OBJECT ORIENTED PROGRAMMING — JAVA

*. CLASES :

→ It defines a new data type.

→ once defined, this new type can be used to create objects of that type.

→ A class is template for an object, and an object is an instance of class.

*. The General form of Class :

```
class classname {
    type instancevariable1 ;
    type instancevariable2 ;
      :
type methodname1 ( parameter-list )
{
    // body of method
}
type methodname2 (parameter - list)
{
    // body of method
}
  :
}
```

→ Collectively, the methods and variables defined within a class are called members of class.

Example : A Simple Class

```
class Box
{
    double width;
    double height;
    double depth;
}
```

```
Box obj = new Box( )        // Create a Box object called o
```

→ To access these variables, you will use the dot (.) operator.

→ The dot operator links the name of the object with the name of instance variable

Example : obj.width = 100;

*. When you assign one object reference variable to another object reference variable, you are not creating a copy of the objects, you are only making a copy of reference.

* - Introducing Methods :

```
typename (parameter)
{
  // body
}
```

typename - type returned by
parameter - variables passing
// body - code

- cannot write method inside main method.
- If method doesnot return a value, it is void.
- parameter list : int a, float b, string c, ...
- If any value is returned, use 'return' & typename - data type

example :

```
void
double volume ( int height, int length, int breadth)
{
    double v = height * length * breadth;
    System.out.println (v);   → If there is no return
    return v;                 → return double value
}
obj.volume ( );               → If no return
obj.volume ( height, length, breadth)   → pass to method & return
```

* . Constructor :

```
class_name( )
{
   // code
}
```

1. Default Constructor :

- If there is no Constructor in class, compiler automatically Creates a default constructor.

```
class volume
{
}
```

⟹

```
class volume()
{
   volume()
   { }
}
```

- The default Constructor is used to provide the default value to the object like 0, null, etc... depending on the type.

2. Parametrized Constructor :

- A constructor which has a specified no. of parameters is called "Parametrized constructor."

- This constructor is used to provide different values to distinct objects. However, you can provide the same values also.

Example :

```
Public class Student {

        int id;
        String name;

        Student (int i, String n)
        {
           id = i;                    → Constructor
           name = n;
        }

    public static void main (String args[])
    {
        Student obj1 = new Student (111, "GVP");
        Student obj2 = new Student (222, "GSC");

        obj1.display ();
        obj2.display ();

    }

    void display { s.print (id + "  " + name); {

}
```

*. The "this" keyword :

→ "this" can be used inside any method to refer to the current object.

→ "this" is always a reference to the object on which the method was invoked.

example :

```
Box (double w, double h, double d)
{
    this.width = w ;
    this.height = h ;
    this.depth = d ;
}
```

(OR) // Use this to avoid name-space collisions.

```
Box (double w, double h, double d)
{
    this.width = width;
    this.height = height;
    this.depth = depth;
}
```

*. If the instance variable & local variable names are same, you assigned local variable to instance variable & tried to print it. Both becomes local variable only. So, the output will be 0.

To avoid this, we use instance variable like

this.instanceVariable = localvariable.

if instanceVariable = x;
and localvariable = x;

```
Class Test
{
    int i;  → instance variable
    void setValues (int i)
    {                  ↓ local variable
        this.i = i;
    }
    void show()
    {
        System.out.println(i);
    }
}
```

```
Class Main
{
    P S V m ( )
    {
        Test t = new Test();
        t.setValues(10);
        t.show();
    }
}
// output = 10
```

---

*. 6 Uses of this Keyword :

IMP

1. this keyword can be used to - refer current class instance variable.
2. to invoke current class method (implicitly).
3. to invoke current class constructor.
4. to pass an argument in method call.
5. to pass an argument in constructor call.
6. to return the class instance from method.

---

*. GARBAGE COLLECTION :

→ Java handles deallocation for you automatically, this is called garbage collection (instead of delete operator).

→ When there is no reference to object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed.

* Package – collection of classes

* Class – Keyword used for developing user defined data type

* class Name – Valid java valid variable name.

* Data member – either instance or static · based on name of class

* User-defined → created by user.

* There are three kinds of variables :

1. Local variable – variable within block { .... }.
2. Instance variable – defined in class outside method
3. Static/class variable : – without object you can access class variable as there are in class.

* Static variables can be called by Classname . variable-name

* INHERITENCE :

→ A class that is inherited is called "super class."

→ A class that does the inheriting is "subclass."

→ Super class – parent class, base class

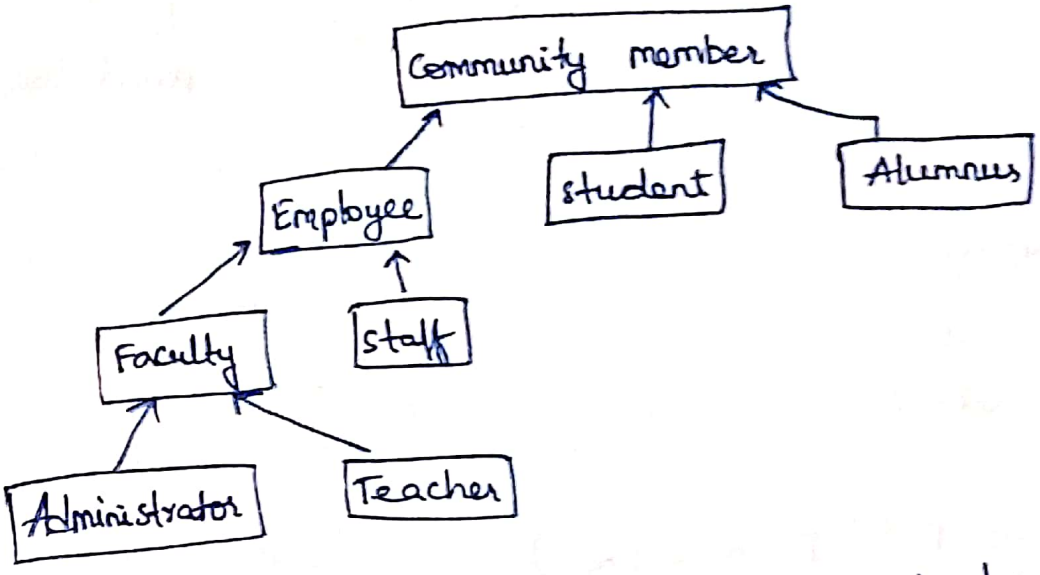
Sub class – derived class, child class

"extends" is defined in inheritence

Inheritence Examples

Super class	Subclass
Student	B-Tech Student, M-Tech Student, Graduate Student
Shape	rectangle, triangle, circle, cube, sphere
Employee	Faculty, Staff, Manager, officer

University Community Member Hierarchy :



→ private variables of super class cannot be accessed using inheritance in sub class.

→ A super class can reference sub class variables also.

→ Assign super class object to sub class object (possible)

* SUPER Keyword :

→ A sub class can call constructor defined by its superclass by use of following form of "super"

super (arg-list)

→ super() must always be first statement executed inside a subclass constructor.

INHERITENCE :

Dog ↓ child class — IS-A relation — Animal ↓ parent class

Example :

```
class Animal
{
    void eat()
    {
        S.O.P("I am eating");
    }
}

class Dog extends Animal        → Parent class
{
    P.S.V.M (String args[])

    Dog obj = new Dog();
    obj.eat();
}
```

Above example code eat() is used in Dog class

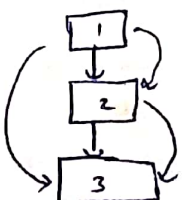
- Inheritence is also called IS-A relation.
- Inheritence main Advantage - Code reusability.
- We can achieve "Polymorphism" through Inheritence

*. TYPES OF INHERITENCE :

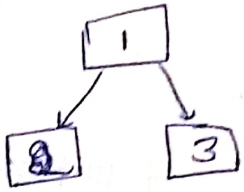
→ 5 types of

1. Single Inheritence —

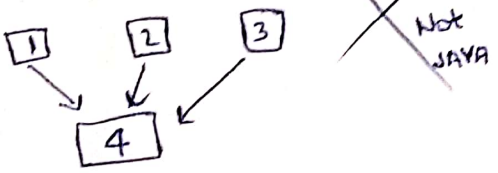
2. Multilevel Inheritence —



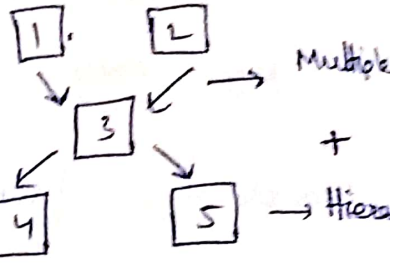
3. Hierarical Inheritence :



4. Multiple Inheritence :



5. Hybrid Inheritence



Any mix of two Inheritence is Hybrid

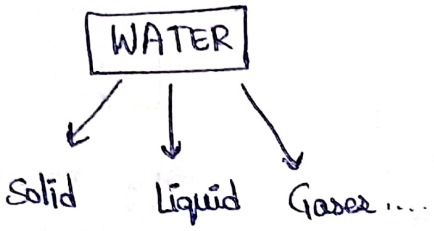
- Java supports — Single Inheritence
  Multiple Inheritence
  Hierarical Inheritence

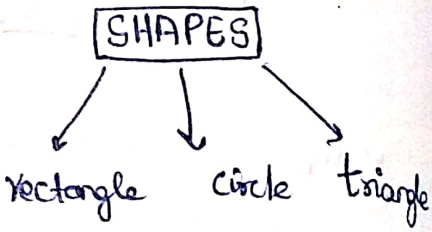
- Constructors of one class donot inherit in other class.
- private methods of one class donot inherit in other class.
- Not all properties donot inherit from one class to other.
- Every class has a super or say class (parent class) i.e. object class (but, object class doesn't have any parent class.)
- object is the parent class of all the classes in JAVA.
- There can be only one super class, not more than that because java doesn't support multiple inheritance.

④ POLYMORPHISM :

poly - many morphism - forms

Example :





→ Types of Polymorphism :

1. COMPILE-TIME POLYMORPHISM : (static also called)

→ Achieved by method overloading.

→ Handles by compiler.

2. RON-TIME POLYMORPHISM : (dynamic also called)

→ Achieved by method overriding.

→ Handled by JVM.

*. METHOD OVERLOADING :	*. METHOD OVERRIDING :
1. Same name of methods.	1. Same name of methods.
2. These methods in same Class.	2. These methods in different Cl
3. No. of Arguments are different Type of Arguments are different order of Arguements are different	3. No. of Arguments are same Type of Arguments are same order of Arguments are same
4. Different Arguments.	4. Same Arguments.
	5. INHERITENCE ✓ compulsory

I. METHOD OVERLOADING :

Example :

```
Class Test
{
    void show()
    {
        S.O.P ("1");
    }
    void show()
    {
        S.O.P("2");
    }
    public static void main (String args)
    {
        Test t = new Test();
        t.show();
    }
}
```

This program gives error called ambeguitte compiler confuses

*. METHOD OVER RIDING :

```
Class Test
{
    void show()
    {
        S.O.P(1);
    }
}
Class XYZ extends Test
{
    void show()
    {
        S.O.P("2");
    }
    PSVM()
    {
```

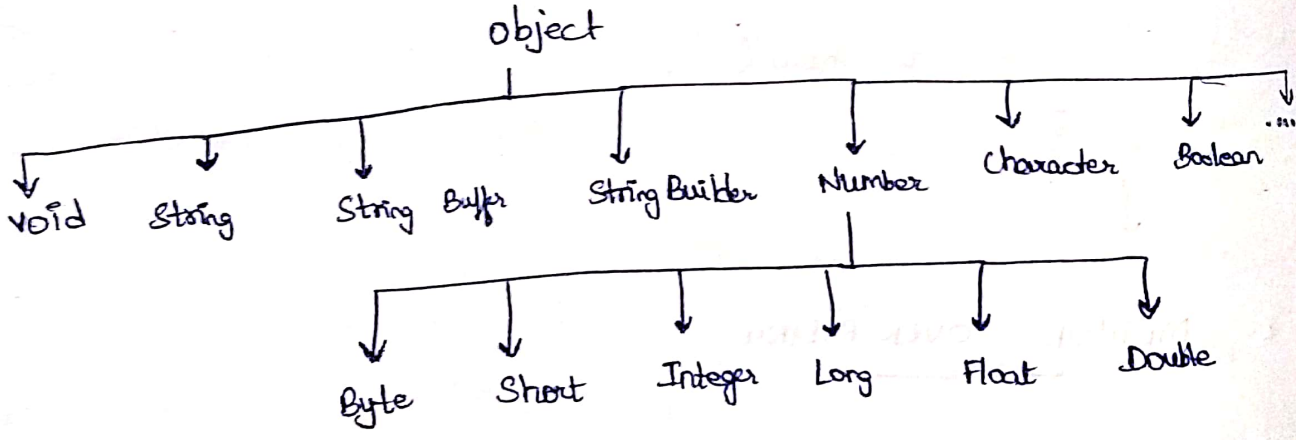
until we write this, it cannot be overding.

// here overiding

*. Method overiding allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its superclass or parent class.

*. The implementation in the subclass overides (replaces) the implementation in the super class by providing a method that has same name, same parameters, or signatures, and same return type as the method in the parent class.

CASE - I : Do overiding method must have same return type (or subtype) ?



Today, It is possible to have different return type for overriding method in child class, but child's return type should be sub-type of parent's return type. This phenomena is known as "Covariant return type".

CASE - II : Over riding & Access - Modifier.

*. The access modifier for an overriding method can allow more, but not less, access than the overridden method. For example,

A potential protected instance method in super-class can be made public, but not private, in the sub-class. Doing, so will generate compile-time error.

*. ABSTRACTION IN JAVA :

Example :

As in the car case, relevant parts like steering, gear, horn, accelerator, breaks etc are shown to driver because they are necessary for driving.

But the driver need not know the internal functioning of engine, gear, etc. Thus, showing relevant data to the user and hiding implementation or details from the user is "Abstraction."

→ 2-Two ways to achieve :

1. By using Abstract class
2. By Interfaces (100 %)

*. A method without body (no implementation) is known as abstract method.

e.g.:

Normally:

```
Void show()
{
  System.out.println("I am hidden");
}
```

⟹ To hide implementation:

```
Void show()
{
  // Hidden
}
```

⇓

*. This can be achieved by abstract key word.

So, it is written like this: abstract void show();

*. A method without body ⟹ Abstract method.

*. A method must always be declared in abstract class, or we can say that is if a class has an abstract method, it class should be declared abstract as well.

*.
```
abstract Class Test
{
    abstract void show();
    int number;
}
```

*. If the regular class extends an abstract class then the class must have to implement all the abstract methods of parent class or it has to be declared abstract as well.

Real - Life example :

```
abstract class Vehicle
{
    int no. of vechiles;
    abstract void start();
}
class Car extends Vehicle
{
    void start()
    {
        System.out.println("Starts with Key");
    }
}
class Scooter extends Vehicle
{
    void start()
    {
        System.out.println("Starts with Kick");
    }
}
```

*. Abstract methods in an abstract class are meant to be overridden in derived concrete class otherwise compiler time error will be thrown.

* . RULES :

① A method without body is Abstract method. This can be achieved by abstract keyword.

② If a class has abstract method then definately, that class must be declared abstract class. but if it is abstract class then there may not be case to declare abstract method compulsory. inside it.

③ If regular class extends an abstract class, then the class must have to implement all the abstract methods of abstract parent class or it has to be declared abstract as well.

④ Abstract methods in abstract class are meant to be overridden in derived concrete classes otherwise compiler throws compile-time error.

⑤ Abstract classes cannot be instantiated, means we can't create an object of Abstract class. but we can create reference.

*- INTERFACES :

→ Interfaces are similar to Abstract class but having all the methods of abstract type.

→ Interfaces are the blueprint of the class. It specify what a class must do and not how.

1. It is used to achieve Abstraction.
2. It supports Multiple Inheritance
3. It can be used to achieve loose coupling.

*. Loose Coupling is design goal that seaks to reduce the inter dependencies between components of a system with the goal of reducing the risk that changes in one component will require changes in any other Component.

Syntax :

```
interface   INTERFACE NAME
{
    // Abstract methods
    // public Abstract type

    // public static final fields
}
```

① public abstract methods } before 8th JAVA version

② public static final variables } before 8th JAVA version

③ default concrete methods } 8th version JAVA

④ Static methods (public) } 8th version JAVA

⑤ private methods } 9th JAVA version

INSIDE INTERFACE CLASS

*. SIMILARITIES BETWEEN ABSTRACT CLASS & INTERFACES

① Both can contain abstract methods.

② We cannot create an instance of abstract class and interfaces.

*. DIFFERENCES BETWEEN :

ABSTRACT CLASS	INTERFACES
1. Abstract class can have instance methods that implements a default	1. Methods of Java interface are implicitly abstract and cannot have implementations.
2. An abstract class may contain non-final variable	2. Public, static & final variables for Interfaces.
3. Any access-modifiers	3. PUBLIC
4. "Extends"	4. "implements"
5. extend another class	5. Can extend another inter-face

*. ENCAPSULATION :

→ Encapsulation in Java is mechanism of wrapping the data (variables) and code acting on data (methods) together as single unit.

→ Steps to achieve :

① Declare the variables of class as private

② Provide public setter and getter methods to modify and view the variables values.

private – this cannot be used beyon that class.
This concept is called Data hiding.

Example :

```
class Employee
{
    private int employee_id;
    public void setEmployeeId (int employee_id1)
    {
        employee_id = employee_id1;
    }
    public int getEmployeeId ()
    {
        return employee_id;
    }
}
```

ENCAPSULATION

*. SUPER Keyword :

→ "Super" Keyword is reference variable which is used to refer immediate parent class object.

→ But "this" Keyword is reference variable but used for only current class object.

Example :

```
Class A
{
    int a = 10;
}
Class B extends A
{
    int a = 20;
    void show (int a)
    {
        System.out.print (a);        S.O.P (this.a);
                                     S.O.P (super.a);
    }
}
Public Static void main (String args[])
{
    B obj1 = new B()
    obj1.show (30);
}
```

output = 30

output = 20

output = 10

*. Uses of Super Keyword :

(1) "Super" Keyword can be used to refer immediate parent class instance variable.

(2) "super" Keyword can be used to invoke immediately parent class method.

(3) Super() can be used to invoke immediate parent class constructor.

JAVA GENERICS

- When you have to store some data what matters you is Content not datatype. This is the case where we use "GENERICS".
- Generics in Java is a language feature in Java that allows you to use Generic types and methods.
- With Generics, you can define algorithms at once, independently of any specific data and then use it for any data type.
- The term Generics means Parameterized types.
- Parameterized types helps you to create classes, interfaces and methods in which the type of

data upon which they operate is specified as a parameter.

- Generic class or Generic Method – A class, interface, or method that operates on parameterized type.

Example :

```
class Gen <T>          → Simple Generic class
                       → Here T is type parameter that will replaced the by a real type when an object of type Gen is created.
{
    T ob;              → declare an object of type T.

    Gen (T o)          → constructor Gen reference to object of type T
    {
      ob = o;
    }

    T getob()          → return ob
    {
      return ob;
    }
    void showType()    → show type of T
    {
       S.O.P ("Type of T is" + ob.getClass().getName());
    }
}

class GenDemo
{
    P.S.V.M
    {
        // code →
    }
```

```
class GenDemo
{
    public static void main (String args[])
    {
        Gen <Integer> iOb;
        iOb = new Gen <Integer> (88);
        iOb.showtype();
        int v = iOb.getob()
        S.o.P (" Value is " + v);
        S.o.P();
        Gen <String> strOb = new Gen <String> ("TEST")
        strOb.showtype();
        String str = strOb.getob();
        System.o.P (" value is" + str);
    }
}
```

EXCEPTION HANDLING

*. Exception - which disturbs the normal flow of a prog.

An exception is an unwanted or unexpected event, which occurs during the execution of program i.e. at run-time, that disrupts the normal flow of program.

*. Exception Handling : Handle / search for alternative for an exception in runtime.

IMP
⊙ object class is the parent class of all the classes in java. ⟹

- Throwable class is the parent class of Exception class and error class in java.

- In most of the cases, Exceptions are occured by the programs. Errors are occured because of lack of System Resources ; not by our programs and thus programmer cannot do anything.

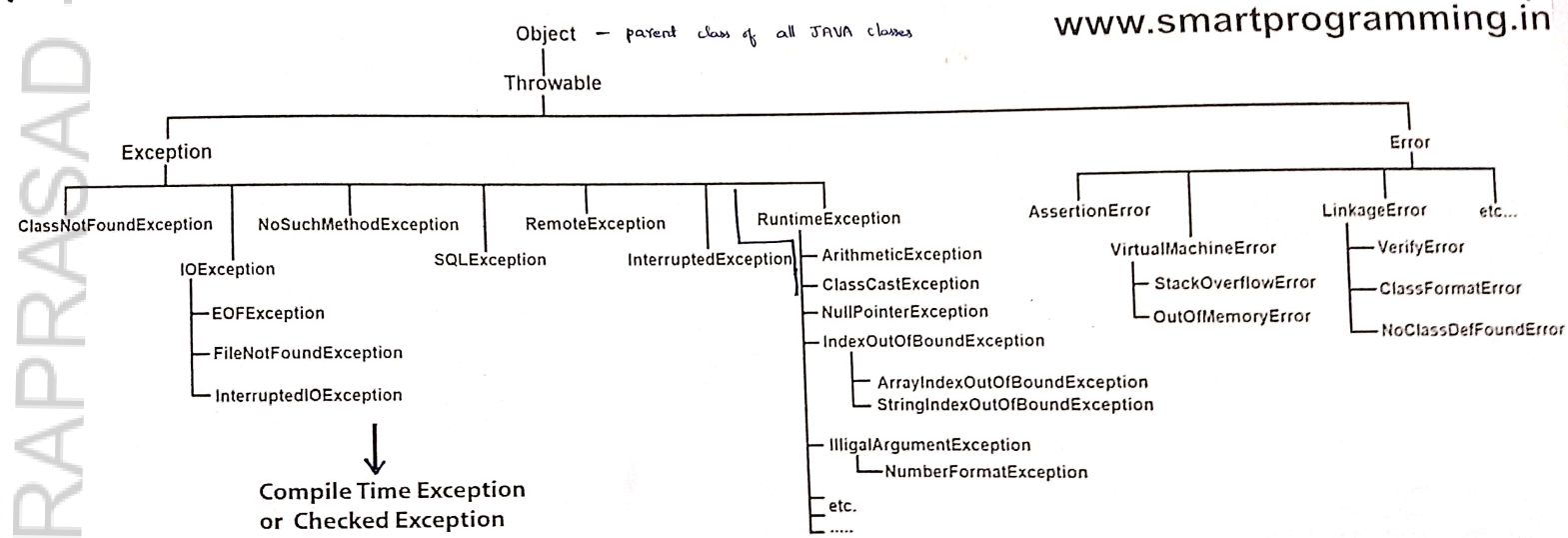
- Exceptions are recoverable / handled, errors are not possible to recoverable, need to contact administrator.

→ Compile time Exception — Checked Exceptions

→ Run-time Exception — Unchecked Exception

→ Run-time Error — Unchecked Exception (e.g. out of memory)

Hierarchy Of Exception class



*. Difference between checked & Unchecked Exception :

- All the exceptions occur only at run-time. No exceptions occur at compile-time.

e.g., FilenotFound exception during compile time.

*. Checked Exception – The exceptions are checked and handled at compile time

*. unchecked Exception – The exceptions are not checked at the compile time.

- If some code within a method throws a checked exception, than the method must either handle the exception or it must specify the exception using "throws" keyword.

- checked Exceptions are direct methods / subclass of Exceptions class but do not inherit from RuntimeException

- They (Unchecked) are direct subclass of RuntimeExceptionCl

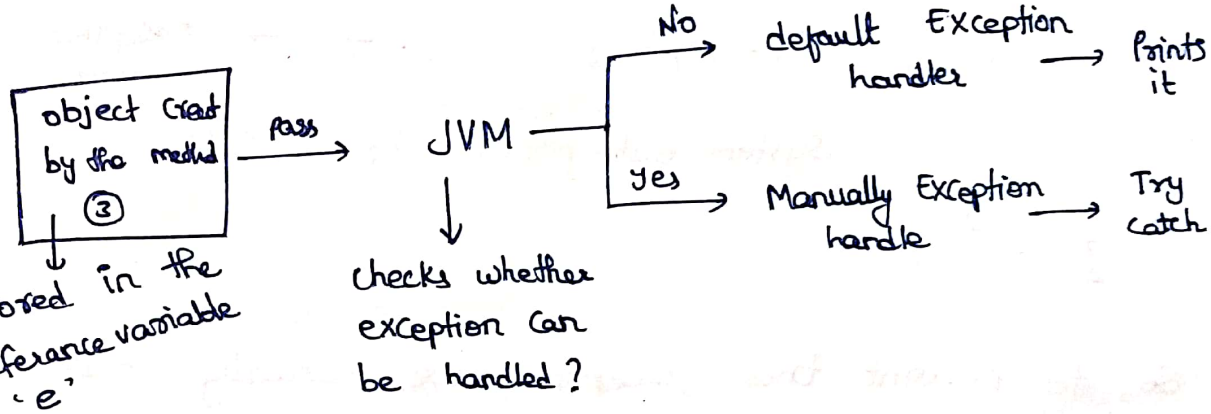
*. checked exception → Compile-time exception.

*. unchecked exception → Run-time exception.

*. TRY - CATCH : Exception Handling

- Whenever there is exception, the method in which exception occurs will create an object and that object will store three things :

① Exception name ② Description ③ Stack trace



IMP
- We can handle exception using 5 keywords :

1- try 2. Catch 3. finally 4. throw 5. throws

Syntax :

```
try
{
    // risky code
}
Catch (ExceptionClassName referVariableName)
{
    // handling code
}
```

Example :

Arithmetic exception division by zero :

```
class Test
{
    public static void main (String args[])
    {
        int a = 100, b = 0, c
        c = a/b ;
        System.out.println (c);
    }
}
```

→ runtime exception occurs here.

So, to prevent this exception, we manually handle the exception using try catch keyword / blocks.

So, Exception handled as follows,

```
class Test
{
    public static void main (String args[])
    {
        try
        {
            int a=100, b=0, c ;
            c = a/b ;
            System.out.println (c);
        }
        catch (ArithmeticException e)        // Exception
        {
            System.out.println (e) ;
        } System.out.println ("you can't divide by 0");
    }
}
```

when you don't know exception name

- Catch block doesnot get control unless there is an exception in try block. If there is no exception in the try block, then catch block doesn't excecute.

Example :

```
class Test
{
    public static void main (String args[])
    {
        System.out.println("1");
        try
        {
            System.out.println("2");
            int a=100, b=2, c;
            c = a/b ;
            System.out.println(c);
            System.out.println("3");
            System.out.println("4");
        }
        catch (Exception e)
        {
            System.out.println("5");
            System.out.println(e);
            System.out.println("6");
        }
        System.out.println("7");
        System.out.println("8");
    }
}
```

→ no exception so,

→ * anything should n't be written between

→ So, catch block doesn't excecute

output :

```
1
2
50
3
7
8
```

*. METHODS To print Exception :

IMP
1) e.printStackTrace();

It prints all the Exception information like name of exception, description, complete details where & why exception occured.

2) System.out.println(e); (or) System.out.print(e.toString())

It prints exceptionname, description but doesnot print StackTrace information.

3) System.out.println(e.getMessage());

It doesn't print exception name ✗.
It prints description of exception.
It doesn't print stack trace info.

*. Finally Block in Java :

- Finally is the block that is always executed whether exception is handled or not.
- Exception is handled by catch block.
- It can be used after try-catch or directly after try block.

(A) If exception occurs, exception is found in try block then object is created by that method and stored in e, the exception is handled by catch block & details of the exception are printed and then it comes to execute finally block.

(B) If exception doesnot occurs, exception is not found in try block, so catch block doesnot executes & directly it comes to execute finally block.

Syntax : IMP

```
try
{
    // risky code
}
catch (Exception e)
{
    // exception handling code
}
finally
{
    // code, cleanup code
}
```

risky code or code you have doubt.

print exceptions, alerts, warnings, info of exception

closing file, connections
release memory, lock
closing database connections

- If any exception occurs while reading or writing a file, then below code will not execute and thus resource will not close.

IMP

- We can use multiple catch blocks with one try block but we can only use single finally block with one try block not multiple.

IMP

- The statements present in the finally block execute even if the try block contains control transfer statements like jump statements, return, break, continue

IMP

- The possibilities that distribs the execution of finally block are :

Case-i : Using the syntax : System.exit() method

→ This closes the JVM & finally block doesn't execute.

Case-ii : Causing a fatal error that causes the process to abort. (out of memory exceptions)

Case-iii : If there are some errors in try block then finally block doesn't execute.

Case-iv : Due to an exception arising in finally block.

Case-v : Death of Thread (Multi-threading)

very IMP

*. Difference between Final, Finally, Finalize : (interviews)

FINAL

1. Keyword
2. Use with :
   a. Variable (value becomes constant or fixed)
   b. method (doesnot to be overide)
   c. class (cannot be inherited)

Example :

```
final int a = 10;
final void show()
{
}
final class Test
{
}
```

extends

FINALLY

1. Block
2. use with either try or try-catch blocks.

example :

```
try
{
  //
}
catch (Exception e)
{
  //
}
finally
{
  //clean up code
}
```

```
try
{
  //
}
finally
{
  //
}
```

3. Used to write clean-up code after error Handling.

FINALIZE

1. Method
2. Method is override for an object
3. Directly can be (finalize) overided.

Example :

```
protected void finalize() throws Throwable
{

}
```

4. Executes before Garabage collection
5. Used to write clean-up code before garbage collection.

*. Different combinations of try - catch - finally

(1)
```
try
{
}
```
✗

(2)
```
Catch (Exception e)
{
}
```
✗

(3)
```
finally
{
}
```
✗

All these individually doesnot work

(4)
```
try
{
}
```
✓
```
Catch (ArithmeticException e)
{
}
```
✓
```
Catch (Exception e)
{
}
```
✓

(5)
```
try
{
}
```
✓
```
Catch (Exception e)
{
}
```
✓
```
Catch (ArithmeticException e)
{
}
```
✗

(6)
```
try
{
}
```
✗
```
Catch (ArithmeticEx
{
}
```
✗ (↑ Same)
```
Catch (ArithmeticEx
{
}
```
✗

(7)
```
try
{
    try
    {
    }
    Catch (—)
    {
    }
}
Catch
{
}
```
✓ ✓ ✓

(8)
```
try
{
}
```
✓
```
Catch (—)
{
    try
    {
    }
    Catch (—)
    {
    }
}
```
✓ ✓

(9)
```
try
{
}
catch (—)
{
}
finally
{
}
```
✓ ✓ ✓

(10)
```
try
{
}
```
✓
```
finally
{
}
```
✓

```
try
{
}
finally
{
    try
    {
    }
    Catch
}
```
✓ ✓

(11)
```
try
{
}
finally
{
   try
   {
   }
   catch(—)
   {
   }
}
```

(12)
```
try
{
}
finally
{
}
catch(—)
{
}
```

(13)
```
try
{
}
System.o.p("");
catch(—)
{
}
```

(14)
```
try
{
}
catch(—)
{
}
S.O.P(" ");
:
:
```

```
try
{
}
S.O.P(" ");
finally
{
}
```

(15)
```
S.O.P( )
try
{
}
S.O.P( )
finally
{
}
```

* "throw" keyword :

Syntax : throw new ExceptionClassName ("String here");

```
class Test
{
    public static void main (String args[])
    {
        throws new ArithmeticException ( );
    }
}
```

- throw keyword used for custom exception / user defined exception.
- Always create unchecked exception

Example :

```
class YoungerAgeException extends RuntimeException
{
    YoungerAgeException (String message)
    {
        super (message);
    }
}

class Voting
{
    public static void main (String args[])
    {
        int age = 16;
        if (age < 18)
        {
            throw new YoungerAgeException ("You can't vote");
        }
        else
        {
            System.out.println (" Please vote");
        }
    }
}
```

output :

*. Throws Keyword :

- "throws" Keyword is used to declare an exception. It gives an information to the caller method that there are may occur an exception so it is better for the caller method to provide the exception handling code. so that normal flow can be maintained.

Example :

```
import java.io.*;
class ReadAndWrite
{
    void readfile() throws FileNotFoundException
    {
        FileInputStream fis = new FileInputStream("d:/A.txt");

    }
```

Syntax : throws ExceptionClass

→ FileInputStream class throws "FileNotFoundException" which is compile time exception or checked exception so we have to handle the exception and for this purpose we have to use either "try-catch" or "throws" keyword.

- Syntax : throws ExceptionClass

```
/* continuation of code */

    void savefile() throws FileNotFoundException
    {
        FileOutputStream fos = fileOutputStream("D:/A.txt");

    }
```

/* Continuation */

```
Class Test
{
    public static void main (String args[])
    {
        ReadAndWrite rw = new ReadAndWrite();
    try {
        rw.readfile();
    }
    Catch ( FileNotFoundException e)
    {
        ——————
    }
    try
    {
        rw=savefile();
    }
    Catch ( — )
    {
        ═══════
    }
    }
}
```

IMP
* throws keyword is always used with Checked Exception or Compile time Exception.

- "throws" keyword is used to declare only for the checked exceptions. If there occurs any unchecked exception such as NullPointException, it is programmers fault that he is not performing check up before the code being used.

* Compile-time exception can be prevented by handling using :

① By using try-catch ② Using "throws" keyword

*. Working of these keywords :

① try : In try block we write statements that can throw exception i.e. it maintains risky code.

② catch : It mantains exception handling code i.e. alternative way for exception

③ finally : It maintains clean up code i.e. closing the resources.

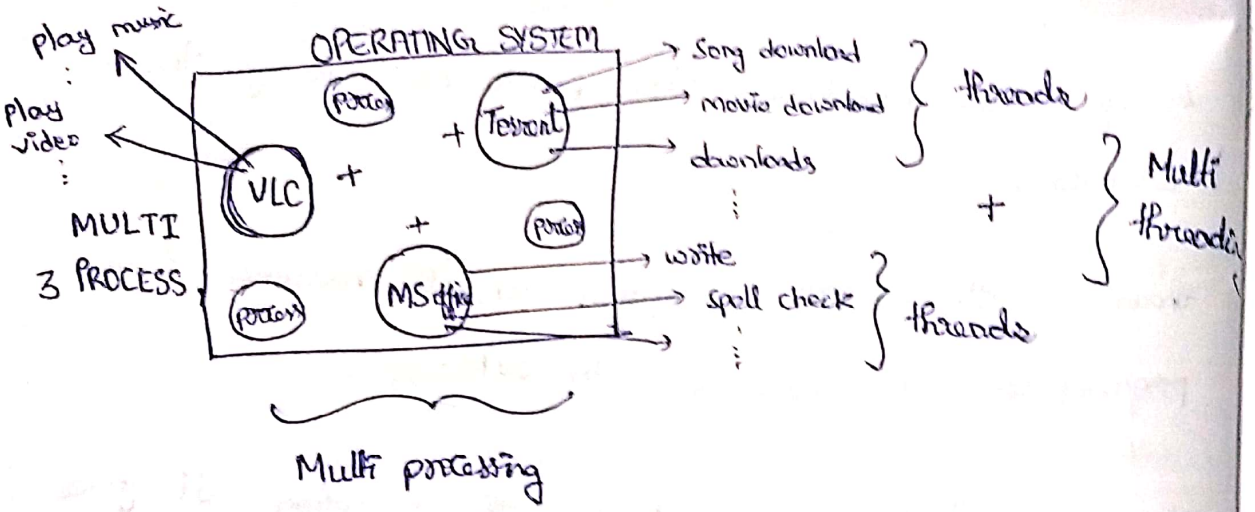
④ throw : It creates exception object manually (by programmers) and handover to JVM.

⑤ throws : It is used to declare the exception. It gives an information to the caller method that there may occur an exception so it is better for the caller method to provide the exception handling code so that normal flow can be maintained.

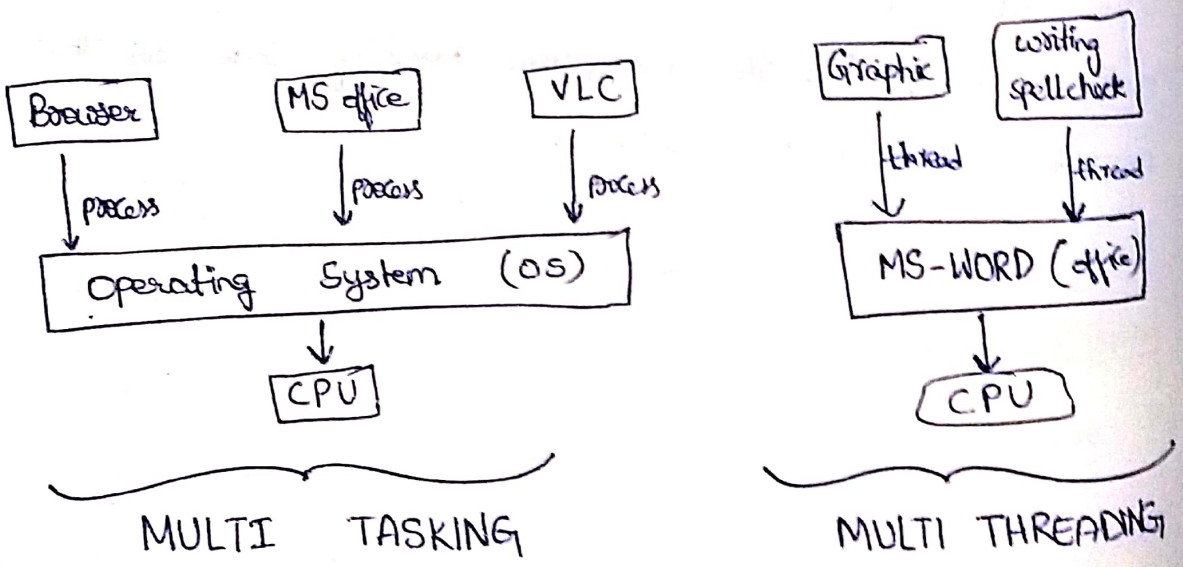
MULTI THREADING

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

- A thread is lightweight sub-process, the smallest unit of processing.



- Difference between Multithreading & Multitasking :



1. Multiprogramming : A computer running more than one program at a time

2. Multiprocessing : A computer using more than one CPU at a

3. Multi tasking : Tasks sharing a common resource

4. Multi threading : extension of multi tasking.

*. MULTI TASKING :

→ Performing multiple task at single time

→ Increases the performance of CPU.

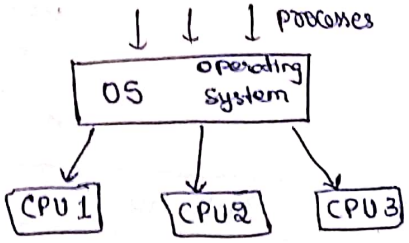
→ 2 types : achieved through

1) process based Multi-tasking (MP)

2) Thread based Multi tasking (MT)

(1.) MULTI PROCESSING :

→ When one system is connected to multiple processors in order to complete the task.

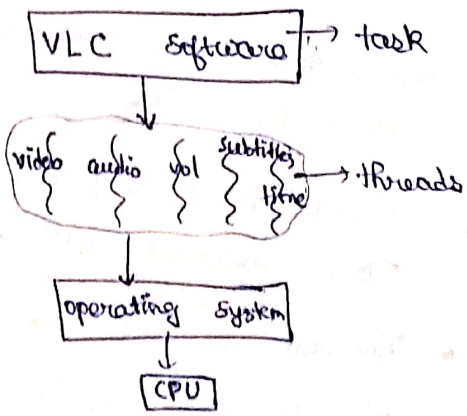


→ It is best suitable at system level or OS level.

(2.) MULTI THREADING :

→ Executing multiple threads (sub process) at single time

→ Used : Softwares, Games, Animations



Example :

```
class VLC
{
    public static void main( )
    {
        ==  playVideo( );
            startMusic( );
    }
}

class Video
{
    void playVideo( )
    {
        ==
    }
}

class Music
{
    void startMusic( )
    {
        ==
    }
}
```

→ Multithreading is best suitable at programming level.

→ Java provides pre-defined API's for Multithreading.

→ Class Names like
1. Thread
2. Runnable
3. ThreadGroup
4. Concurrency
5. ThreadPool

→ thread 1 is created

→ create thread 2

takes so much time for creating

takes less time for creating

PROCESS	THREAD
• A program which is in executing state.	• It is sub part of process
• A process is heavy weight	• A thread is light weight
• process takes more time for context switching	• thread takes less time for context switching.
• Inter process communication takes more time	• Inter thread communication takes less time.
• Each process has different address space.	• Threads in same process have same address space.
• process is Independ of each other	• Threads are dependent on each other
• process doesn't need Synchronization.	• Threads need synchronization
• process uses so many resources	• Resource consumption is less

→ 2 ways to create thread :

(I) Thread class (II) Runnable Interface

→ Thread class is in java.lang package

Example :

```
class Test extends Thread
{
    public void run()
    {
        // tasks are provided threads
    }
    public static void main(String args[])
    {
        Test t = new Test();
        t.start();
    }
}
```

Algorithm to create thread.

1. Extends thread class
2. Override run() method
3. Create an object of Our class
4. Start the thread by using start() method.

Thread lifecycle 5 stages

1. Create new thread
2. Runnable state of new thread
3. JVM will allocate the processor by using Thread scheduler
4. Running state
5. Dead state
6. Sleep mode / non runnable threads.

Check picture (after)

```
class Thread implements Runnable
{
    // constructor
    // methods
    1) run()
    2) start()
    3) sleep()
    4) join()
    5) getName()
    6) setName()
    7) interrupted, priority
    8) daemon
}
```

```
public interface Runnable
{
    // method only 1
    public abstract void run();
}
```

→ package java.lang

"Thread" class Constructors

```
public class Thread implements Runnable
{
    public Thread() { - }
    public Thread(Runnable target) { - }
    public Thread(String name) { - }
    public Thread(Runnable target, String name) { - }
    public Thread(ThreadGroup group, Runnable target) { - }
    public Thread(ThreadGroup group, String name) { - }
    public Thread(ThreadGroup group, Runnable target, String name) { - }
    public Thread(ThreadGroup group, Runnable target, String name, long stackSize) { - }

    ----- (and methods)
}
```

"Thread" class methods (Part 1)

```
public class Thread implements Runnable
{
    public void run() { - }
    public synchronized void start() { - }
    public static native Thread currentThread();
    public final native boolean isAlive();

    public final String getName() { - }
    public final synchronized void setName(String name) { - }

    public final boolean isDaemon() { - }
    public final void setDaemon(boolean on) { - }

    public final int getPriority() { - }
    public final void setPriority(int newPriority) { - }

    ----- (many more methods)
}
```

Basic Methods

Naming Methods

Daemon Thread Methods

Priority Based Methods

"Thread" class methods (Part 2)

```
public class Thread implements Runnable
{
    ----- (many more methods)

    public static native void sleep(long millis) throws InterruptedException;
    public static native void yield();
    public final void join() throws InterruptedException { - }
    public final void suspend() { - }
    public final void resume() { - }
    public final void stop() { - }
    public void destroy() { - }

    public void interrupt() { - }
    public boolean isInterrupted() { - }
    public static boolean interrupted() { - }
}
```

Prevent Thread Execution Methods

removed from java

Deprecated methods

Interrupting a thread Methods

(I)

```
class Test extends Thread
{
    public void run()
    {
        // tasks of thread
        System.out.println("thread task");
    }
    public static void main(String args[])
    {
        Test t = new Test();
        t.start();
    }
}
// Cannot invoke thread twice. If done, it shows Exception.
```

(II)

```
class Test implements Runnable
{
    public void run()
    {
        // tasks of thread
        System.out.println("thread task 2")
    }
    public static void main(String args[])
    {
        Test t = new Test();
        Thread th = new Thread(t);   <- IMP step
        th.start();
    }
}
```

IMP * Better way : Runnable way is better by implemented.

Reason : Multiple inheritance is not possible in java.

If extends + extends → not possible

But extends + implements → possible

Like, class A extends B → if we want to make A class thread then extends B extends thread ✗ not possible

So, class A extends B implements Runnable ✓

Algorithm :

1. Implements the Runnable Interface.
2. Override the run() method.
3. Create an object of (Test) the class.
4. Create an object of Thread class and pass the parameter in constructor in Thread class.
5. Start (invoke) the thread.

*. MULTI THREAD :

① Performing single task from single Thread:

```
class Test extends Thread
{
    public void run()
    {
        S.O.P("Task 1");
    }
    public static void main(String[] args)
    {
        Test t = new Test();
        t.start();
    }
}
```

Total threads = 2 → main thread by JVM
→ User created thread

- If user creates 'n' threads ⟹ Total threads = (n+1)

② Performing single Task from Multiple Thread :

```
class Test extends Thread
{
    public void run()
    {
        s.o.p ("Task 1");
    }
    public static void main (String[] args)
    {
        Test t1 = new Test();
        t1 - start();
        Test t2 = new Test();
        t2 - start();
        :
    }
}
```

Total threads = 3 → main thread by JVM
→ 2 user created threads.

③ Performing Multiple task from Single Thread :

- This case is not possible because a single thread cannot perform Multiple task.
- Like example for VLC player, A single thread isn't enough for doing video task, music task, progress task, It happens in a way like 1st video task then music task, then subtitles, then progress. It is the similar old scenario.

IMP

④ Performing multiple task from Multiple Thread.

```
class MyThread1 extends Thread
{
    public void run()
    {
        S.O.P ("task 1");
    }
}
class MyThread2 extends Thread
{
    public void run()
    {
        S.O.P("task 2");
    }
}

class Main
{
    public static void main (String[] args)
    {
        MyThread1 t1 = new MyThread1();
        t1.start();

        MyThread2 t2 = new MyThread2();
        t2.start();
    }
}
```

- To Create 2 threads, we need to create 2 classes. So, to Create n threads, we need to create n classes and from Main method, Create n objects of those n classes to start the threads (invoke). (threads)

* METHODS OF THREAD CLASS :

Example :

```
public class Main
{
    public static void main (String args[])
    {
        System.out.println ("Hello");
    }
}
```

Here whenever you write code inside main method, JVM creates a thread & invokes it. By default this process happens irrespective of other extends, implements.

So, to get its' name use :

```
System.out.println ( Thread.currentThread().getName()); //main
Thread.currentThread().setName ("GVP");
System.out.println (Thread.currentThread().getName()); //GVP
```

Example :

```
class Test extends Thread
{
    public void run()
    {
        System.out.println ("Task thread");
        System.out.println (Thread.currentThread().getName()); // Thread0
    }

    public static void main (String[] args)
    {
        Test t = new Test();
        t.start();
        System.out.println (Thread.currentThread().getName()); // main
    }
}
```

Thread created by main
Thread named by JVM
Thread executed by programmer

or here
or same
here
t.getName();
t.setName();

In Main if you code

```
System.out.println ( Thread.currentThread.isAlive ()); // true
```

But if a user defined class / method is

```
System.out.println ( t.isAlive ()); // false or True
                                    depending on execution
```

- Here t is object of class created in main method.

*. DAEMON THREAD :

- which run in the background of another thread
- It provides service to the threads running only.
- Example : Garbage collector, finalizer, Attach listner, Signal dispatch

Like In case of Garbage collector, Suppose the main method creates a main thread to run. But if there is no sufficient memory / memory leakage for main method, Garbage collector creates a thread that runs in the background of another thread. It provides service to the thread to remove unwanted variables & Creates some memory to run main method / thread properly.

- Methods : a. public final void setDaemon (boolean b)
  b. public final boolean isDaemon ()

setDaemon → converts a given thread to Daemon thread if set True
isDaemon → Is it Daemon Thread or not a Daemon Thread.

Example :

```
class Test extends Thread
{   public void run() {
      System.out.println (" child Thread");
    }
    public static void main (String args[])
    {
        Test t = new Test ();
        t.setDaemon (true);
        t.start ();
    }
}
```

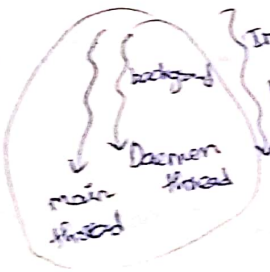
Right process ✓

```
t.setDaemon(true);
t.start();
```

Wrong process ✗

```
t.start();
t.setDaemon(true);
```

- We have to create Daemon thread before starting of the thread if we create daemon thread after starting it, it will now throw run-time exception i.e. IllegalThreadStateException
- we cannot create Main method (thread) as Daemon thread because main method starts before only (it will start,) after that if we create daemon thread, it will throw runtime excep
- The life of Daemon Thread depends on another thread for which it is servicing.
- Daemon Thread inherits properties / nature from its' parent thread.



→ setDaemon(boolean);

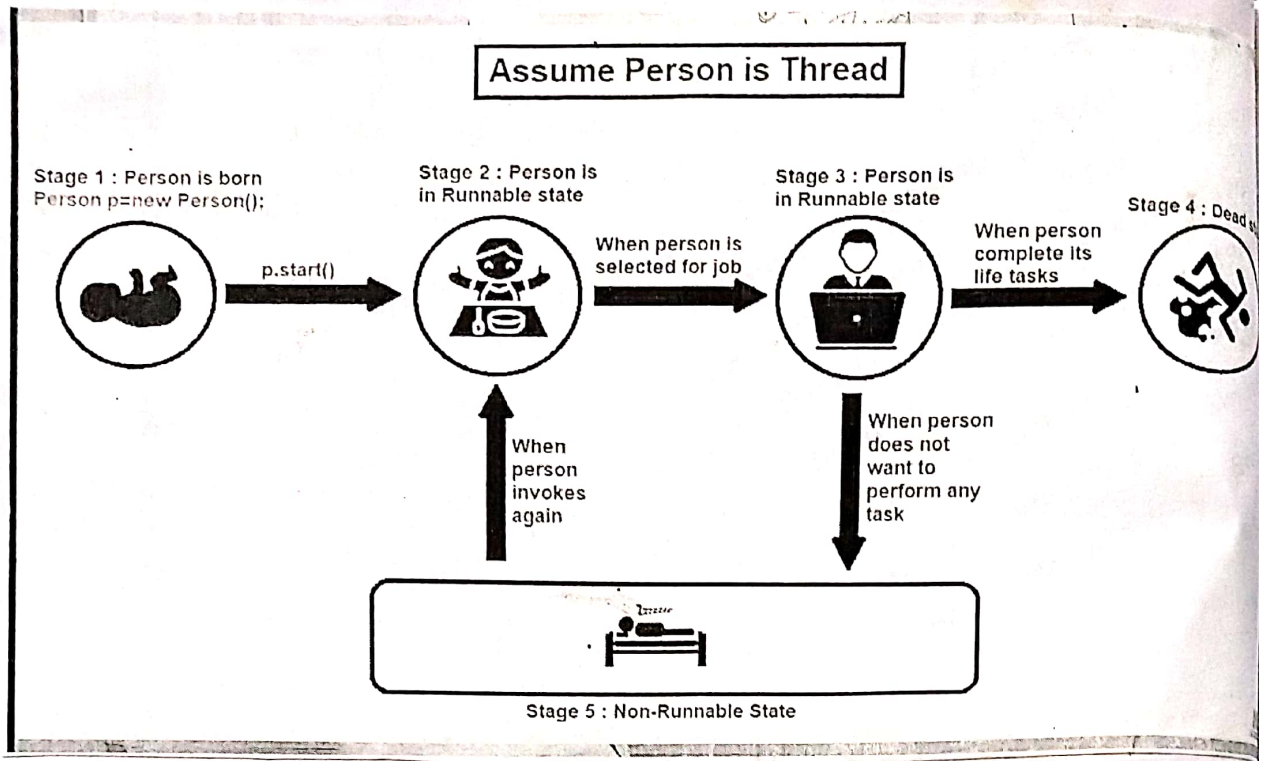
→ isDaemon();

- JVM doesn't depend on Daemon thread running. It will make Daemon thread dead state (kill) & it will shut-down.
- Always set Daemon Thread priorities low. By default it will be low priority most of times. We can change its' priority according to our need.

"Thread" class methods (Part 3)

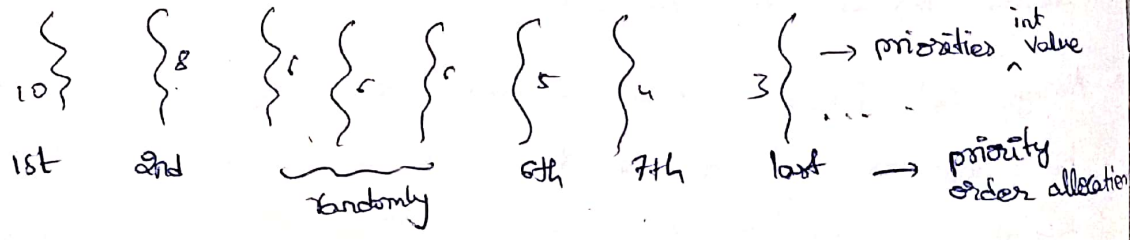
```
public class Object
{
    public final void wait() throws InterruptedException { - }
    public final native void notify();
    public final native void notifyAll();
}
```

→ Inter-Thread Communication Methods



*. THREAD PRIORITIES : (priorities)

- JVM provides the priorities to each thread and according to those priorities JVM allocates the processor.
- JVM decides (based on internal algorithms) if the priorities of all the threads is same.
- Priority of thread is just an integer number given inorder to allocate the processor to it depending on its value



- priorities are represented in the form of integer value from (1 to 10) range only.

✓ 1 → MIN_PRIORITY
5 → NORM_PRIORITY
10 → MAX_PRIORITY

✓ 0, <1, >10, high, low, medium are not the priorities.

✓ Main thread default priority = 5.

- public final void setPriority (int value);
- public final int getPriority ();

Example :

```
class Test extends Thread
{
    public void run()
    {
        System.out.println ("child thread");
        System.out.println (Thread.currentThread().getPriority()); //5
    }
    public static void main (String[] args)
    {
        System.out.println (Thread.currentThread().getPriority()); //5
        Test t = new Test();
        t.start();
            t.setPriority (10);
    }
}
```

- Priorities are inherited from parent thread.
- By default Main Thread priority = 5.
- If priority value is not between (1 to 10) in this range, it throws run time exception i.e. IllegalArgumentException.

IMP
- Priorities depends on the platform & windows doesnot support priorities. (O.S)

*. SleepMethod();

1. public static native void sleep (long milli) throws InterruptedException
2. public static void sleep (long milli, int nano) throws InterruptedException

overloaded method

Example :

```
class Test extends Thread
{
    public void run
    public static void main (String[] args)
    {
        for (int i=1 ; i<=5 ; i++)
        {
            try
            {
                Thread.sleep(1000);
                System.out.println (i);
            }
            Catch (Exception e)
            {
                System.out.println(e);
            }
        }
    }
}
```

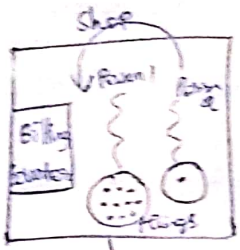
→ Start
1
→ 1 sec gap
2
→ 1 sec gap
3
→ 1 sec gap
4
→ 1 sec gap
5
→ exit

[OR]

```
class Test extends Thread
{
    public void run()
    {
        for (int i=0 ; i<=5 ; i++)
        {
            try
            {
                Thread.sleep(1000);
                System.out.println(i);
            }
            catch (Exception e)
            {
                System.out.println(e);
            }
        }
    }
    P. S. V. M ( )
    {
        Test t = new Test();
        t.start();
    }
}
```

yeild : which stops the currently executing thread & give a chance to other threads for execution.

Example:



→ stops wait & gives chance to another person

Java 5 : internally it works sleep();

Java 6 : Thread provides the hint to the thread-scheduler, then it depends on Thread-scheduler to accept or ignore the hint.

- public static native void yield();
- output mostly depends on Thread scheduler.

Example:

```
class Test extends Thread
{
    public void run()
    {
        for(int i=0; i<=5; i++)
        {
            S.O.P(Thread.currentThread().getName()+"-"+i);
        }
    }
p. s. v. m()
{
    Test t = new Test();
    t.start();
    for(int i=0; i<=5; i++)
    {
        Thread.yeild();
        System.out.println("Main thread"+i);
    }
}
}
```

*. Join Method ( ) :

- If a thread wants to join / to wait for another thread to complete its task then use join( ) method.

- a. public final void join ( ) throws InterruptedException { - }
  b. public final synchronized void join (long milli) throws Interrupt
  c. public final synchronized void join (long milli, int nano) " "

Example :

```
class Test extend Thread
{
   public void run( )
   {
      try {
      for (int i=0 ; i<=5 ; i++)
      {
         s.o.p(i);
      }
   }
   catch {—} {— }
}
 p. s. v. m ( )
{
    Test t = new Test();
     t.start();  ——————→ t.join();
    try {
     for (int i=0 ; i<=5 ; i++)
    {
       s.o.p ("Main thread" +i);
      Thread.sleep (1000);

   }
  }
  catch(—) { — }
}
}
```

sleep() method important points

1. If the value of milliseconds is negative then "IllegalArgumentException" is thrown.
2. If the value of nanoseconds is not in the range 0-999999 then "IllegalArgumentException" is thrown.
3. Whenever we want to use the sleep() method we also need to handle the "InterruptedException", If we will not handle it, the JVM will show a compilation error.
4. When any thread is sleeping and if any other thread interrupts it, then it throws "InterruptedException".
5. The sleep() method always pauses the current thread execution. When the JVM finds the sleep() method in code, it checks that which thread is running and pause the execution of thread.
6. When we use sleep() method to pause the execution of thread. the thread scheduler assigns the CPU to another thread if any thread exists. So, there is no guarantee that the thread wakes up exactly after the time specified in sleep() method. It totally depends on the thread scheduler.
7. While the thread is sleeping, it doesn't lose any locks or monitors that it had acquired before sleeping.

sleep(), yield() & join() methods prototype

sleep() method Prototype :-

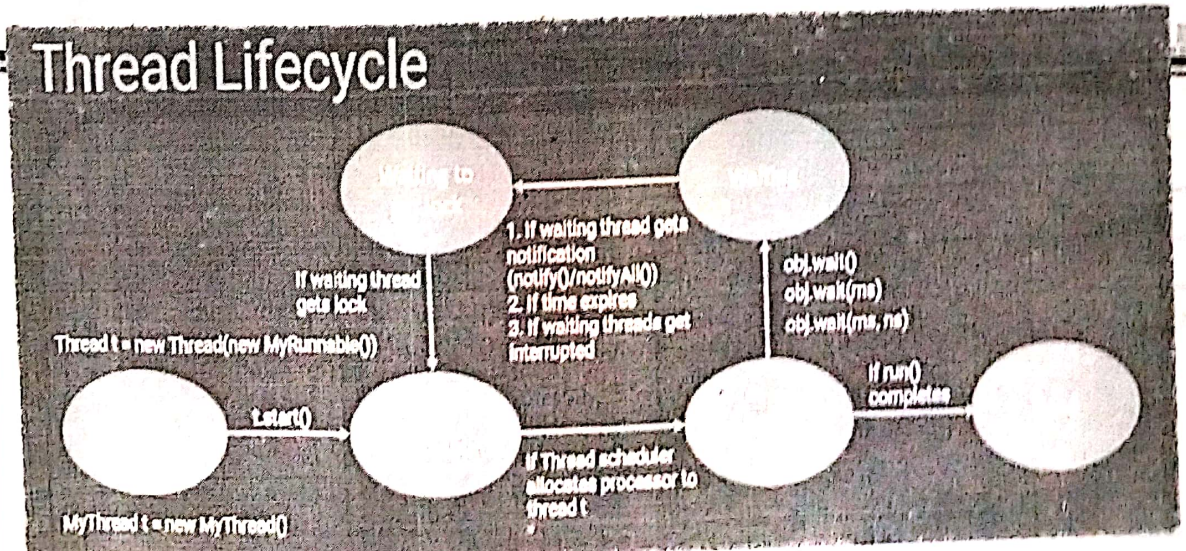
1. public static native void sleep(long ms) throws InterruptedException;
2. public static void sleep(long ms, int ns) throws InterruptedException;

yield() method Prototype :-

1. public static native void yield();

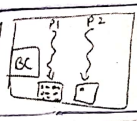
join() method Prototype :-

1. public final void join() throws InterruptedException;
2. public final synchronized void join(long ms) throws InterruptedException;
3. public final synchronized void join(long ms, int ns) throws InterruptedException;



Differences between sleep, Yeild, join();

These all 3 methods temporary stops the thread execution

property	sleep()	yield()	join()
1) purpose	If any thread does not want to perform any operation for particular time	It stops the current executing thread & provide chance to another threads of same or higher priority to execute	If a thread wants to wait for another thread to complete its task
2) Examples	1) Timer, ppt, blinking bulbs	1) Shopping 	1) Licence Dept.
3) How the thread implees again?	1) automatically after provided time period. 2) If thread is interrupted.	1. Automatically invoked by thread-schedules	1. Automatically invokes after completion of another thread task. 2. After completion of time period. 3. If thread is interrupted.
4) Methods:-	1) sleep(long ms) 2) sleep(long ms, int ns)	1) yield()	1) join() 2) join(long ms) 3) join(long ms, int ns)
5) Is method overloaded?	Yes	No	Yes
6) Exception	Yes(InterruptedException)	No.	Yes(InterruptedException)
7) Is method final	No	No	Yes
8) Is method static	Yes	Yes	No
9) Is method native	1) native 2) non-native	Yes	No

*. THREAD SYNCHRONIZATION :

- When two or more threads needs access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time.
- The process by which this is achieved is called Synchronization.
- Key to synchronization is the concept of monitor (like semaphore).
- A monitor is an object that is used as mutually exclusive lock, or mutex. Only one thread can own a monitor at a given time. When a thread acquires a lock, it is said to have entered the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor. These other threads are said to be waiting for the monitor.

*. Java Synchronized method :

- If you declare any method as synchronized, it is known as synchronized method.
- Synchronized method is used to lock an object for any shared resource.
- When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.
- To do this we need to use keyword synchronized just before the method.

Example :

```
class WithSynchronization
{
    synchronized void printTable(int n)
    {
        for (int i=1; i<=5; i++)
        {
            S.O.P(n*i);
            try
            {
                Thread.sleep(400);
            }
            catch
            {
                System.out.println(e);
            }
    } } }

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

class MyThread2 extends Thread
{
    WithSynchronization wSync;
    MyThread2(WithSynchronization wSync)
    {
        this.wSync = wSync;
    }
    public void run()
    {
        wSync.printTable(100);
} }
```

OUTPUT :

5 — sleep for 0.4 s
10 — sleep for 0.4 s
15 — sleep for 0.4 s
20 — sleep for 0.4 s
25 — sleep for 0.4 s
100 — sleep for 0.4 s
200 — sleep for 0.4 s
300 — sleep for 0.4 s
400 — sleep for 0.4 s
500 — sleep for 0.4 s

```
public class WithSynchronizationExample
{
    public static void main(String[] args)
    {
        WithSynchronization object = new WithSynchronization();
        MyThread t1 = new MyThread1(object);
        MyThread t2 = new MyThread2(object);
        t1.start();
        t2.start();
    }
}
```

*. Synchronized block :

- Synchronized block can be used to perform synchronization on any specific resource of the method.
- Suppose you have 50 lines of code in your method, but you want to synchronized only 5 lines, you can use synchronized block.
- If you put all the codes of the method in the synchronized block, it will work same as the synchronized method.

Syntax :

```
synchronized (object reference expression)
{
    // code block
}
```

Example :

```
Class Table
{
    void printTable(int n)
    {
        synchronized (this)
        {
            for (int i = 1; i <= 5; i++)
            {
                s.o.p (n * i);
```

```
            try
            {
                Thread.sleep(400);
            }
            catch (Exception e)
            {
                System.out.println(e);
            }
        }
    }
}

public class TestSynchronizedBlock2
{
    P.S.V.m()
    {
        final Table obj = new Table();
        Thread t1 = new Thread(){
        {
            public void run()
            {
                obj.printTable(5);
            }
        };

        Thread t2 = new Thread()
        {
            public void run()
            {
                obj.printTable(100);
            }
        };

        t1.start();
        t2.start();
    }
}
```

OUTPUT :

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

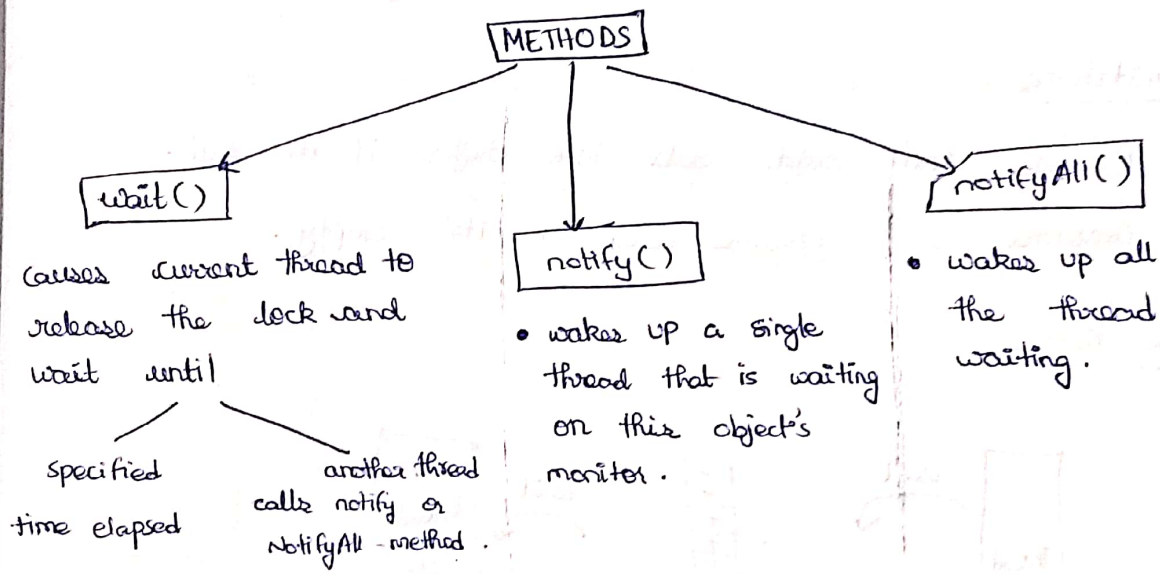
---

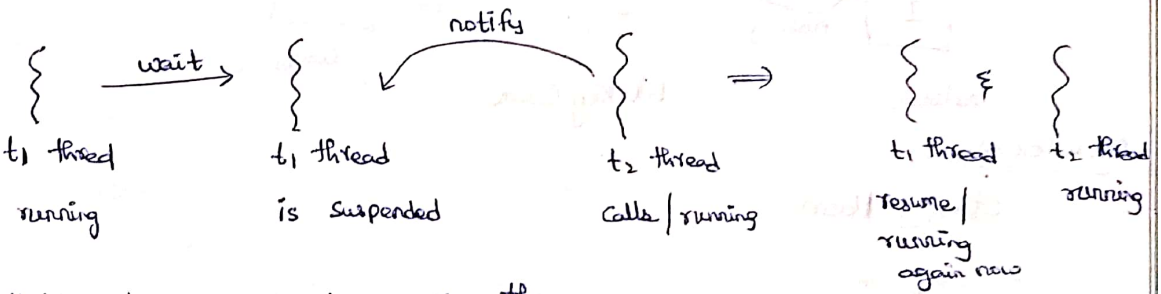
* . INTER - THREAD COMMUNICATION :

- Used to allow synchronized threads to communicate with each other.

* - METHODS used to :

- To avoid polling, java uses three methods, namely wait(), notify() & notifyAll();
- All these methods belong to object class as final so that all classes have them.
- They must be used within a synchronized block only.
- Polling : To check some condition repeatedly to take appropriate action, once the condition is true.





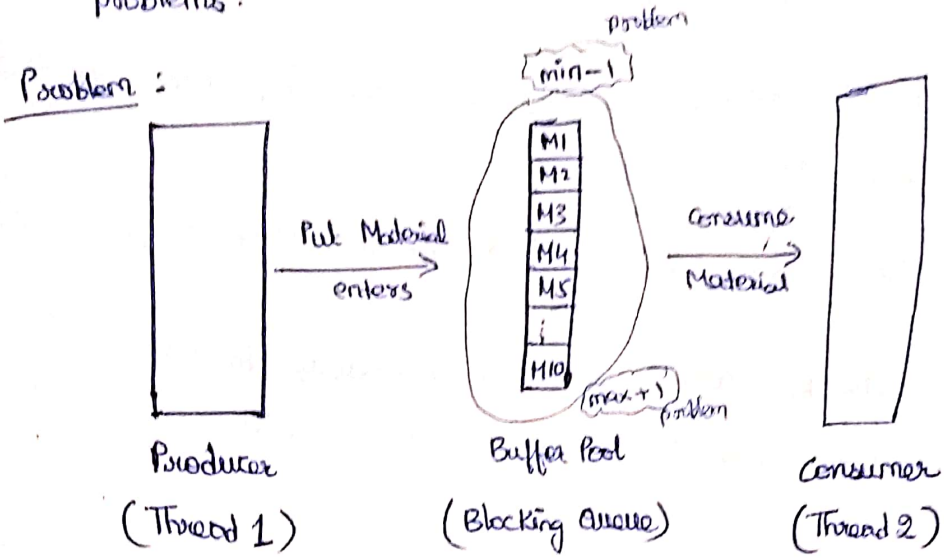
- notifyAll() does to wakeup all the threads that are waiting.

```
Public class Object
{
    public final void wait() throws InterruptedException{-}
    public final native void notify();
    public final native void notifyAll();
}
```

*. PRODUCER - CONSUMER PROBLEM :

- It is most commonly used multi-process Synchronization problems.

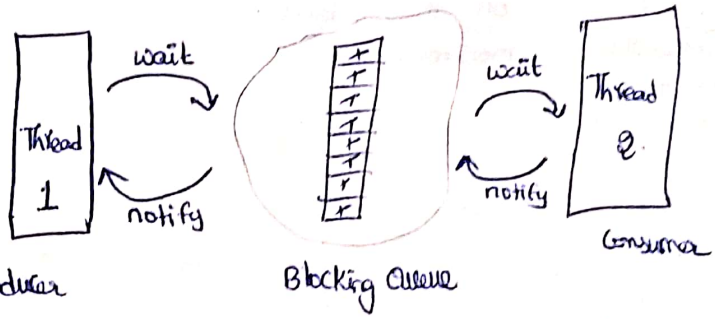
Problem :



Limitations :

- Producer can't add data into Buffer if its full.
- Consumer won't consume data if its empty.

Solution :



PSeudocode :

```
Class Item
{
    int i;
    boolean produced = false;
    public void synchronized produce (int x)
    {
        if (produced)
        {
            try
            {
                wait();
            }
            Catch(){ }
```

```
        i = x ;
        // print statements
        produced = true;
        notify ();
    }
```

```
Consume ()
{
    if (! produced)
    {
        try
        {
            wait();
        }
        catch() { }
    Produced = false;
    notify();
}
```

thread thread

Create class with these

Example :

```
public class ProducerConsumerTest
{
    public static void main (String[] args)
    {
        Container c = new Container();
        Producer p1 = new Producer(c,1);
        Consumer c1 = new Consumer(c,1);
        p1.start();
        c1.start();
    }
}
```

main method objects threads invoke Starts

```
class Container
{
    private int contents ;
    private boolean available = false

    public synchronized int get()
    {
        while (available == false)
        {
            try
            {
                wait();
            }
            catch (InterruptedException e)
            {
                System.out.println(e);
            }
        }
```

class for Container, Blocking Queue

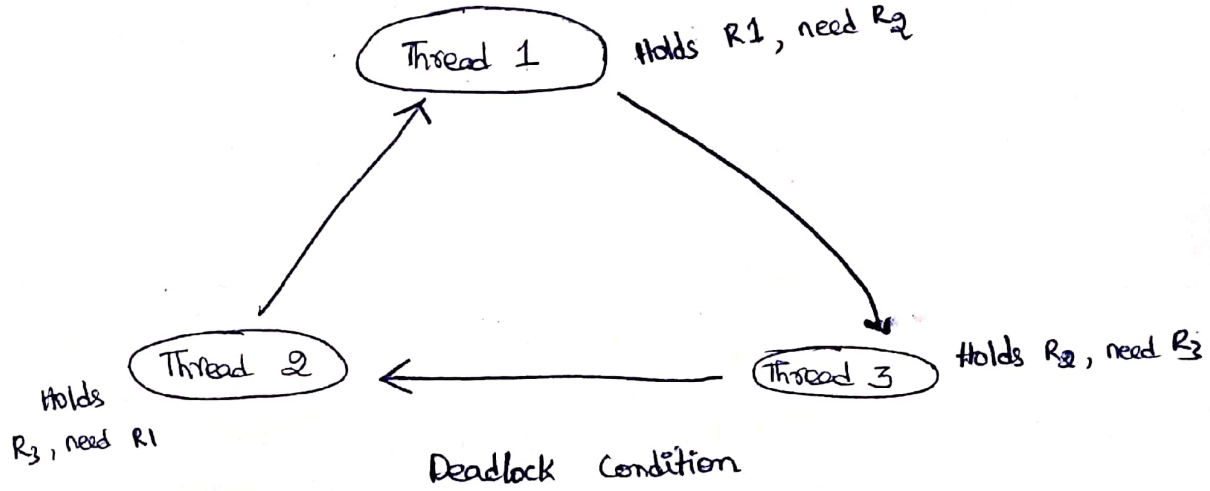
```
        available = false;
        notifyAll();
        return contents;
    }

    public synchronized void put (int value)
    {
        while (available == true)
        {
            try
            {
                wait();
            }
            catch (Exception e)
            {
                S.O.P(e);
            }
        }
        contents = value;
        available = true;
        notifyAll();
    }
}

class Consumer extends Thread
{
    private Container container;
    private int number;
    public Consumer (Container c, int number)
    {
        container = c;
        this.number = number;
    }

    public void run()
    {
        int value = 0;
        for (int i = 0; i < 10; i++)
        {
            value = container.get();
} } } S.O.P (this.number + value);
```

```
class Producer extends Thread
{
    private Container container;
    private int number;
    public Producer (Container c, int number)
    {
        container = c;
        this.number = number;
    }
    public void run()
    {
        for (int i=0; i<10; i++)
        {
            Container.put(i);
            S.O.P (this.number + i);
            try
            {
                sleep (1000);
            }
            catch (e)
            {
                S.O.P(e);
            }
        }
    }
}
```


---

*. DEAD LOCK :

- This occurs when two threads have a circular dependency on a pair of synchronized objects.



Deadlock Condition