

Getting the TL variable :: sachin