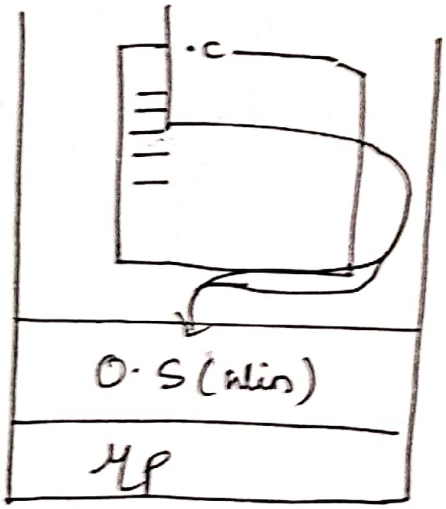
Module - 4

Exception Handling

Exception Handling Fundamentals

Exception : They are the mistakes which occurs at the runtime during the execution of many programs.

Case 1:

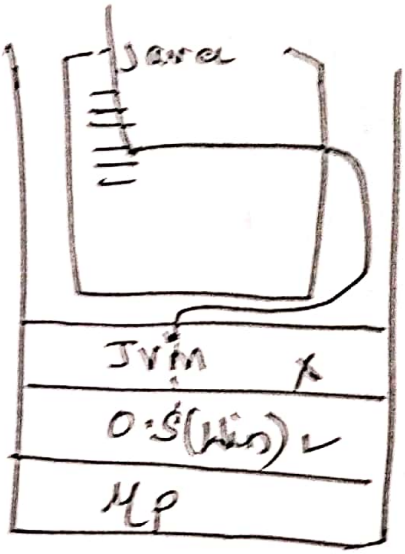


"Corruption or Crash"

"C pgm not safe"

Here, in C when we are compiling if any thing went wrong then it directly goes to O·S, so OS will be not be safe.

Case 2:

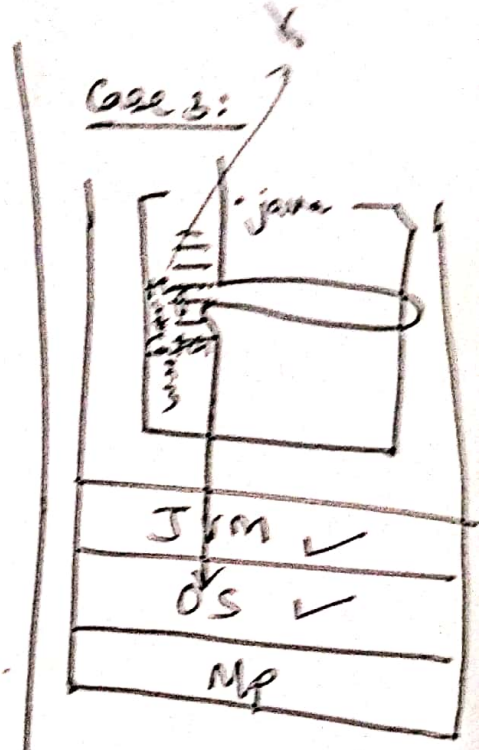


"OS not crashed"

"Mistake or problem is solved by JVM"

Here in Java, if we are not using Exception i.e try & catch, if any error occurs it should be solved by JVM and then JVM will send it to O/S. So JVM will be harmed

Case 3:



"Mistake or Problem is solved by Programmer"

Here in Java, if we are using Exception then if any error occurs it will be solved by programmer and then sent to JVM smoothly and finally sent to O.S

Scanned with CamScanner

Case 2:

```
import java.util.Scanner
Public Class Demo1
{
  Public static void main(String[] args)
  {
    System.out.println("Executing Normally \n");
    Scanner scan = new Scanner(System.in);
    System.out.println("Enter the value of a:");
    int a = scan.nextInt();
    System.out.println("Enter the value of b:");
    int b = scan.nextInt();

    int c = a/b;
    System.out.println("The result is :"+c);
    System.out.println("Exited Normally");
  }
}
```

O/P: Executing Normally
Enter the value of a:
10
Enter the value of b:
0
Exception in thread

try & catch keyword comes in pair

```
try
{
} // Block of code to try
catch
{
} // Block of code to handle errors
```

try → A try allows you to define block of code to be tested for errors while it is being executed

catch → statement allows you to define a block of code to be executed, if an error occurs in the block

Scanned with CamScanner

Case 3:

```
try
{
 int c = a/b;
 System.out.println ("The result is :"+c);
}
 Catch (Exception e)
 {
  System.out.println ("Exception handled by
                                    user \n");
 }
  System.out.println ("Exited Normally");
 }
}
```

O/P: Executed Normally
Enter the value of a:

10
Enter the value of b:

20
The result is: 0
Exited Normally

Consequences of an Uncaught Exception

if the exceptions are not handled by the programmer then it would result in abnormal termination of a code.

Scanned with CamScanner

```
Public class Exception1
{
  Public static void main (String[] args)
  {
   System.out.println ("Connecting to database!!!");
   int arr[] = new int[5];
   for (int i=1 ; i<=6 ; i++)
   {
     arr[i] = i;
   }
   System.out.println ("Closing the database!!!");
  }
}
```

O/P: Executing ~~Normally~~
Enter the value of a:
connecting to database!!!
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 5
at Exception1.main(Exception1.java:15)

In the above program, the program is connecting to database, but during the execution because of the faulty coding, the program would result in abnormal termination as a result of which closing the database connection is not executed. Therefore the programmer should compulsorily handle the exception.

Scanned with CamScanner

Using Multiple catch clauses

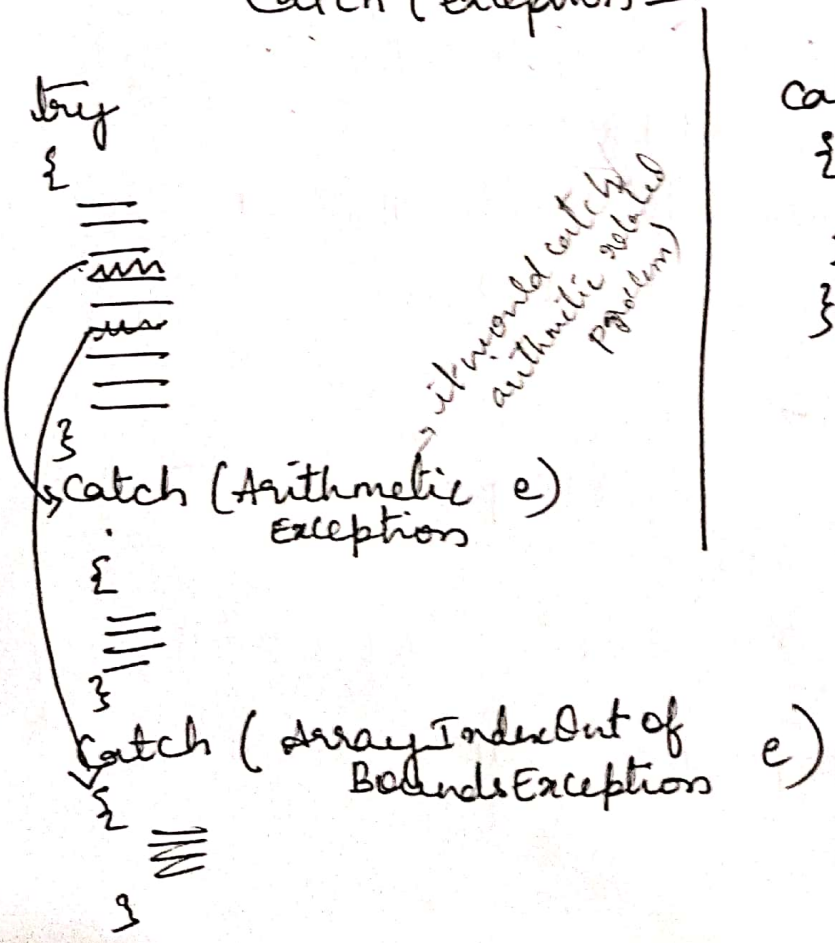
→ if we want to perform different tasks at the occurance of different Exceptions then you can associate more than one catch clause with a try.

→ Each catch must catch a different type of exception.

→ At a time only one exception is occured and at a time one catch block is executed.

Syntax:

```
try
{
  // code that may throw exception
}
Catch (Exception_class_name1 ref1) { //code //
Catch (Exception_class_name2 ref 2){ // code //
```



Scanned with CamScanner

→ A single try block can have multiple catch block

→ However, in the catch block ladder always a specific catch block should be kept followed by generic catch block.

→ if a generic catch block is kept at the top a catch block ladder then it would throw a compile time error.

```
import java.io.FileReader;
import java.util.Scanner;

Public class TryCatchApp
{ Public static void main(String[] Args)
  try
  {
   Scanner scan = new Scanner(System.in);
   System.out.println("Enter the value of a:");
   int a = scan.nextInt();
   System.out.println("Enter the value of b:");
   int a = scan.nextInt();
   int c = a/b;
   int arr[] = new int[5]
   int arr[6] = 25;
   FileReader Fr = new FileReader("E:\\javapgm\\java.text");
   System.out.println("File Created");
  }
```

Scanned with CamScanner

```
Catch (ArithmeticException e)
{
System.out.println("Arithmetic exception occured");
}
Catch (ArrayIndexOutOfBoundException e)
{
System.out.println("Array Exception
                    occured");
}
catch(Exception e)
{
System.out.println ("Exception occured");
}
}
}
```

O/P:

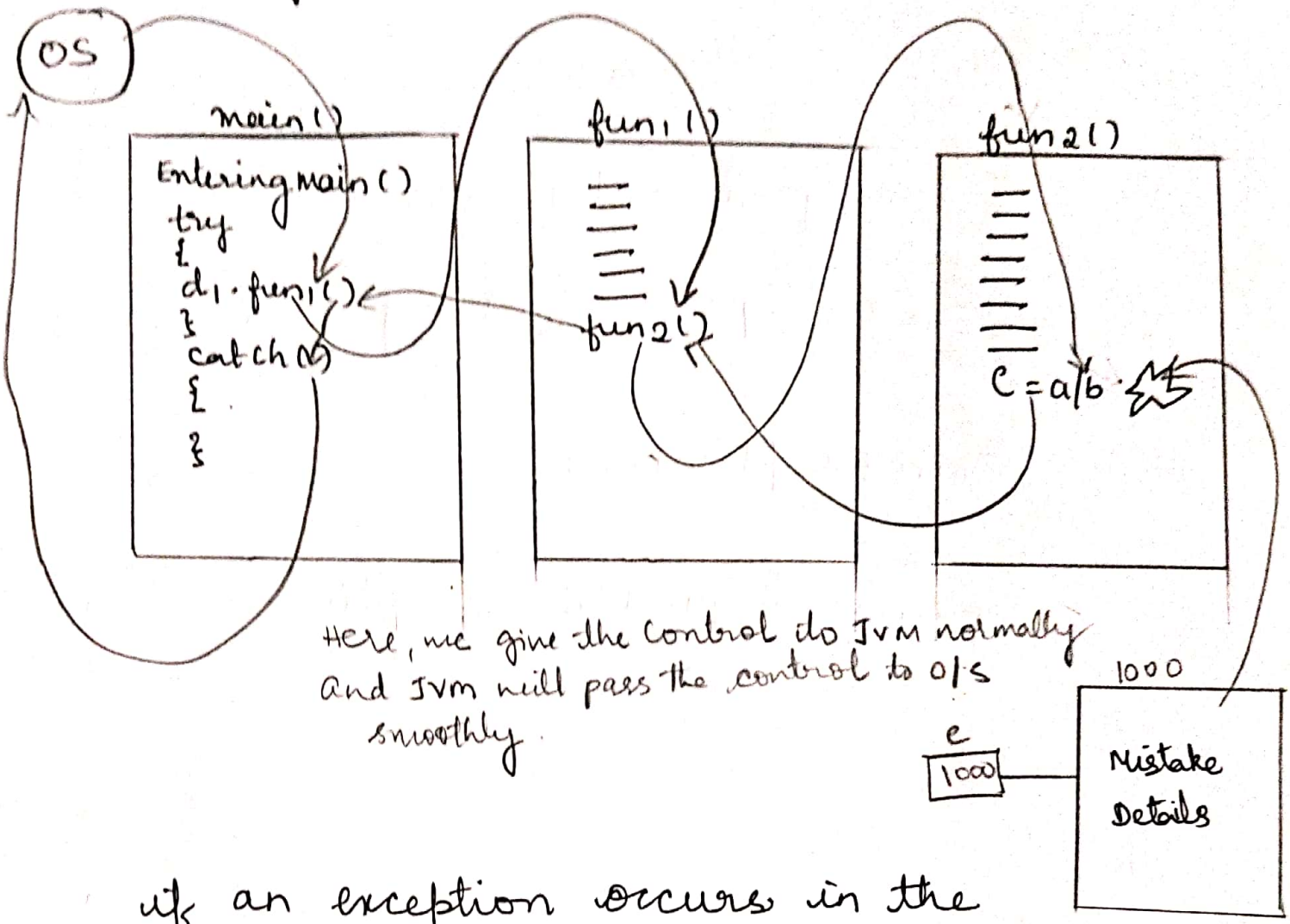
Enter the value of a:
10
Enter the value of b
0
Arithmetic exception occurred

Enter the value of a
10
Enter the value of b
2
Array Exception occurred (if that is also correct

Exception occured.

Scanned with CamScanner

Throwing an Exception



if an exception occurs in the called method (fun2) and if the user doesn't handles the exception then the exception object would be thrown back to the caller(fun1). This mechanism is technically reffered as "Ducking".

→ "throw" keyword is used to explicitly throw an exception.

Scanned with CamScanner

```
class Demo
{
 Public void fun1()
 {
  System.out.println("Entering fun1()");
  fun2();
 }
 Public void fun2()
 {
  System.out.println("Entering fun2()");
  int a=10;
  int b=0;
  int c=a/b;
  System.out.println(c);
 }
}
Public class ThrowingExceptionApp
{
 Public static void main(String[] args)
 {
  System.out.println("Entering main()");
   Demo d1 = new Demo();
   try
   {
     d1.func1();
    {catch(Exception e)
     {System.out.println("Exception Occured");
     }
     System.out.println("Exiting main()");
   }}
 }
```

Scanned with CamScanner

O/P:

```
Entering main()
Entering func1()
Entering func2()
Exception occured
Exiting Main()
```

Rethrowing an Exception

Whenever exception occurs in called method it is the duty of the called method to handle the exception as well as inform the exception to the caller. This process is refered as rethrowing an exception. it is done using throw keyword in java

Eg:

```
class Demo
{
Public void fun1
{
System.out.println("Entering fun1()");
try
 {
  fun2();
 }
 Catch (Exception e)
 {
 System.out.println("Exception handled
                     in fun1()");
  throw e;
 }
```

Scanned with CamScanner

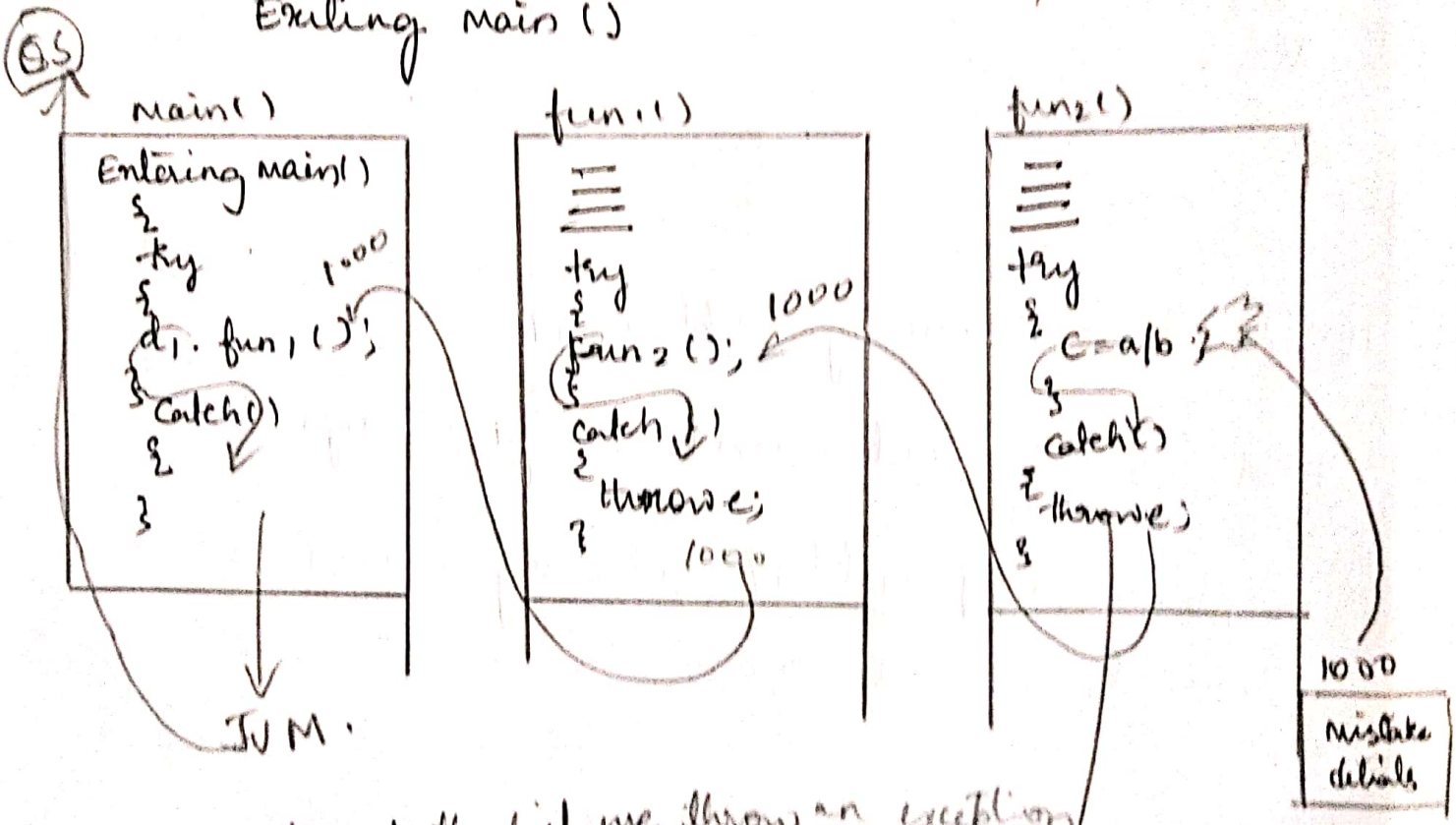
```
Public void fun2()
{
 System.out.println("Entering the fun2()");
 try
 {
  int a=10;
  int b=0;
  int c=a/b;
  System.out.println(c);
 }
 Catch(Arithmetic Exception e);
 {
  System.out.println("Exception handled in fun2()");
 throw e;
 }
}
}
Public class ThrowingExceptionsApp
{
  Public static void main(String[] args)
  {
   System.out.println("Entering main()");
   Demo d1 = new Demo();
   try
   {
    d1.fun1();
   }
```

Scanned with CamScanner

```
catch (Exception e)
{
System.out.println("Exception resolved in main()");
}
System.out.println("Exiting main()");
}
}
```

o/p:
```
Entering main()
Entering fun1()
Entering fun2()
Exception handled in fun2()
Exception handled in fun1()
Exception resolved in main()
Exiting main()
```

This is the best approach



In the main method if we throw an exception object, then the exception object would be handled by JVM

explicitly will be sent by the user.

Scanned with CamScanner

Using Finally

→ Used to execute important code such as closing connection

→ Always executed whether exception is handled or not

→ Must be followed by try or catch block

→ The finally clause is optional. However, each try block there can be zero or more catch blocks, but only one finally block.

Syntax

```
try
{
// code
}
catch (Exception-class-Name, ref1) { // code }
catch (Exception-class-Name2 ref2){ // code }
finally { // code }
```

Eg:

```
class Demo;
{
public void fun1()
{
System.out.println("fun1 connecting to database");
System.out.println("fun1 connection to database");
try
{
fun2();
}
```

Scanned with CamScanner

```
catch (Exception e)
{
System.out.println ("Exception handled in fun2");
throw e;
}
finally
{
System.out.println ("Fun2 closing the connection");
System.out.println ("Fun2 closed the connection");
}
}
public void fun2()
{
System.out.println("fun2 connecting to database");
System.out.println("fun2 connected to database");
try
{
int a=10;
int b=0;
int c=a/b;
}
catch (Exception e)
{
System.out.println("Exception handled in fun2()");
throw e;
}
finally
{
System.out.println("fun2 closing the connection");
System.out.println("fun2 closed the connection");
}}
```

whenever the exception occurs and if the exception object is caught in the catch block, if the exception object is thrown then all the stmts which is written below throw would not be executed. In order to execute those stmt even upon the throw keyword, then such stmts should be placed in finally block.

Scanned with CamScanner

```
Public class FinalApp
{
 Public static void main (String[] args)
{
 System.out.println("main() connecting to database");
System.out.println("main() connected to database");
 try
 {
  Demo1 d1 = new Demo1();
   d1.fun1();
   }
   catch (Exceptions e)
   {
   System.out.println("Exception handled in main()");
   }
   finally                                          ;
   {
   System.out.println("main() closing the database");
   System.out.println("Main() closed the database");
  }
}
```

O/P: Main() connecting to database
" connected " "
Fun1() connecting " "
" connected " "
Fun2() connecting " "
" connected " "
Exception handled in fun2()
Fun2() closing the database connection
" closed " "
Exception handled in fun1()

Scanned with CamScanner

fun, closing the database connection
" closed " "
Exception handled in main ()
Main () closing the database
" closed the database

Difference b/w final, finally and finalize.

Final

→ Final is used to apply restriction on class, method and variable
→ Final class can't be inherited
→ Final class can't be overridden
→ Final variable value can't be changed
→ "Final" is a keyword

Finally

→ it is used to place important code
→ it will be executed whether exception is handled or not
→ "Finally" is a block

Finalize

→ it is used to perform clean up processing and before object in garbage collected
→ "finalize" is a method

Scanned with CamScanner

Throws

In java exception object can also be thrown using "throws" keyword

Eg:
```
public class SampleApp
{
  Public static void main (String[] args) throws Exceptions
  {
   System.out.println("Entering main()");
   int a=10;
   int b=0;
   int c=a/b;
   System.out.println("Exiting main()");
  }
}
```

O/P: Entering main()
Exception in thread "main" java.lang.ArithmeticException:
/by zero at SampleApp.main(SampleApp.java:10)

Difference b/w throw and throws

throw	throws
1. Used to explicitly throw an exception	used to declare an exception
2. Checked exception cannot be propaged using throw only	checked exception can be propaged with throws
3. Followed by an instance	Followed by class

Scanned with CamScanner

4. used within the method
5. cannot throw multiple Exception

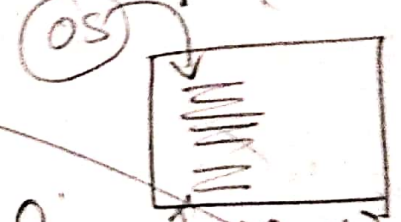
used with the method signature
can declare multiple Exceptions
Eg: public void method() throws IOException, SQLException.

Catching subclass Exception

it refers to process of writting the exception according to the needs of clients requirements -

1. Create a class with a exception name by inheriting the exception class.
2. override get message method according to the client requirement

Eg:
```
import java.util.Scanner;
class AlessthanBException extends Exception
{
  @override
  Public String getMessage()
  {
    return "Subtraction not possible because a<b"
  }
}
Public class customException
{
 Public static void main(String[] args)
 {
```



Scanned with CamScanner

```
Scanner Scan = new Scanner (System.in);
System.out.println ("Enter the value of a :");
int a = Scan.nextInt ();
System.out.println ("Enter the value of b:");
int b = Scan.nextInt ();
if (a<b)
{
 try
 {
  AlessthanB Exception e = new AlessthanB Exceptions;
  throw e;
 }
 Catch (AlessthanB Exception e)
{
 System.out.println (e.getMessage ());
 }
}
 else
 {
  int c = a - b;
  System.out.println ("The result is :"+c);
 }
}
}
```

o/p: Enter the value of a

10

Enter the value of b

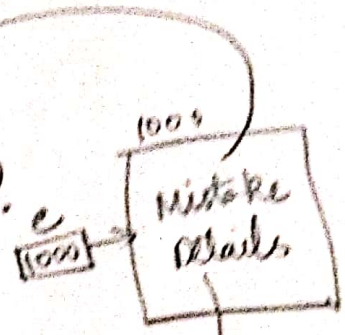
20

Subtraction is not possible

Scanned with CamScanner

Closer Look at ~~Exception~~ Throwable

```
class Exception
{
 Public string getMessage()
  {
    ===   } why the cause of exception?
  }
  Public string PrintstackTrace()
   {
     ===   } where the exception occur
   }
}
```



exception <{ }>
if exception occured it will be collected by exception itself

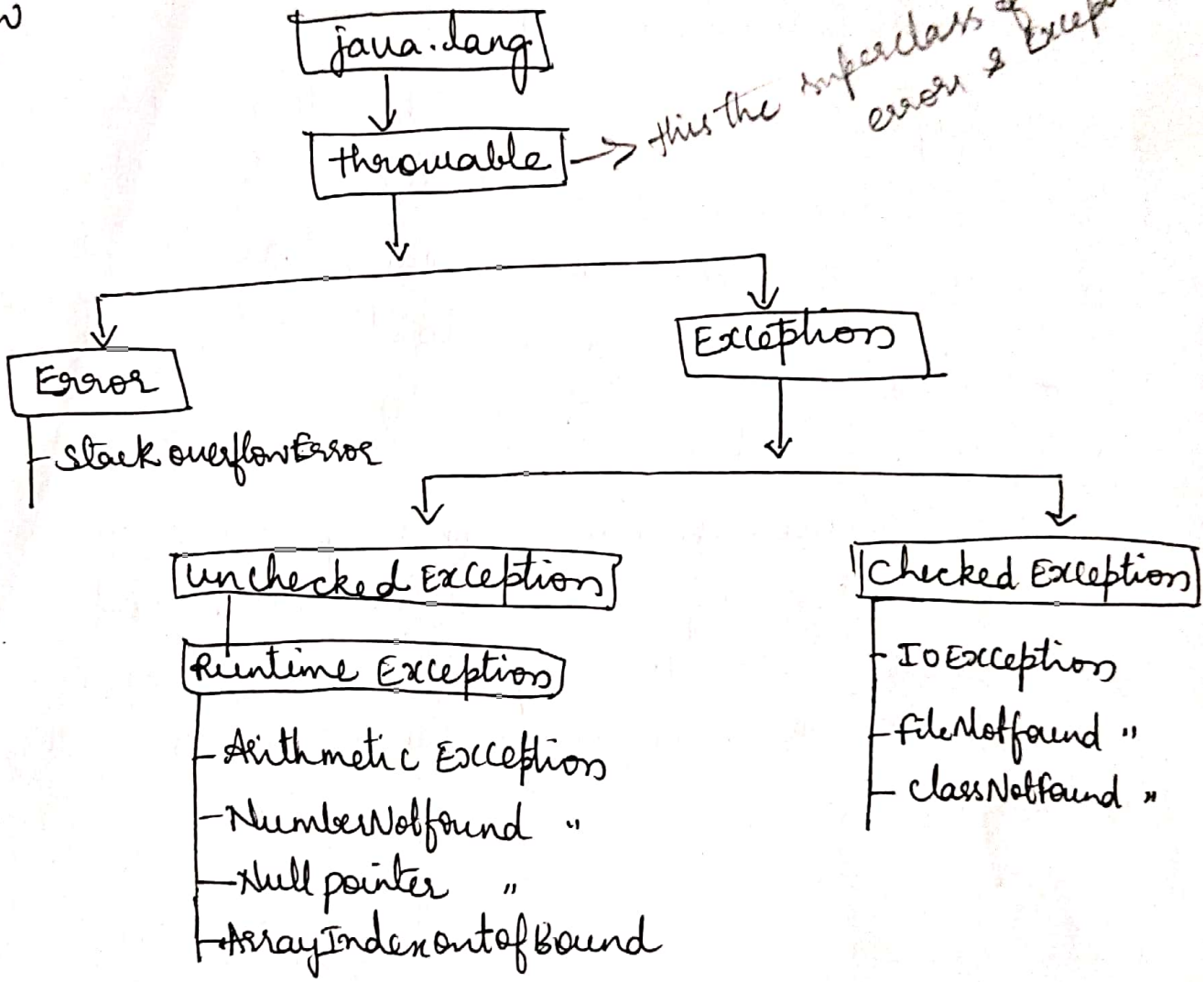
Eg:
```
Public class samplApp
 {
  Public static void main(String[] args)
  {
   System.out.println("Entering main()");
  try
  {
    int a=10;
    int b=0;
    int c=a/b;
    System.out.println(c); //error.
  }
   catch(Exception e)
   {
    System.out.println(e.getmessage());
    e.printstackTrace();
   }
   System.out.println("Exiting main()");
} }
```

o/p: Entering main()
div by zero
Exiting main()
java.lang.ArithmeticException: /by zero
at samplApp.main(samplApp.java:11)

Scanned with CamScanner

The Exception Hierarchy

All exception classes are derived from a class called java.lang.Exception, but exception is a subclass of the Throwable class. So, throwable class is the superclass of all errors and exceptions in the java language is shown below



Checked - checked exception is such type of exception which allows the user to place the code compulsory within try catch block.

Eg: IOException, SQLException

Scanned with CamScanner

Unchecked exception - Unchecked exceptions are such exceptions which allows the user to write the code without try catch.

Any exception object can be thrown if and only if it has the properties of throwable.

try blocks can be nested

→ One try block can be nested within another
→ An exception generated within the inner try block that is not caught by a catch associated with that try is propagated to the outer try block.

Why use nested try block

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

Syntax:

```
try
{
  try
  {
  }
  catch()
  {
  }
}
catch()
{
}
```

Scanned with CamScanner

Eg:

	0	1	2	3
num	8	16	24	32

	0	1	2
den	4	0	4

8/4 = 2

16/0 = " "

24/4 = 6

32 → ArrayIndexOutOfBoundException

```
Public class NestedApp
{
  Public static void main (String[] args)
  {
    int num[ ] = { 8, 16, 24, 32};
    int den[ ] = {4, 0, 4 }
    try  //outer try
    {
      for (int i=0 ; i< num.length ; i++)
      {
        try  //inner try
        {
          System.out.println ("num[i]+" / "+ den[i] + " = "
                                + num[i] / den[i]);
        } //Exit from inner try
        Catch (ArithmeticException e)
        {
          System.out.println (e.getMessage ());
        }
      }
    } //Exit from outer try
    Catch (ArrayOutOfBoundException e)
    {
      System.out.println(e);
    }
    System.out.println ("Rest of the code");
  }
}
```

Scanned with CamScanner

o/p 8/4 = 2
/ by zero
24/4 = 6
java.lang.ArrayoutofBound Exception : 3
Rest of the code.

Built in Exceptions

→ checked Exception
→ unchecked Exception

Checked Exception

→ it allows the user to place the code compulsorily within try catch block

→ A checkedException is an exception that occurs at the compile time, these are also called as compile time exceptions.
→ These exceptions cannot be ignored at the time of compilation, the programmer should handle these exceptions.
Eg: IO Exceptions, SQL Exceptions

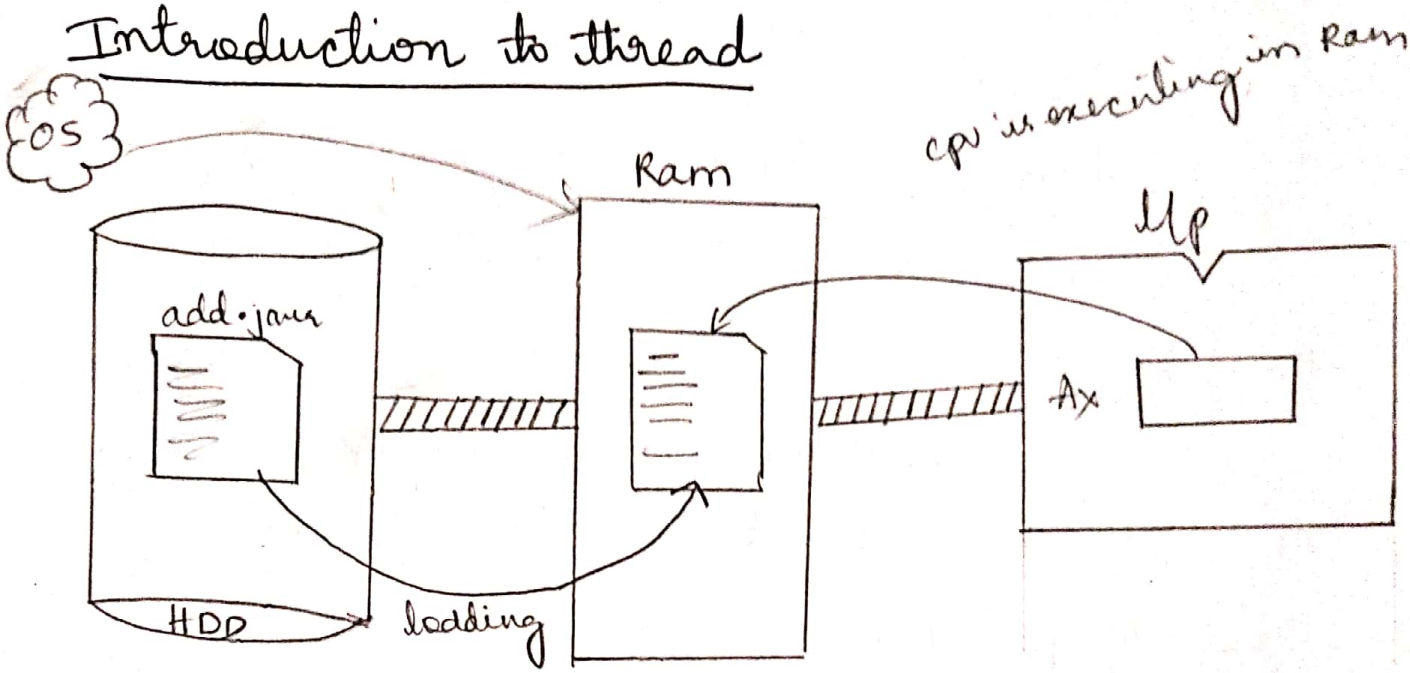
Unchecked Exceptions

→ The unchecked exception is an exception that occurs at the time of execution. These are also called as runtime exceptions.
→ These include programming bugs, such as logic errors or improper use of API.
→ Runtime exceptions are ignored at the time of compilation

Scanned with CamScanner

Multithreaded programming

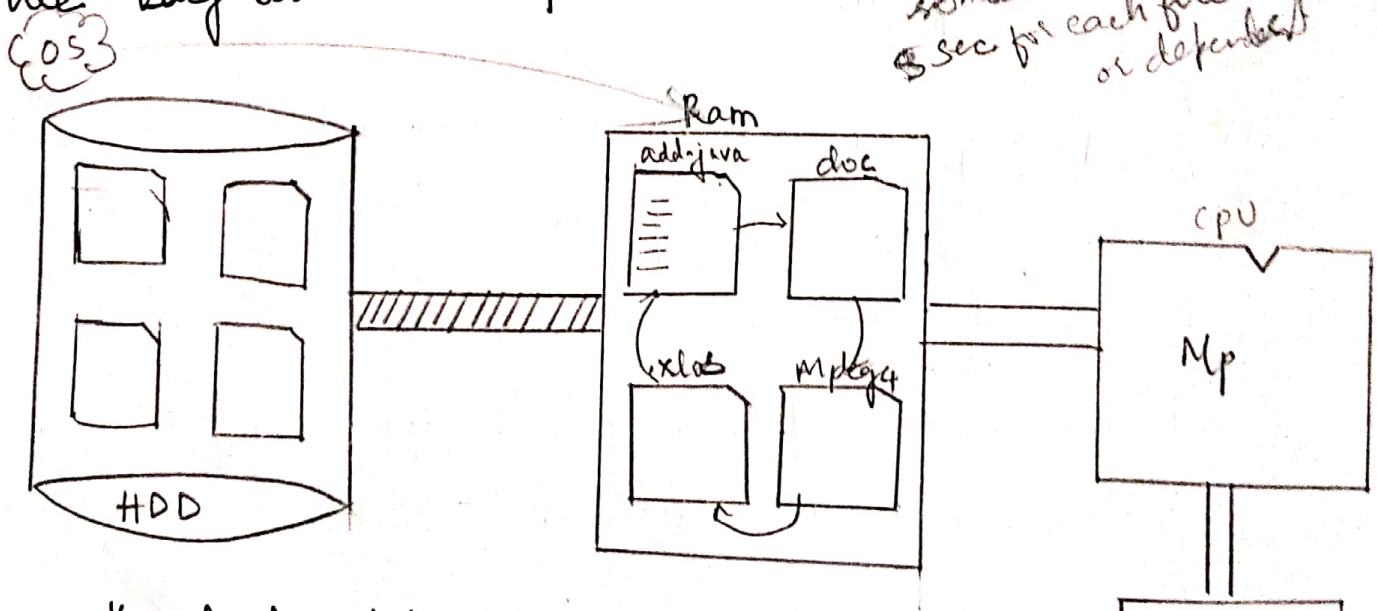
Introduction to thread



How many pgm is there & how many u r executing ?

"Process" → Single process and Single tasking O/S

if any program is under execution then we say it has a process



"ContextSwitching"

multiprocess and multitasking O/S

Scanned with CamScanner

it is the operating system which would allocate specific amount of time for every process on the ram.

if the o/s loads single program on the ram then it is refered as single tasking o/s.

if the o/s allocates multiple pgms on the ram then it is refered as multitasking o/s.

Note:

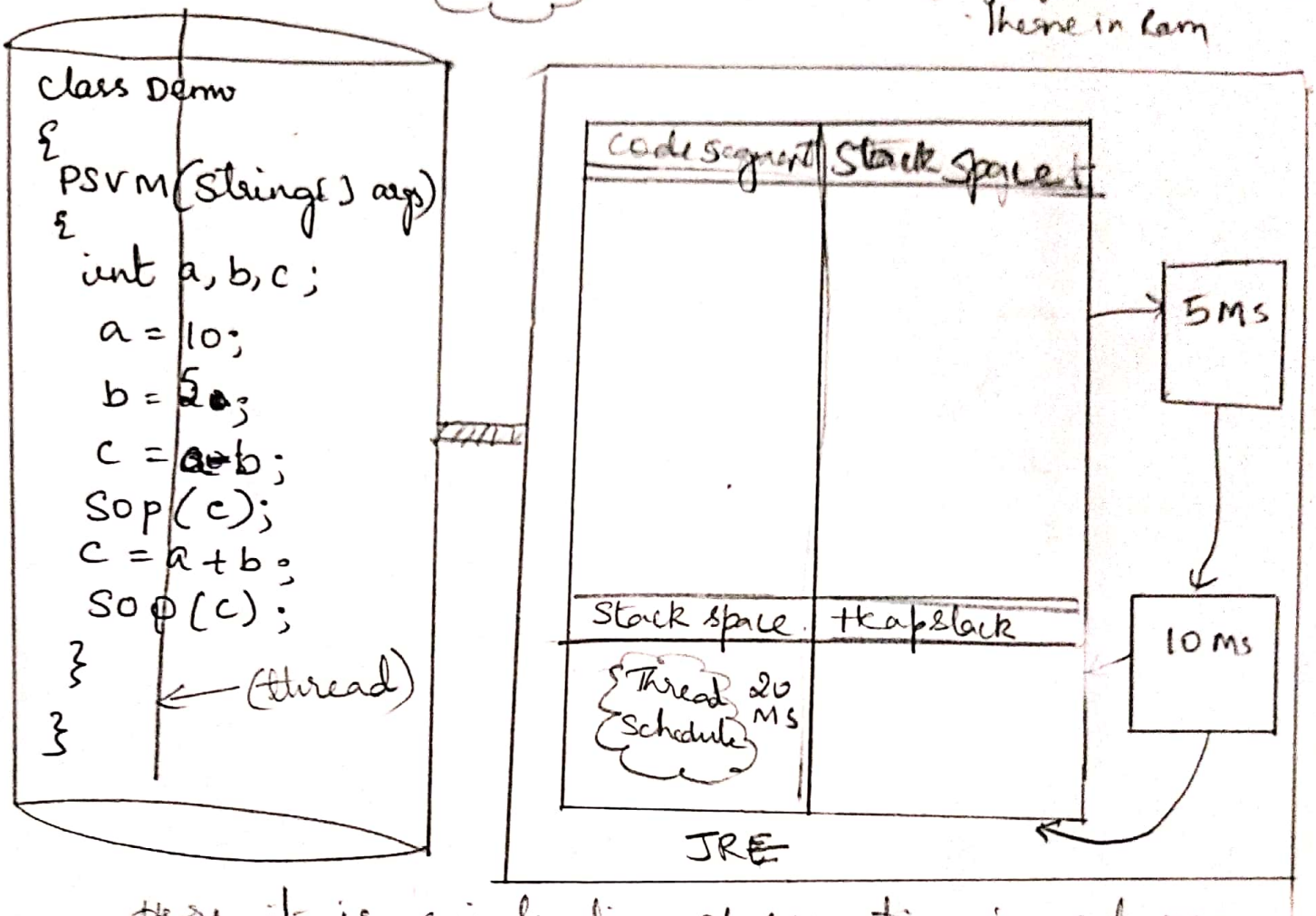
Operating system would specify the time for every program for its execution.

Java program would be executed under the control of the program called thread schedular.

Thread is refered as a line of execution which is shown below

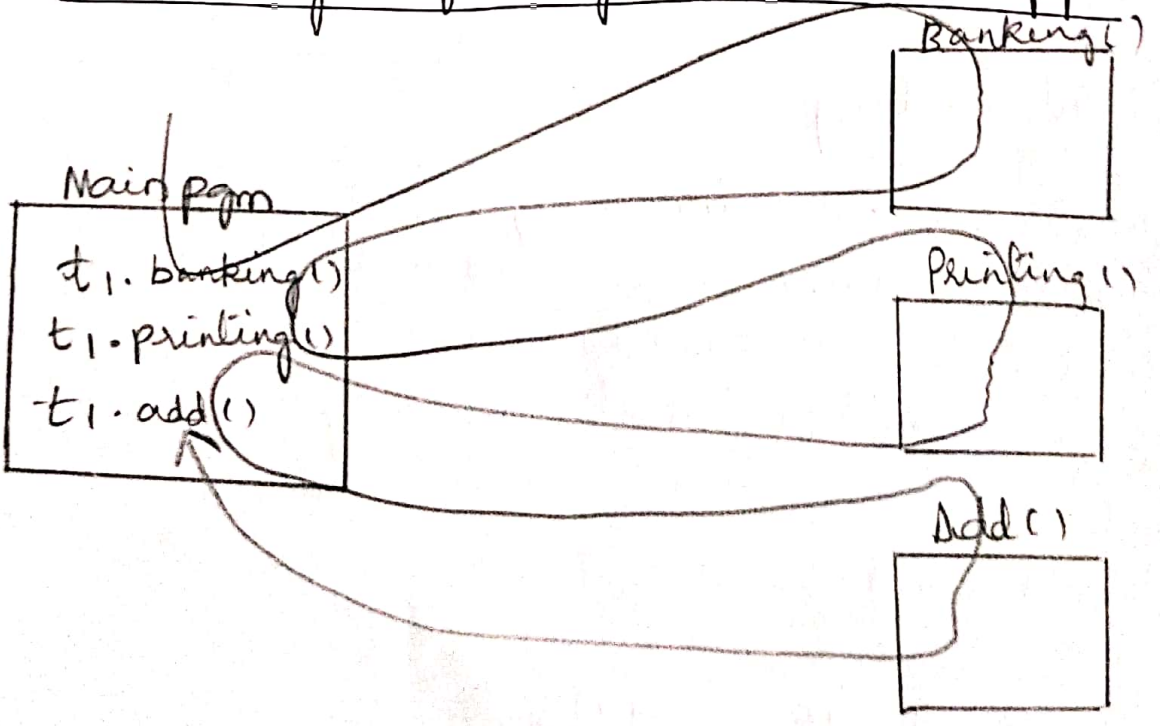
Scanned with CamScanner

OS is MTOS so n no of pgms will be there in Ram

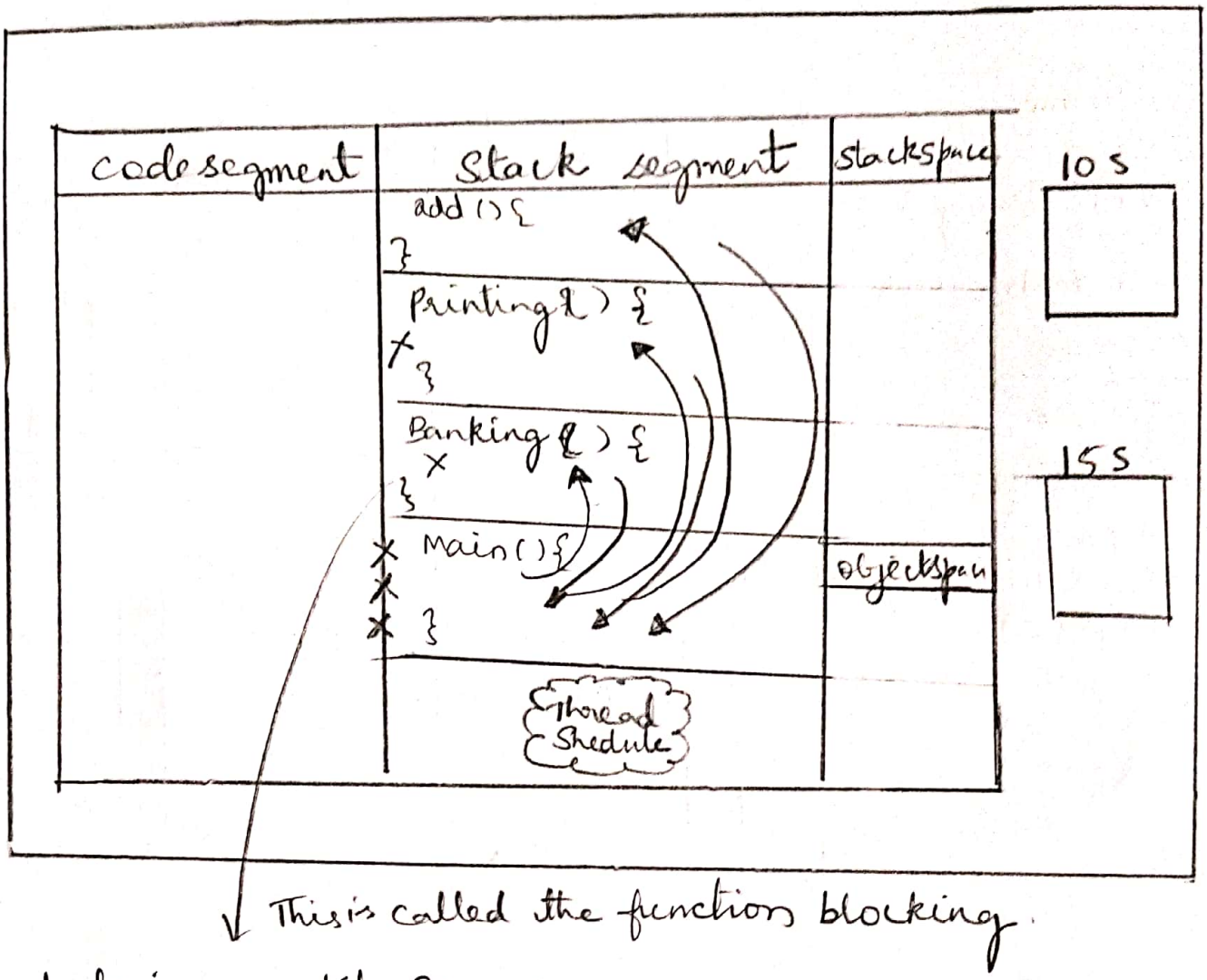


Here it is single line of execution i.e only one thread is executed

Disadvantage of single threaded App.



Scanned with CamScanner




```
import java.util.Scanner;
Class Task
{
 Public void banking()
 {
  Scanner scan = new scanner (System.in);
  System.out.println("Enter the amount");
  int amt = Scan.nextInt();
  System.out.println("collect ur money");
}
  Public void printing()
  {
    for(int i=1 ; i<=5 ; i++)
    {
      System.out.println("Bit");
```

Scanned with CamScanner

```
    try
    {
      thread sleep (3000)
    }
    Catch (Exception e)
     {
      e.printStackTrace();
     }
  }
}
 Public void add()
  {
     int a=10;
     int b=20;
      int c=a+b;
     System.out.println("The sum is :"+c);
  }
}
 Public class ThreadApp
{
 Public static void main(String[] args)
 {
   Task t1 = new Task();
    t1.banking();
    t1.printing();
     t1.add();
   }
 }
```

Scanned with CamScanner

O/P: Enter the amount
1000
Collect ur money
Bit
Bit
Bit
Bit
Bit
The sum is : 30

if take 1 sec to print once

In the above program there are 3 independent task, since the tasks are independent and if the pgm is created in a single threaded environment then the efficiency of application will come down, to resolve this problem we use multithread environment.

How to create multiple threads

it can be done in 2 ways.

1. Extending the thread class
2. Implementing runnable interface

Extending the thread class

→ while creating a thread all the independent task should run with a separate thread

→ The logic of a thread should be placed within run method.

→ A thread is activated by making a call to start() method.

→ In multithreaded environment the efficiency of an application would be increased.

Scanned with CamScanner

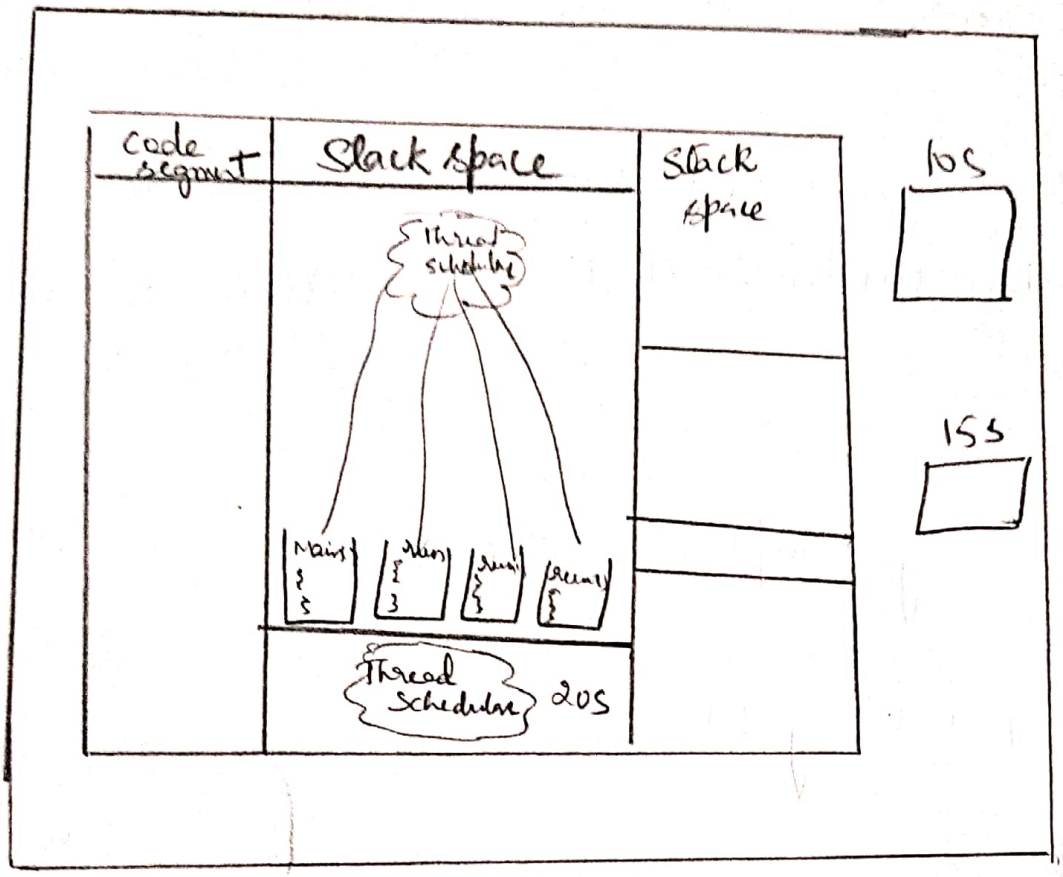
```
import java.util.Scanner;
class Banking extends Thread
{
  Public void run()
  {
    Scanner scan = new Scanner (System.in);
    System.out.println ("enter the amount ");
    int amt = Scan.nextInt();
    System.out.println ("collect ur " + amt)
  }
}
class printing extends Thread
{
  Public void run()
  {
    for (int i=1; i<=5; i++)
    System.out.println ("Bit");
    try
    {
      Thread.sleep(3000);
    }
    catch (Exception e)
    {
      e.printstackTrace();
    }
  }
}
```

Scanned with CamScanner

```
Class Add extends Thread
{
  Public void run()
  {
    int a=10;
    int b=20;
    int c=a+b;
    System.out.println("The sum is :"+c);
  }
}
Public class MultiThreadApp
{
  Banking b1=new Banking();
  Add a=new Add();
  Printing pr=new Printing();
   b1.start();
   a.start();
   Pr.start();
  }
}
```

O/P: BIT
The sum is: 30
Enter the amount
BIT
400 BIT
0
collect ur 4000
BIT
BIT

Scanned with CamScanner



2. Implementing runnable interface

In case of runnable interface we create a thread in constructor and also invoke start() method within a constructor

```
import java.util.Scanner;
class Banking implements Runnable
{
  Public Banking()  → constructor
  {
   Thread t1 = new Thread(this);  → we are creating thread for particular class i.e Banking
    t1.start() → to start thread/
  }                                activate thread
```

Scanned with CamScanner

```
Public void run ()
{
Scanner scan = new scanner (system.in);
System.out.println ("Enter the amount");
int amt= scan.nextInt ();
System.out.println ("collect ur amount " +amt);
 }
}
Class printing implements Runnable
{
  Public printing ()
 {
  Thread t2 = new Thread (this);
  t2.start ();
 }
 Public void run ()
 {
  for (int i=1 ; i<=5 ; i++)
  {
   System.out.println ("Bit");
   try
    {
     Thread.sleep(3000);
    }
     Catch (Exception e)
    { e.printstackTrace ();
} } } }
```

Scanned with CamScanner

```
class Addition implements Runnable
{
 Public Addition ()
 {
 Thread t3 = new Thread (this);
  t3.start ();
 }
 Public void run ()
  {
    int a = 10;
    int b = 20;
    int c = a + b;
    System.out.println("The sum is : "+c);
  }
}
Public class InterfaceApp
{
 Public static void main(String[ ] args)
 {
    Banking b1 = new Banking ();
    Addition add = new Addition ();
    Printing pr = new printing ();
  }
}
```

O/P

```
The sum is : 30
  BIT
Enter the amount
BIT
4000
Collect us 4000
 BIT
```

```
BIT
BIT
```

Scanned with CamScanner

Commonly used methods of Thread class.

Methods	use
Public void run()	→ is used to perform action for a thread
Public void start()	→ Starts the execution of the thread
Public void join()	→ waits for a thread to die.
Public void sleep()	→ Can be used to pause the execution of current thread for specified
Public String getName()	→ returns the name of the thread
Public void setName(String name)	→ changes the name of the thread
Public int getID()	→ rethorns the ID of the thread
Public void suspend()	→ is used to suspend the thread
Public void stop()	→ is used to stop the thread
Public void resume()	→ is used to resume the suspended thread.

Scanned with CamScanner

Determining when a Thread Ends

i) isAlive() → helps to determine whether a thread has finished

Syntax: final boolean isAlive()

returns true if the thread upon which it is called is still running otherwise false.

```
Class FirstThread implements Runnable
{
  Thread t;
  Public FirstThread()
  {
     t = new Thread(this);
     t.start();
     System.out.println(t.isAlive());
  }
  Public void run()
  {
    for(int i=1; i<=5; i++)
    {
      System.out.println(i);
      try
      {
        Thread.sleep(3000);
      }
      Catch(InterruptedException e)
      {
      }
    }
  }
}
```

Scanned with CamScanner

```
Public class ThreadEndApp
{
 Public static void main(String[] args)
 {
  FirstThread fs = new FirstThread ();
 }
}
```

O/P

```
1
true
2
3
4
5
```

isAlive method is used to check whether the thread is alive or not. if the thread is alive it prints the boolean value as true.

Thread priorities

By default JVM would assign Priority as 5 to the main thread. As a Programmer I can change the priority of a thread using set priority method.

```
Public class Threadpriority
{
 Public static void main(string[] args)
 {
```

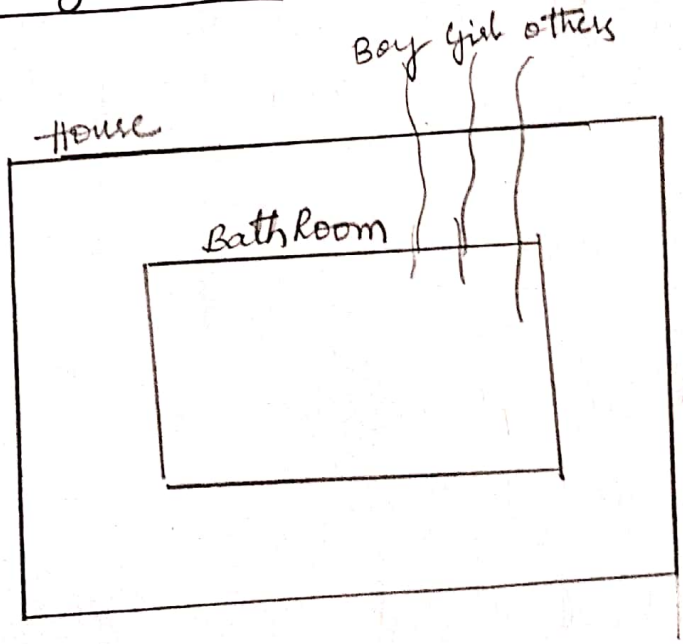
Scanned with CamScanner

```
Thread t1 = Thread.currentThread();
System.out.println("Before setting the priority");
System.out.println(t1);
System.out.println("After setting the priority");
t1.setpriority(7);
t1.setName("BIT");
System.out.println(t1);
}
}
```

O/P:

```
Before setting the priority
Thread[main,5,main]
After setting the priority
Thread[BIT,7,main]
```

Synchronization



1) Enter the bathroom
2) close the door
3) taking bath
4) dressing up

Scanned with CamScanner

```
Class House implements Runnable
{
 Public House()
 {
   Thread t1 = new Thread(this);
   Thread t2 = new Thread(this);
   Thread t3 = new Thread(this);

    t1.setName("Boy");
    t2.setName("Girl");
    t3.setName("Other");

    t1.start();
    t2.start();
    t3.start();
  }
Synchronized public void run()
{
 System.out.println(Thread.currentThread().getName()
                    +" has entered the bathroom");
 try
 {
  Thread.sleep(3000);
 }
 catch (InterruptedException e)
  {
   e.printStackTrace();
 }
 System.out.println(Thread.currentThread().getName()
                    +" has closed the door");
```

Scanned with CamScanner

```
try
{
 Thread.sleep(3000);
}
catch (InterruptedException e)
{
 e.printStackTrace();
}
System.out.println("Thread.currentThread.getName()+"
                   is taking the bath")
try
{
 Thread.sleep(3000);
}
 catch (InterruptedException e)
{
 e.printStackTrace();
}
System.out.println(Thread.currentThread.getName()+
                   "is dressing up!!!");
try
{
 Thread.sleep(3000);
}
catch(InterruptedException e)
{
 e.printStackTrace();
}
}
}
Public class BathRoomApp
{
 Public static void main(String[] args)
 {
```

Scanned with CamScanner

```
        House h = new House();
    }
}
```

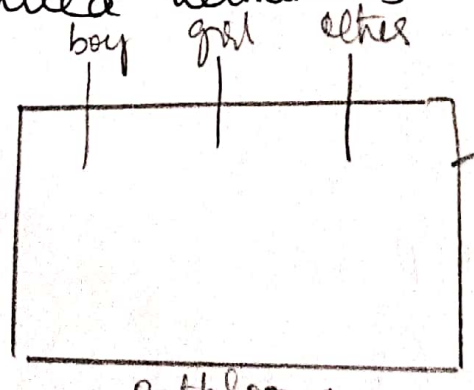
O/P: Boy has entered the Bathroom
" has closed the Bathroom
" is taking the bath
" is dressing up.
others
"
"
"
girl
"
"
"

In multithreaded environment if a common resource has to be used then all the threads will access the same resource.

if we want only one thread to access the resource at any given point of time, then such resource (logic) should be kept in synchronized block.

join()

→ wait for a thread to finish
→ This method waits until the thread on which it is called terminates



t1.start()
t1.join()
t2.start()
t1.join()
t3.start()

Scanned with CamScanner

```
Class House Implements Runnable
{
 Public House()
 {
  Thread t1 = new Thread(this);
  Thread t2 = new Thread(this);
  Thread t3 = new Thread(this);

    t1.setName("Boy");
    t2.setName("Girl");
    t3.setName("Others");

   try
   {
    t1.start();
    t1.join();
    t2.start();
    t2.join();
    t3.start();
  }
  catch(Exception e)
  {
   e.printStackTrace();
  }
}
 Public void run()
 {
  System.out.println(Thread.currentThread().getName()
                     + "has entered the bathroom");
```

Scanned with CamScanner

```
try
{
 Thread.sleep(3000);
}
 catch (InterruptedException e)
 {
   e.printStackTrace();
 }
System.out.println (Thread.currentThread().getName()
                        +" has closed the door");
try
{
  Thread.sleep(3000);
}
 catch (InterruptedException e)
 {
   e.printStackTrace();
 }
System.out.println (Thread.currentThread().getName()
                        + "is taking the bath");
 try
 {
  Thread.sleep(3000);
 }
 catch (InterruptedException e)
 {
  e.printStackTrace();
 }
 System.out.println (Thread.currentThread().getName()
                        + "is dressing up!!!");
```

Scanned with CamScanner

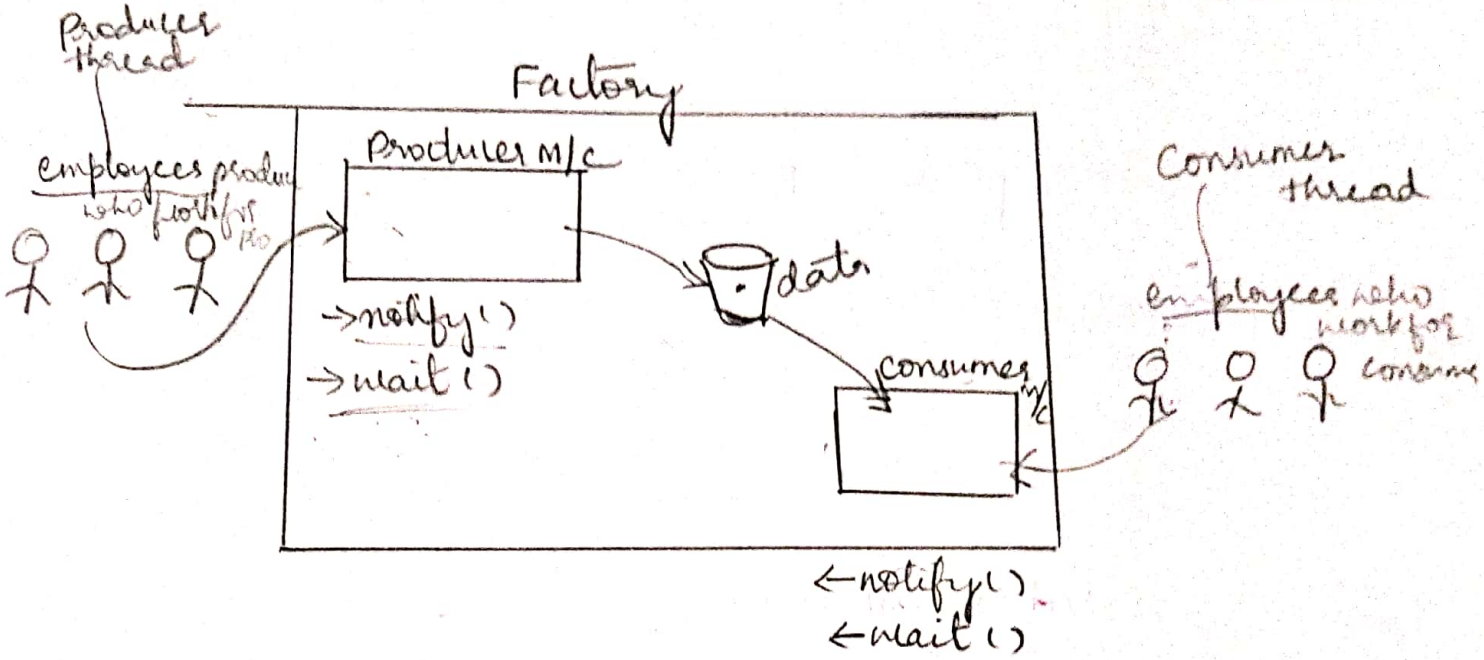
```
try
{
Thread.sleep(3000);
}
catch (InterruptedException e)
{
  e.printStackTrace();
}
}
}
Public Class Bathroomjoin
{
  Public static void main(String[] args)
  {
    House h = new House();
  }
}
```

O/P:

```
Boy has entered the bathroom
Boy has closed the door
Boy is taking Baths
Boy is dressing up !!!
Girl has entered the bathroom
Girl has closed the door
Girl is taking bath
Girl is dressing up !!!
others has entered the bathroom
others has closed the door
others is taking baths
others is dressing up !!!
```

Scanned with CamScanner

Producer consumer problem




```
Class factory
{
 int item=0;
Synchronized public void producerMachine()
 {
  try
  {
   item = item + 1;
   System.out.println("item" + item + "is producing");
  Thread.sleep(4000);
   System.out.println("item" + item + "is produced");
   notify();
   wait();
  }
  catch(Exception e)
  {
   e.printstacktrace();
  }
}
```

Scanned with CamScanner

```
Synchronized public void consumerMachine()
{
 try
 {
 System.out.println("item" +item + "is consuming");
 Thread.sleep(2000);
 System.out.println("item" +item + "is consumed");
 notify();
 if (item <5)
 {
  wait()
 }
 }catch(Exception e)
  {
   e.printStackTrace();
  }
 }
}
class producer extends Thread
{
  Factory x;
  public producer(factory x)
  {
    this.x= x;
  }
  Public void run()
  {
    for(int i=1;i<=5;i++)
    {
      x.producerMachine();
  } } }
```

Scanned with CamScanner

```
class consumer extends Threads
{
  factory x ;
  Public Consumer(Factory x)
  {
    this .x = x;
  }
  Public void run()
  {
    for (int i=1 ; i<=5 ; i++)
    {
      x.consumerMachine ();
    }
  }
}
Public class Labs
{
  Public static void main(String[ ] args)
  {
      factory f = new factory();
      Producer p = new Producer();
      Consumer c = new Consumer();
      P. start ();
      C. start ();
  }
}
```

Scanned with CamScanner

O/P:

item 1 is Producing
item 1 is produced
item 1 is consuming
item 1 is consumed

Scanned with CamScanner

Supending resuming and stopping Threads

```
Public class ThreadApp1 extends Thread
{
 Public void run()
 {
  for(int i=1; i<=5 ; i++)
  {
   try
   {
    Thread . sleep(3000);
   System.out.println(Thread.currentThread().
                      getName() + "t" + i);
  }
  catch(InterruptedException e)
  {
   e.printStackTrace();
  }
}
}
Public class ThreadResume
{
Public static void main(String[] args)
{
 ThreadApp1 t1 = new ThreadApp1();
 ThreadApp1 t2 = new ThreadApp2();
  t1. setName("first");
  t2. setName("second");
  t1. start();
  t2. start();
```

Scanned with CamScanner

```
try
{
 Thread.sleep(1000);
  t1.suspend();
 System.out.println("Thread1 is suspended");
 Thread.sleep(500);
  t1.resume();
 System.out.println("Thread1 is resumed");
 Thread.sleep(1000);
  t2.suspend();
 System.out.println("Thread2 is suspended");
 Thread.sleep(500);
  t2.resume();
  System.out.println("Thread2 is resumed");
 }
 catch(Exception e)
  {System.out.println(e.getMessage());
   }
 }
}
```

o/p: Thread1 is suspended
Thread2 is resumed
Thread2 is suspended
Thread2 is resumed

First 1	first 3	first 5
Second 1	Second 3	Second 5
First 2	first 4	
Second 2	Second 4	

Scanned with CamScanner

Note:

NotifyAll() → This method is used to give a signal to all the threads in the applications. Highest priority threads in the application will be activated.

Stop() This method is used to stop the working of particular thread. Stop func is not Preffered.

Scanned with CamScanner