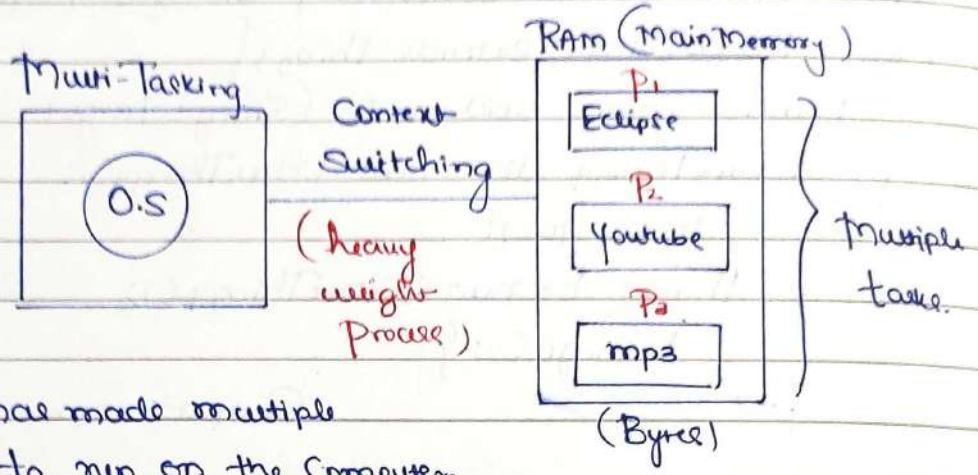
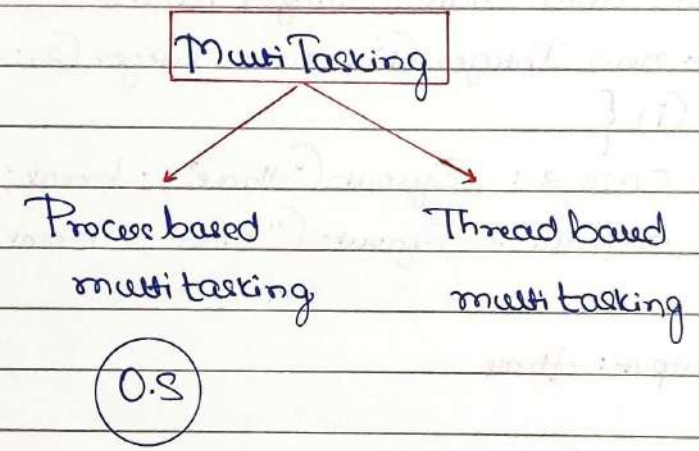
Multi-threading in Java.



O.S has made multiple tasks to run on the Computer.



Multitasking: Executing several tasks simultaneously is the concept of multitasking.

There are two types of multitasking:

a. Process based multitasking
b. Thread based multitasking

Process Based Multitasking: Executing several tasks simultaneously where each task is a seperate independent process such type of multitasking is called "process based multitasking".

eg:: listening to music, watching youtube

Process based multitasking is best suited at OS level.

Thread based multitasking: Executing several tasks simultaneously where each task is a seperate independent part of the same program, is called "Thread based multitasking".
Each independent part is called "Thread".

1. This type of multitasking is best suited at 'Programatic Level'.
The main advantages of multitasking is to reduce the response time of the system and to improve the performance

2. The main important application areas of multithreading are
a. To implement multimedia graphics
b. To develop web applications servers
c. To develop video games
d. To develop animations

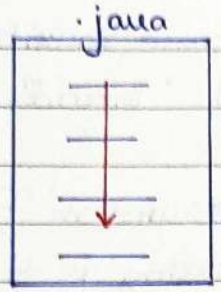
3. Java provides inbuilt support to work with threads through API called Thread, Runnable, ThreadGroup, ThreadLocal...

4. To work with multithreading, java developers will code only for 10% remaining 90% java API will take care.

Q5. What is a thread?
Seperate flow of execution is called "Thread". If there is only one flow then it is called "Single thread programming".
For every thread there would be a seperate job.

In java we can define a thread in 2 ways:
a. Implementing runnable interface
b. Extending thread class.

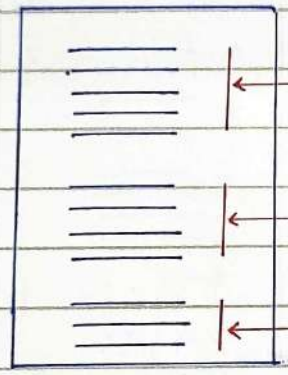
Jvm - Single-thread



More the waiting time, performance of the application would decrease.

Jvm - (threads are light weight)
Multiple - Thread / Multi Threading

T.S
Thread Scheduler
It divide the time to all the tasks.

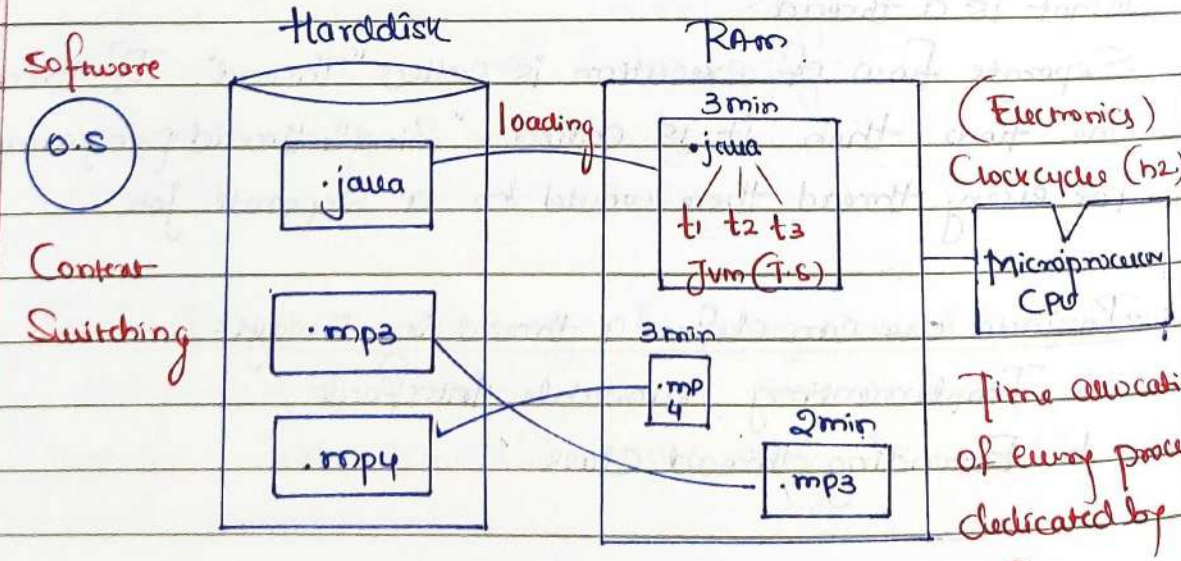


.java

* Less the response time from application, more would be the performance.

* To utilize CPU time effectively in our application

* To use multiple task and each task assign to one thread and promote "multi-threading".



Time allocation of every process is dedicated by O.S

Extending Thread Class

We can create a thread by extending a Thread.

```
Class MyThread extends Thread {
    @Override
    public void run(){
        for (int i=0; i<2; i++)
            { System.out.("Child Thread"); }
    }
}
```

→ //definining a thread (writing a class and extending a thread)

→ //job a thread (code written inside run())

```
Class ThreadDemo {
    public Static void main(String[] args)
    {   MyThread t = new MyThread(); // Thread initialization
        t.start(); // Staring a thread
            // at this line 2 threads are there
    for (int i=0; i<2; i++)
        { System.out.("main Thread"); }
    }
}
```

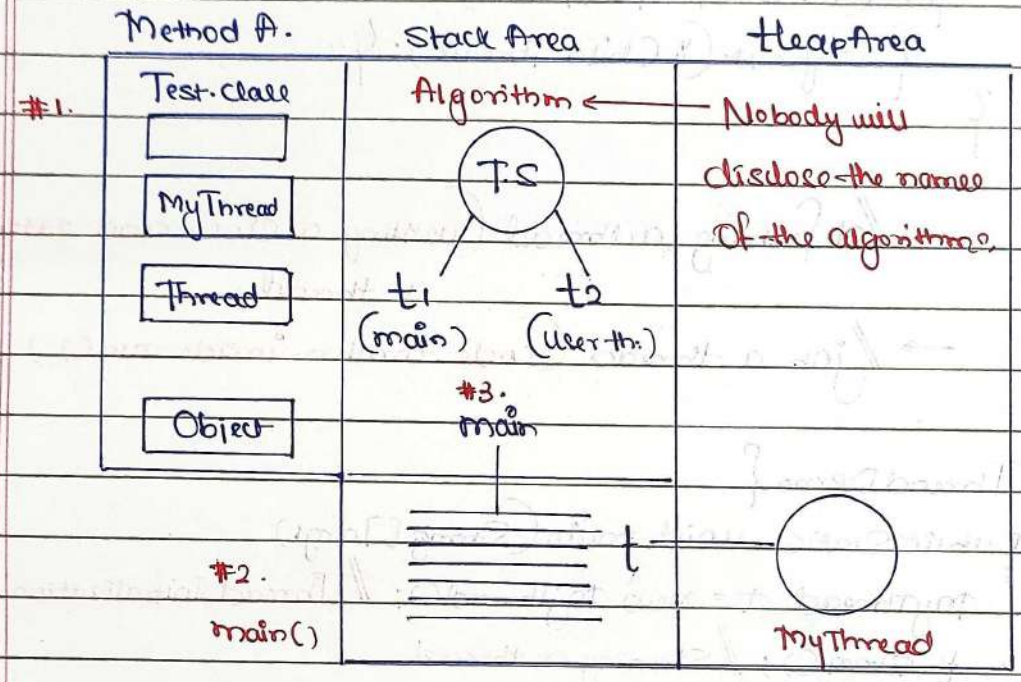
Behind the Scenes:

1. Main thread is created automatically by Jvm
2. Main thread creates child thread and start the child thread

Thread Scheduler:

If multiple threads are waiting to execute, then which thread will execute first is decided by the Thread Scheduler which is a part of Jvm.

* In case of multi-threading we can't predict the exact output. Only possible output we can expect.
* Since jobs of threads are important, we are not intrested in the order of execution it should just execute such that performance should be improved.



Case 2: Difference between t.start() and t.run()

* If we call t.start() and seperate thread will be created which is responsible to execute run() method.
* If we call t.run(), no seperate thread will be created rather the method will be called just like normal method by main thread.
* If we replace t.start() with t.run() then output of program would be:

Child thread
Child thread
main thread
main thread

Case 3:: Importance of Thread class start() method

For everythread, required mandatory activities like registering the thread with ThreadScheduler will be taken care by Thread Class Start() method and programmer is responsible of just doing the job of the Thread inside run() method. Start() acts like an assistance to programmer.

```
public void start()
{   register Thread with ThreadScheduler
    All other mandatory low level activities
    invoke / calling run() method }
```

We can conclude that without executing Thread Class Start() method there is no chance of starting a new Thread in java. Due to this Start() is considered as heart of Multithreading.

Case 4:: If we are not overriding run() method

If we are not overriding run method then Thread class run method will be executed which has empty implement-ation and hence we won't get any output.

eg::

```
Class MyThread extends Thread{ }
Class ThreadDemo{
    public static void main (String[] args)
    {   MyThread t = new Thread();
        t.Start(); }
}
```

It is highly recommended to override run() method, otherwise don't go for multi-threading.

Case 5 :: Overloading of run() method

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

eg::

```
Class MyThread extends Thread{
    public void run()
    { System.out ("no arg method"); }

    public void run (int i)
    { System.out (" zero arg method"); }
}

Class ThreadDemo{
    public static void main (String... arg)
    { MyThread t = new MyThread();
        t.start(); }
}
```

Output : no arg method

Case 6 :: Overriding of Start() method

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

eg::

```
Class MyThread extends Thread{
    public void run(){
        System.out ("no arg method"); }
    public void start()
    { System.out ("start arg method"); }
}
```

```
Class ThreadDemo{
    public Static void main (String... args)
    { MyThread t = new MyThread();
            t.start(); }
}
```

Output : Start arg method.

It is never recommended to override Start() method.

Case 7 ::

```
Class MyThread extends Thread{
    Public void start(){
        Super.Start();
        Sysout("Start method"); }
    }
    public void run(){
        Sysout("run method"); }
    }

Class ThreadDemo{
    public Static void main(String... args)
    { MyThread t = new Mythread();
            t.Start();
        Sysout ("Main method"); }
    }
```

Output : MainThread : main method
Start method
UserDefined Thread : run method

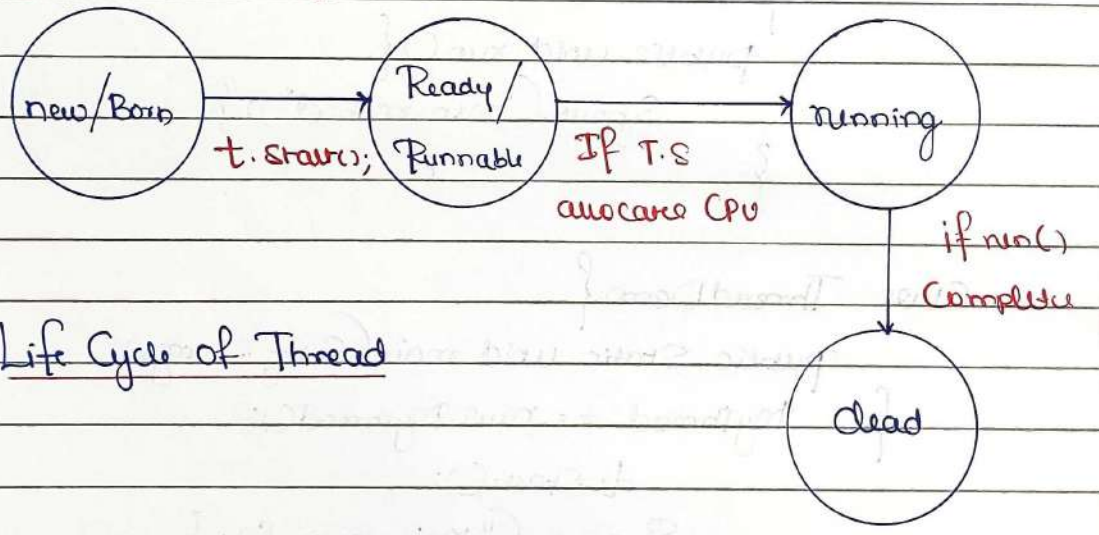
Case 8 :: Life cycle of a thread

```
MyThread t = new MyThread(); // Thread is in born state
    t.start(); // Thread is in ready/runnable state
```

If thread Scheduler allocates CPU time then we say thread entered into running state. If run() is completed by thread then we say thread entered into dead state.

* Once we created a Thread Object then the Thread is said to be in new state or born state.
* Once we call start() method then the Thread will be entered into Ready or Runnable State.
* If the Thread Scheduler allocates CPU then the Thread will be entering/entered into running state.
* Once run() method Complete then the Thread will entered into dead state.

MyThread = new MyThread();



Life Cycle of Thread

Case 9::

After starting the thread, we are not supposed to start the same Thread again, then we say Thread is in "Illegal Thread State Exception".

```
MyThread t = new MyThread(); // Thread is in born state
    t.start(); // Thread is in ready state
    :
    t.start(); // IllegalThreadStateException
```

Creation of Thread using Runnable interface.

1. Creating a thread using java.lang.Thread class.
   a. Use start() from Thread Class.
   b. Override run() and define job of thread.

2. Creation of a Thread requirement to SUNMS is an SES

```
interface Runnable{
        void run();}

Class Thread implements Runnable{ //Adaptor Class
  public void start(){
    1. Register the thread with ThreadScheduler
    2. All other mandatory low level activities (memory
       level)
    3. invoke or call run() method.}

  public void run(){
        job for a thread ; }
}
```

Shortcuts of eclipse :

* Ctrl + Shift + T :- To open a defination of any class
* Ctrl + O :- To list all the methods of a class.

Note :

public java.lang.Thread();

→ thread class start(), followed by thread class run()

public java.lang.Thread (java.lang.Runnable);

→ thread class start(), followed by implementation class of Runnable run()

Defining a Thread by implementing Runnable Interface

```
public interface Runnable {
    public void abstract void main();}

public class Thread implements Runnable {
    public void start(){
      1. register thread with Thread Scheduler
      2. All other mandatory low level activities
      3. Invoke run() }

    public void run(){
        //empty implementation
    }
}
```

eg::

```
Class MyRunnable implements Runnable {
    @Override
    public void run (){
        System.out ("Child thread"); }
}

public class ThreadDemo{
    public static void main (String... args)
{   MyRunnable r = new MyRunnable ();
    Thread t = new Thread (r); // Calle MyRunnable run()
    t.start ();
    System.out ("main thread"); }
}
```

Output: main thread
Child thread

Case Studies:

```
MyRunnable r = new MyRunnable();
Thread t1 = new Thread();
Thread t2 = new Thread(r);
```

Case 1: t1.start();

A new thread will be created, which is responsible for executing Thread class run()

Output: main thread : main thread

Case 2: t2.start();

A new thread will be created, which is responsible for executing MyRunnable run()

Output: main thread : main thread
user defined thread : user thread

Case 3: t1.run();

No new thread will be created, but Thread class run() will be executed just like normal method call.

Output: main thread: main thread

Case 4: t2.run()

No new thread will be created, but MyRunnable class run() will be executed just like normal method call.

Output: main thread : Child thread
main thread.

Case 5: r.start();

It results in Compile Time Error.

Case 6: r.run();

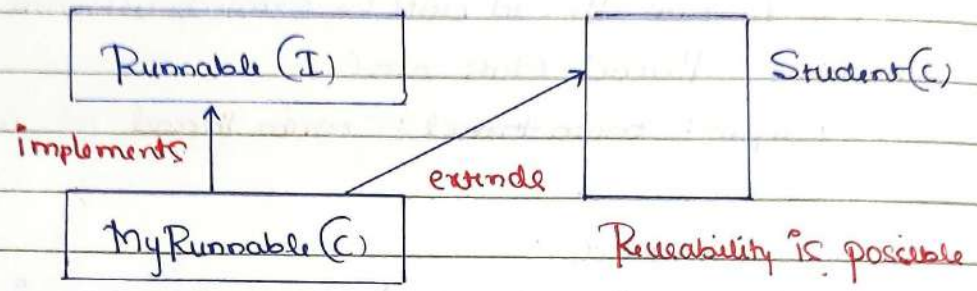
No new thread will be created, MyRunnable class run()

will be executed just like normal method call.

Output: main-thread: Child-thread
main-thread

[Good Approach]

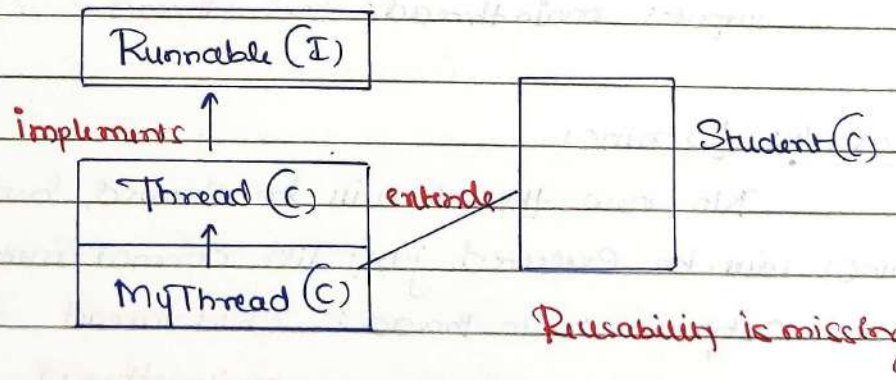
1st Approach: implementing Runnable




```
MyRunnable r = new MyRunnable();
Thread t = new Thread(r);
t.start();
```

[Not Suggested]

2nd Approach: Extending Thread Class




```
MyThread t = new MyThread();
t.start();
```

Q Which approach is the best approach?

Ans * Implements runnable interface is recommended because our class can extend other class through which inheritance benifit can be brought into our class. Internally performance and memory level is also good when we work with interface.

* If we work with extends feature then we will miss out the inheritance benifit because already our class has inherited the feature from "Thread Class", so we normally wont prefer extends approach rather implements approach is used in real time for working with "MultiThreading".

Various Constructors available in Thread Class.

a. Thread ()
b. Thread (Runnable r)
c. Thread (String name)
d. Thread (Runnable r, String name)
e. Thread (ThreadGroup g, String name)
f. Thread (ThreadGroup g, Runnable r)
g. Thread (ThreadGroup g, Runnable r, String name)
h. Thread (ThreadGroup g, Runnable r, String name, long stk size)

Alternate approach to define a thread (not recommended)

```
Class MyThread extends Thread{
    public void run(){
        System.out.("Child thread"); }
}
Class ThreadDemo{
    public static void main(String [] args)
{   MyThread t = new MyThread();
    Thread t1 = new Thread(t);
    t1.start(); System.out("main thread"); } }
```

Output : MainThread : main thread
Child Thread : Child thread

Internally related	Names of the Thread
Runnable ↑ Thread ↑ MyThread	Internally for every thread, there would be a name for the thread. a. name given by jvm b. name given by user.

eg:: Class MyThread extends Thread {
}

```
public Class TestApp{
   public Static void main(String... args)
{  Sysout (Thread. current Thread(). getName());// main

      MyThread t = new MyThread();
      t. Start();
      Sysout (t. getName()); // Thread-0

   Thread. current Name(). SetName("Yash");
   Sysout (Thread. current Thread(). getName());// Yash

   Sysout (10/0); // Exception in thread "Yash" java.lang...
   }
}
```

* It is possible to change the name of the Thread using setName().
* It is possible to get the name of the thread using getName().

eg:

```
class MyThread extends Thread{
    @Override
    public void run(){
    System.out.("run() executed by Thread:"+ Thread.currentThread()
            .getName()); }
}
public class TestApp{
    public static void main(String[]args)
{   MyThread t = new MyThread();
    t.start();
    System.out.("main() executed by Thread :"+ Thread.current
            Thread().getName()); }
}
```

Output: run() executed by Thread : Thread-0
main() executed by Thread : main

Code Snippets :

1.
```
public static void test(String str)
{   int check = 4;
    if (check = str.length)
    System.out.(str.charAt(check--1)+",");
    else System.out.(str.charAt(0)+",");}
```

and the invocation:
test("foxi");
test("tea");
test("to");

Output: Compilation Fails because no boolean value in "if statement".

2.
```
public interface A{ public void m1();}
class B implements A{ } // CE
class C implements A{ public void m1(){ }}
class D implements A{ public void m1(int x){ }} // CE
```

```
abstract class E implements A { }
abstract class F implements A { public void m1() { } }
abstract class G implements A { public void m1(int x) { } }
```

Exactly 2 classes do not compile.

3.
```
Class TestA {
    public void start() { System.out("TestA"); }
}
public Class TestB extends TestA {
    public void start() { System.out("TestB"); }
    public static void main (String [] args)
{   ((TestA) new TestB()). start(); }
}
```

Output: TestB

4.
```
Class Line {
    public class Point { public int x, y; }
    public Point get Point()
    { return new Point(); }
}
Class Triangle {
public Triangle () {
        // insert code here }
}
```

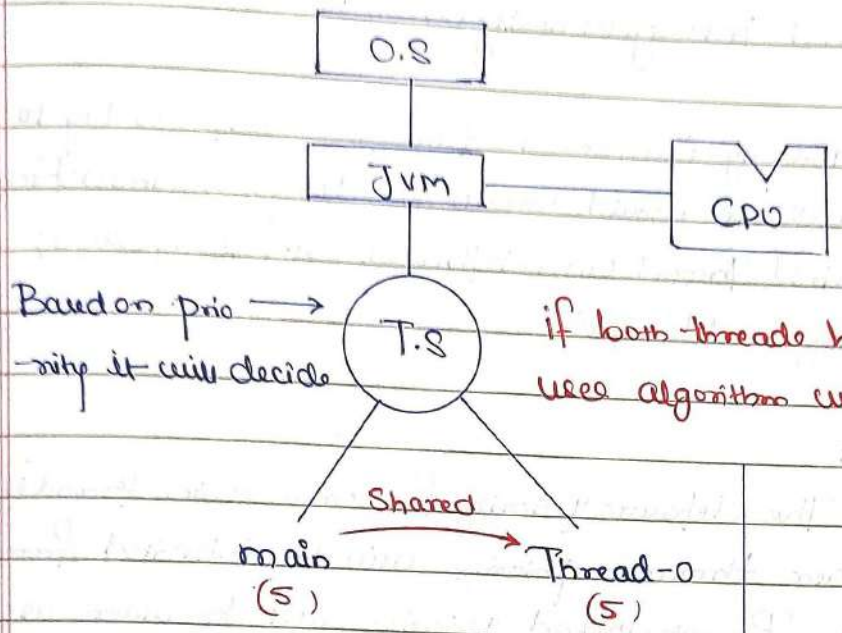
Which code when inserted, retrieves a local instance of a Point Object?

Line.Point p = (new Line()).
get Point()

5.
```
public Class Breaker {
    Static String o = "";
public static void main (String [] args)
{   z:o = 0 + 2;
    for (int x = 3; x < 8; x++) {
        if (x == 4) break;
        o = o + x; } System.out(o); }
```

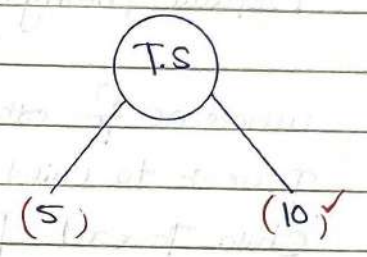
// Compilation fails

Thread Priorities



if both threads have same priority, then TS uses algorithm which is vendor dependent.

JVM will create main thread and it only starts with a default priority "5".



High priority, CPU time will be allocated

For every thread in java has some priority. Valid range of priority is 1-10. If we try to give a different value then it would result in "IllegalArgumentException".

Thread.MIN-PRIORITY=1
Thread.MAX-PRIORITY=10
Thread.NORM-PRORITY=5

Thread class does not have priorities is Thread.LOW-PRIORITY, Thread.HIGH-PRIORITY.

Thread Scheduler allocates time (CPU time) based on "priority".
If both the threads have the same priority then which thread will get a chance as a pgm we can't predict because it is vendor dependent

We can set and get priority values of the thread using the following methods:

a. public final void setPriority (int num)
b. public final int getPriority()

The allowed priority number is from 1-10, if we try to give other values it would result in "Illegal Argument Exception".

```
System.out.print(Thread.currentThread.setPriority(100));
    // Illegal Argument Exception.
```

Default Priority:

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

```
eg:: Class MyThread extends Thread { }
     public Class TestApp{
   public static void main(String[] args)
  { Sysout (Thread.CurrentThread().getPriority());
    Thread.CurrentThread().setPriority(7);
     MyThread t = new MyThread();
    Sysout (Thread.CurrentThread().getPriority()); }
}
```

Reference:

Thread
↑ extends
MyThread

MyThread is created by "main Thread", so priority of main will be shared as a priority for "MyThread".

```
eg:: Class MyThread extends Thread {
     @Override
     public void run()
```

```
{ for (int i=0; i<5; i++)
        System.out.("Child-thread"); }
}
public class TestApp {
  public static void main(String [] args)
  { MyThread t = new MyThread();
    t.SetPriority(7); // line 1
    t.start();
  for (int i=0; i<5; i++)
        System.out.("main thread"); }
}
```

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

Some platforms won't provide proper support for thread priorities.

eg:: windows 7, windows 10..

<u>We can prevent Threads from Execution:</u>

a. yeild()
b. Sleep()
c. join()

<u>yeild()</u> : * It causes to pause current executing Thread for giving chance for waiting threads of same priority.

* If there is no waiting Thread or all waiting threads have low priority then same Thread can continue its execution.
* If all the threads have same priority and if they are waiting then which thread will get a chance we can't expect; it depends on ThreadScheduler.

The Thread which is yilded, when it will get the chance once again depends on the mercy of "Thread Scheduler" and we can't expect exactly.

→ public static native void yild()

```
MyThread t = new MyThread() // new state
    t.start // enter into ready/runnable state
```

* If Thread Scheduler allocate processor then enters into running state.
  - if running class Thread call yild() then it enters into runnable state.
* If run is finished with execution then it enters into dead state.

eg::
```
class MyThread extends Thread{
    @Override
    public void run(){
  for (int i=1; i<=5; i++){
        System.out("Child thread");
        Thread.yild(); } // Line1
    }
}
public class TestApp{
  public static void main(String... args)
{ MyThread t = new MyThread();
    t.start();
  for (int i=1; i<=5; i++)
        System.out("parent thread"); }
}
```

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

Some Platforms won't provide proper support for yield(), because it is getting the execution code from the language preferably from 'C'.

join(): If the thread has to wait until the other thread finishes it's execution then we need to go for join().
If t1 executes t2.join() then t1 should wait till t2 finishes it's execution.
t1 will be entered into waiting state until t2 completes, once t2 completes then t1 will continue its execution

eg:: venue fixing → t1.start()
wedding card printing → t2.start() → t1.join()
wedding card distribution → t3.start() → t2.join()

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

Note: While one thread is in waiting state and if one more thread interrupts then it would result in "InterruptedException". It's checked exception and should be handled.

```
Thread t = new Thread(); // born state
t.start(); // ready state
```

* If T.S. allocates cpu time then Thread enters into running state.
* If currently executing thread invokes t.join() / t.join(1000), t.join(1000,10), then it would enter into waiting state.
* If the thread finishes the execution / time expires / interrupted then it would come back to ready state / runnable state.
* If run() is completed then it would enter into dead state.

eg::
```
Class MyThread extends Thread{
    @Override
    public void run(){
    for(int i=1; i<5; i++){
        Sysout(" Sita thread");
    try{
        Thread.sleep(2000); }
    Catch (InterruptedException e){ }
    }
  }
}

public Class Test3{
  public static void main(String..args) throws
    InterruptedException{
    MyThread t= new MyThread();
    t.Start();
    t.join(10000); // line n1
  for(int i=1; i<5; i++){ Sysout("rama thread"); }
  }
}
```

If the line n1 is commented then we can predict the output because it is the duty of the T.S to assign CPU time.
If line n1 is not commented, then rama thread (main thread) will enter into waiting state. till sita thread (child thread) finishes it's execution.

Output : 2 Threads : a. Child thread
Sita thread
Sita thread
:

b. Main thread
rama thread
rama thread

Waiting of child thread until completing main thread:
We can make main thread to wait for child thread as well as we can make child thread also to wait for main thread.

eg:: Class Mythread extends Thread{
Static Thread mt;
@Override
public void run(){
try{ mt.join() }

Catch (InterruptedException e){ }
for (int i=0; i<10; i++) Sysout ("child thread"); }
}

Public Class Test3{
public Static void main(String[] args) throws InterruptedException
{ MyThread.mt = ~~new~~ MyThread();
Thread.CurrentThread();

```
    MyThread t = new MyThread();
    t.start();
  for(int i=0; i<10; i++){
        System.out("main thread");
        Thread.sleep(2000); }
    }
}
```

Output: Main Thread : main thread
⋮
Child Thread : Child thread
⋮

eg::

```
Class MyThread extends Thread{
  Static Thread mt;

  @Override
  Public void run(){
    try{ mt.join(); }

  Catch (InterruptedException e){ }
  for(int i=0; i<10; i++) System.out("Child thread"); }
}

Public Class Test3{
public static void main(String[]a) throws InterruptedExc
{ MyThread.mt = Thread.CurrentThread();
  MyThread t = new MyThread();
  t.start();
  t.join();
  for (int i=0; i<10; i++) System.out("main thread");
        thread.sleep(2000); }
}  }
```

Output: 2-threads (main, Child thread)

main-thread

⋮

Child thread.

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

eg:: public class Test-3{
public static void main (String...a) throws InterruptedException
{ Thread.currentThread().join(); }
}

Output: Deadlock!, because main thread is waiting for main thread itself.

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

Signature:

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

* Every sleep method throws InterruptedException, which is a checked exception so we should compulsorily handle the exception using try-catch or by throws keyword otherwise it would result in Compile Time Error.

Thread t = new Thread(); // new or born state
t.start(); // ready/runnable state

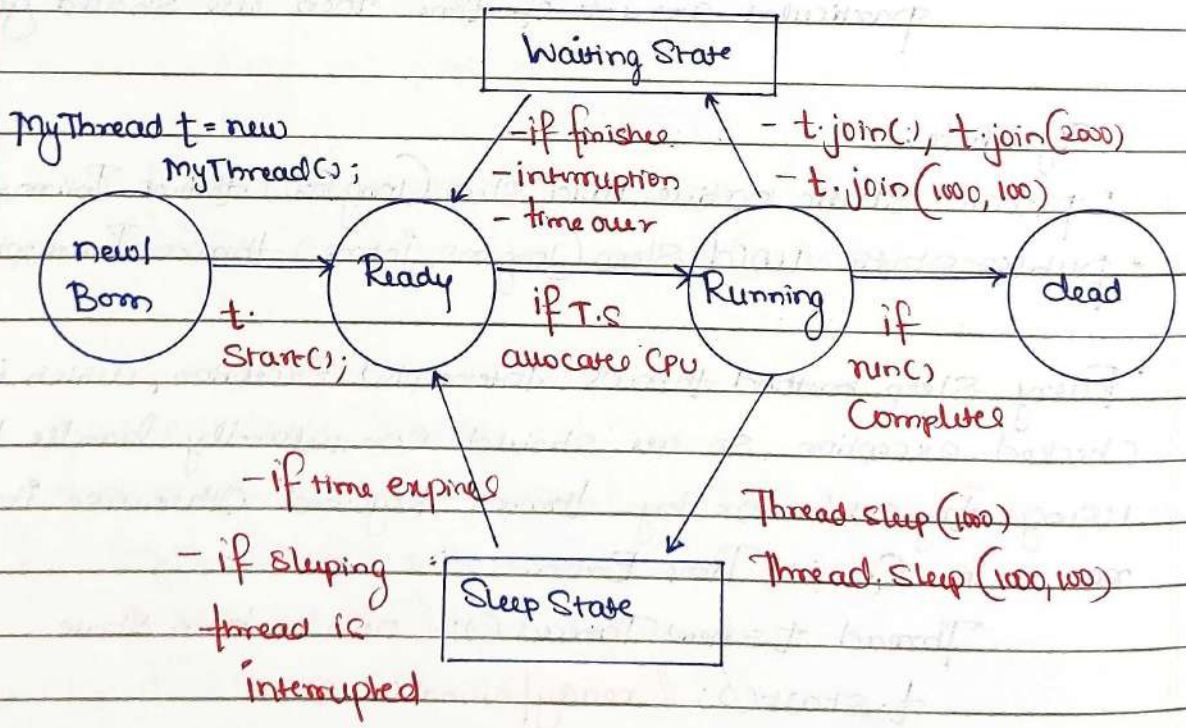
* If T.S allocates CPU time then it would result/enter into running state.
* If run() completes then it would enter into dead state.

* If running thread invokes Sleep(1000) | Sleep(1000, 100) then it would enter into Sleeping State.
* If time expires | if sleeping thread got interrupted then thread would come back to "ready | runnable State".

eg::
```
public class SlideRotator{
    public static void main(String [] args)
        throwe InterruptedException
{   for(int i=0; i<10; i++){
        Syout("Slide:" + i);
        Thread.sleep(5000);}
    }
}
```

Output: Slide: 1
Slide: 2
⋮

Life cycle of Thread



Interrupting a Thread

Public void interrupt()

→ If thread is in sleeping state or in waiting state we can interrupt a thread.

```
eg:: Class MyThread extends Thread{
        @Override
        public void run(){
          try{
              for(int i=0; i<10; i++){
                  System.out("I am a lazy thread");
                  Thread.sleep(2000); }
              }
          Catch (InterruptedException e)
          { System.out("I got interrupted"); }
          }
      }
  public Class Test3{
      public static void main(String [] args) throws InterruptedExcept.
  { MyThread t = new MyThread();
      t.start();
      t.interrupt();
      System.out("End of main"); }
  }
```

Output: main Thread : main thread
Child Thread : I am a lazy thread
I got interrupted

```
eg::
    Class MyThread extends Thread{
      @Override
      public void run(){
```

```
    for(int i=1; i<10; i++) Sysout(" lazy thread");
    }
    Sysout("I am entering into sleeping state");
    try{
        Thread.sleep(2000); }
    Catch (InterruptedException e)
    { ie.printStackTrace(); }

}  }
```

Interrupt from main

```
public class TestApp{
  public static void main(String[]args) throws InterruptedException {
      MyThread t= new Mythread();
      t.start();
      t.interrupt();
      Sysout("main thread"); }
}
```

Interrupt will wait till the Thread enters into waiting state / sleeping state.

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

* Interrupt() can be waiting till target thread enters into waiting state / sleeping state so that this call won't be wasted.
* Once the target thread enters into waiting state / sleeping state then interrupt() will interrupt and it causes the exception.
* Interrupt call will be wasted only if thread does not enter into waiting state / sleeping state.

yeild() join() sleep()

1. Purpose :

* yeild() - To pause current executing Thread for giving the chance of remaining waiting Threads of same Priority.
* join() - If a thread wants to wait until completing some other thread then we should go for join.
* sleep() - If a thread don't want to perform any operation for a particular amount of time then we should go for sleep().

2. Is it static?	3. Is it final?	4. Is it overloaded?
yeild() - yes	yeild() - no	yeild() - no
join() - no	join() - yes	join() - yes
sleep() - yes.	sleep() - no	sleep() - yes

5. Throws IE?	6. Is it native method?
yeild() - no	yeild() - yes
join() - yes	join() - no
sleep() - yes	sleep() :
	- sleep(long ms) - native
	- sleep (long ms, int ns) - non native

Note: Using lambda expression :

```
Runnable r = () -> {
        for(int i=0; i<5; i++)
        { System.out ("Child thread"); }
    };

Thread t = new Thread(r);
t.start();
```

Using anonymous inner class

```
new Thread (new Runnable (){
        @Override
        public void run (){
        for (int i=5; i<10; i++) System.out ("Child-thread");
        }
    }. start ();
```

Synchronization:

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

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

Key points:

* If t1 thread invokes m1() then on the Object x lock will be applied
* If t2 thread invokes m2() then m2() can't be called because lock of x object is with m1.
* If t3 thread invokes m3() then execution will happen because m3() is non Synchronized. Lock concept is applied at the Object level not at method level.

7. Every object in java has a unique lock. Whenever we are using Synchronized Keyword then only lock concept will come into picture.
8. If a thread wants to execute any Synchronized method on the given object, first it has to get the lock of that object. Once the thread got the lock of that object then it will allow to execute any Synchronized method on that object. If the Synchronized method on that object execution completes then automatically thread releases lock.
9. While a thread executing any Synchronized method the remaining threads are not allowed to execute any Synchronized method on that object simultaneously. But remaining threads are allowed to execute any non-Synchronized method simultaneously. [lock concept is implemented based on object but not based on method]

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

Synchronized area: write a code only to perform update, insert, delete.

Non Synchronized area: Write the code only to perform select operation.

```
class ReservationApp{
    checkAvailability(){ // perform read opn }
```

```
Synchronized BookTicket-(){
        //perform update operation }
}
```

eg::
```
Class Display{
    public void wish(String name)
{   for(int i=0; i<10; i++)
    {  Sysout(" Good Morning:" );
       try{
            Thread.Sleep(2000); }
       Catch (InterruptedException e){ }
        Sysout(name); }
}}
```

```
Class MyThread extends Thread{
    Display d;
    String name;
    MyThread (Display d, String Name)
{   this.d = d;
    this.name = name; }
    @Override
    public void run(){
            d.wish (name); }
}
```

```
Public Class Test3{
  public static void main (String... args)
{   Display d = new Display();
    MyThread t1 = new MyThread (d, "Dhoni");
    MyThread t2 = new MyThread (d, "Sachin");
```

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

Output: As noticed below the output is irregular because at a time on a resource called wish() 2 threads are simultaneously.

3 Threads:

Main Thread	Good Morning : Good Morning : ...
Child Thread-1	⋮ ⋮ ⋮
Child Thread-2	⋮ ⋮ ⋮

eg:: 2.

```
Class Display{
    public synchronized void wish (String Name)
{ for (int i=0; i<10; i++)
    { System.out.print(" Good Morning :");
      try{
          Thread.sleep(2000); }
      Catch (InterruptedException e){ }
      Sysout(name); }
    }
}

Class MyThread extends Thread{
    Display d;
    String name;
  MyThread (Display d, String name)
{   this.d=d;
    this.name=name; }
  @Override
    public void run() { d.wish(name); }
}
```

```
public class Test3{
  public static void main (String... args) throws InterruptedExc.
{  Display d= new Display();
   MyThread t1 = new Mythread(d,"dhoni");
   MyThread t2 = new Mythread (d, "yuvi");
      t1. Start();
      t2. Start(); }
}
```

Output: 3 Threads:

a. Main Thread

b. Child Thread-1

Good Morning : dhoni

⋮

c. Child Thread-2

Good Morning : yuvi

Note: As noticed above there are 2 threads which are trying to operate on single object called "display" we need "Synchronization" to resolve the problem of "Data Inconsistency".

Case Study::
```
Display d1 = new Display();
Display d2 = new Display();
MyThread t1 = new Mythread (d1, "yuvraj");
MyThread t2 = new MyThread (d2, "Sachin");
   t1.start();
   t2. start();
```

In the above case we get irregular output, because two different object and since the method is synchronized lock is applied w.r.t object and both the threads will

start simultaneously on different java objects due to which the output is "irregular".

Conclusion:

* If multiple threads are operating on multiple objects then there is no impact of synchronization.
* If multiple threads are operating on same java objects then synchronized concept is applicable.

Class level lock

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

```
eg:: Class X{
        Static synchronized m1(){ } //class level lock
        Static synchronised m2(){ }
          Static m3(){ } //no lock required
        Synchronized m4(){ } // object level lock
        m5(){ } // no lock required.
}
```

t_1 — m1() → Class level lock is applied and chance is given
t_2 — m2() → enters into waiting state

t3 — m3() → gets a chance of execution without any lock
t4 — m4() → Object lock applied and chance is given.
t5 — m5() → gets a chance for execution without any lock.

eg:: 1

```
Class Display {
  Public synchronized void displayNumbers (){
   for (int i=1; i<10; i++)
   {   Sysout (i);
       try{ Thread.Sleep(2000); }

       catch (InterruptedException e){ }
   }
  }
  public Synchronized void displayCharacters (){
   for (int i=65; i<=75; i++)
   {   Sysout ((char) i);
       try { Thread. Sleep (2000); }
       Catch (InterruptedException e) { }
   }
}
}

Class MyThread1 extends Thread {
   Display d;
  MyThread1 (Display d) { this.d=d; }
   @Override
   public void run() { d. displayNumbers (); }
}

Class MyThread2 extends Thread {
   Display d;
  MyThread2 (Display d) { this. d=d; }
```

```
@Override
public void run(){ d.displayCharacters(); }
}

public Class Test3{
  public static void main(String... args)
{ Display d1 = new Display();
  MyThread1 t1 = new MyThread1(d1);
  MyThread2 t2 = new MyThread2(d1);
     t1.start();
     t2.start(); }
}
```

Output: 3 Threads
a. Main Thread
b. UserDefinedThread
displayCharacters()
c. UserDefinedThread
displayNumbers()

Synchronized block

```
Synchronized void m1(){
  :  :  :
  :  :  :
}
```

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

Case Study:

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

```
Synchronized (this){
    . . . .
    . . . .
}
```

b) To get a lock of particular object :: B.

```
Synchronized (B){
    . . . .
    . . . .
}
```

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

c) To get class level lock we have to declare Synchronized block as follows:

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

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

Synchronized block (continued)

eg:: 1

```
Class Display{
    public void wish (String name){
        ; ; ; ; // many lines
        Synchronized (this){
            for (int i=0; i<5; i++)
            {   System.out ("Good Morning");
                try{ Thread.sleep (2000); }
            Catch (InterruptedException e){ }
```

```
        System.out(name); }
    } ;;;;; //many lines of code
  }
}

Class MyThread extends Thread {
    Display d;
    String name;
  MyThread (Display d, String name)
  { this.d=d;
    this.name=name; }
  Public void run() { d.wish(name); }
}

public static void main (String... args)
{   Display d = new Display();
    MyThread t1 = new MyThread (d, "dhoni");
    MyThread t2 = new MyThread (d, "yuvi");
      t1.start();
      t2.start(); }
```

Output:
Good Morning yuvi
Good Morning yuvi
⋮ ⋮ ⋮
Good Morning dhoni
Good Morning dhoni
⋮ ⋮ ⋮

eg:: 2 (for the above code)

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

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

eg:: 3 (for same code in display class)

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

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

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

eg:: int x = 10;

```
Synchronized (x){...   // CE: Unexpected type.
...}
```

Inter Thread Communication (remember postbox example)

Two threads can communicate with each other with the help of

a. notify()
b. notifyAll()
c. wait()

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

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

* If a thread wants to call wait(), notify() / notifyAll() then compulsorily the thread should be the owner of the object otherwise it would result in "IllegalMonitorStateException".
* We say thread to be the owner of that object if thread has lock of that object.
* It means these methods are part of synchronized block or method, if we try to use outside synchronized area then it would result in RuntimeException called "IllegalMonitorStateException".
* If a thread calls wait on any object, then first it immediately releases the lock on that object and it enters into waiting state.
* If a thread calls notify() on any object, then he may or may not release the lock on that object immediately.
* Except wait(), notify(), notifyAll(), lock can't be released by other methods.

Note:

* yeild(), Sleep(), join() - can't release the lock
* wait(), notify(), notifyAll() - will release the lock, otherwise Interthread Communication can't happen.

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

Method prototypes:

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

4. public final native void notify()
5. public final void notifyAll()

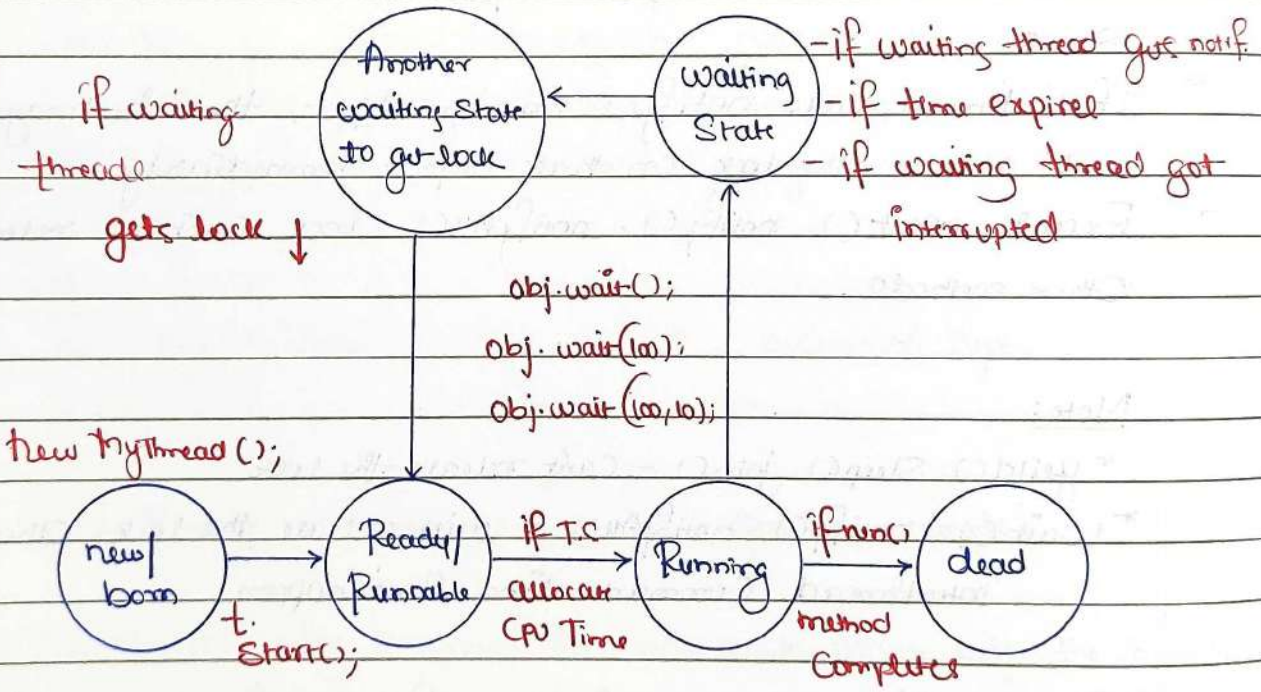
Interview Question.

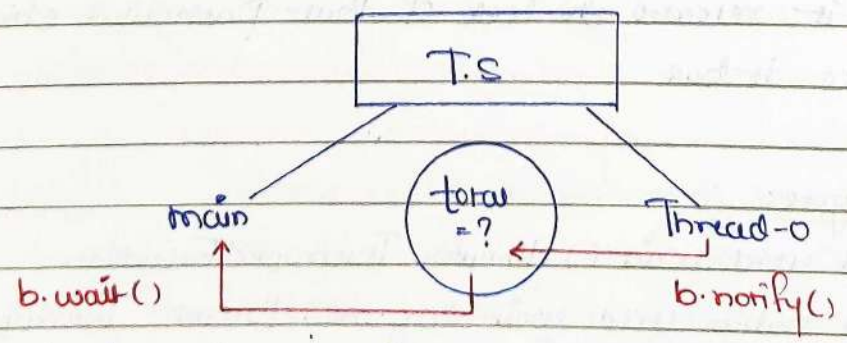
Q1. Methods like wait(), notify(), notifyAll() are present inside Object Class, why not in Thread Class?

Ans. Thread will call wait(), notify(), notifyAll() on Objects like PostBox, Stack, Customer...

→ Obj.wait(), Obj.notify(), Obj.notifyAll()

These methods should be available for every object in java, if method has to be available for every object in java then those methods should come from "Object" class.





main thread in waiting State

Sending notification to waiting thread.

* main thread got the notification, waited for the lock to be released from the other thread, once the lock is released it will use ThreadB total variable

eg::
```
Class ThreadB extends Thread{
    int total=0;
    @Override
    public void run(){
      for (int i=0; i<=100; i++)
      { total += i; }
    }
}
```

```
public static void main (String ... args)
{   ThreadB b= new ThreadB();
    b.Start();
    // Statement
    System.out(b.total); }
```

A> If we replace the "Statement" with Thread.Sleep(10,000) then thread will enter into waiting state but within 10s the updation value is ready. Within 10s if the updation is not ready, then we should not use Thread.Sleep(10000);

B> If we replace with b.join(), then main thread will enter into waiting state, then child will execute for loop, till then main thread has to wait.

main thread is waiting for updation result

```
for (int i=0; i<100; i++) { total += i; }
        // 1Cr lines of code is available
```

main thread has to wait till 1Cr lines of code, why main thread should wait for the completion of the code.

eg:: Class ThreadB extends Thread{

int total =0;

@Override

public void run(){

Synchronized (this) {

Sysout ("child thread started");

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

Sysout ("child thread giving notification");

this.notify(); }

}

}

("different class")

public static void main (String[] args) throws InterruptedException

{ ThreadB b= new ThreadB();

b.start();

Thread.sleep (10,000);

Synchronized (b){

Sysout ("main thread is calling wait on object B");

b.wait();

Sysout ("Main thread got notification");

Sysout (b.total); }

}

Output: Child thread started calculation

Child thread trying to give notification

Main thread is calling wait on object B

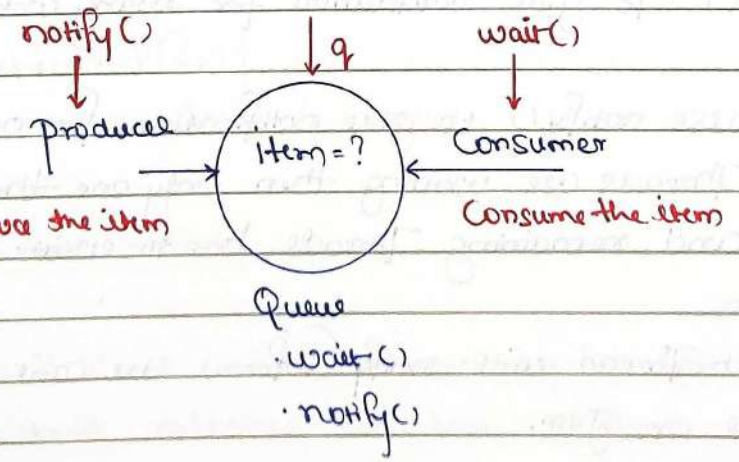
because of Thread.sleep(1000) main thread will never get notification.

eg:: (for same code)

```
public static void main (String [] args) throws InterruptedException
{   ThreadB b = new ThreadB();
    b.start();

    Synchronized (b){
     System.out ("main thread calling wait on B object");
     b.wait (10000);
     System.out ("main thread got notification");
     System.out (b.total); }
}
```

<u>Output</u>: Child thread Started Calculation
Child thread trying to give notification
Main thread is waiting on B Object
Main thread got notification
5050

<u>Producer and Consumer Problem</u>:



Producer: produce the item and update in the queue.
Consumer: Consume the item from the queue.

```
Class Producer extends Thread{
    Producer (){
      Synchronized (q){
      //produce item and update to queue
       q.notify(); }
    }
}

Class Consumer extends Thread{
    Consumer (){
      Synchronized (q){
      if (q isEmpty ){q.wait();}
      else
        //consume item from the queue }
    }
}
```

<u>Difference between notify() and notifyAll()</u>

* notify() → To give notification for one waiting thread.
* notifyAll() → To give notification for many waiting thread.

→ We can use notify() to give notification for only one thread. If multiple Threads are waiting then only one thread will get a chance and remaining Threads has to wait for further notification

But which Thread will notify (inform) we can't expect because it depends on JVM.

<u>Waiting State</u>

	Obj1.wait(); // Go threads waiting.
Obj1.notify()	Obj1.notify();

<u>Running State</u>

Among 60 threads which thread will get a chance we don't have control over that, it is decided by Jvm (Thread Scheduler).

→ We can use notifyAll() method to give notification for all waiting threads of particular object.
All waiting threads will be notified and will be executed one by one, because they required lock.

Waiting State		Note: On which object we are calling wait(), notify(), and notifyAll() that corresp. object lock we have to get but not other object locks.
	Obj1.wait();	
Obj1.notifyAll()	Obj2.wait();	
Running State		

eg:: Stack S1 = new Stack();
Stack S2 = new Stack();
synchronized (S1){
S2.wait(); } // RE: Illegal Monitor State Exception

synchronized (S2){
S2.wait(); } // valid.

Questions based on lock

1. If a thread call wait() immediately it will enter into waiting state without releasing any lock → False

2. If a thread call wait() it releases the lock of that object but may not immediately → False.

3. If a thread call wait() on any object, it releases all locks

acquired by that thread and enters into waiting State → False.

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

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

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

Deadlock

* If two Threads are waiting for each other forever (without end), such type of Situation (infinite waiting) is called dead lock.
* There is no resolution technique for deadlock but Several Prevention (avoidance) technique are possible.
* Synchronized Keyword is the cause for deadlock hence whenever we are using synchronized keyword we have to take special care.

```
eg:: Class A{
        public void d1(B b){
        try{ Thread.sleep(5000); }
        Catch (InterruptedException e){ }
        Sysout("Thread-1 trying to call b.last");
          b.last(); }
     public void last(){
          Sysout("Inside A last()"); }
     }
     Class B{
        public void d2(A a){
          Sysout("Thread-2 Starts execution of d2");
```

```
    try{ Thread.sleep(5000); }
    Catch (InterruptedException e) { }
    Sysout ("Thread-2 trying to call A last");
        a.last(); }
    public void last() { Sysout ("Inside B last() method"); }
}

public Class Test extends Thread{
    A a = new A();
    B b = new B();
    public void m1(){ this.start();
            a.d1(b); } // executed by main thread

    public void run(){ b.d2(a); } // executed by child thread
    public Static void main (String... args)
{   Test t = new Test();
            t.m1(); }
}
```

Since methods are not Synchronized, lock is not required, so no deadlock.

<u>Output</u>: Thread-1 Starts execution of d1()
Thread-2 Starts execution of d2()
Thread-1 trying to call B last()
Inside B last() method
Thread-2 trying to call A last()
Inside A last() method

eg:: 2 (using Synchronized Keyword for every method in A, B)

```
    Public Static void main (String... args)
{   Test t = new Test();
        t.m1(); } // main thread executing
```

(Rest all code Same as above)

In the above program, there is a possibility of "deadlock".

Output: Thread-1 starts execution of d1()
Thread-2 starts execution of d2()
Thread-1 trying to call B last()
Thread-2 trying to call A last()
// here cursor will be waiting.

→ t1 → Starts d1(), Since d1() is synchronized and a part of 'A' class so t1 applies lock of (A) and starts the execution, while executing it encounters Thread.sleep(). so TS gives chance for t2 thread.
After getting a chance again by TS, it tries to execute b.last but after lock of b is with t2 thread, so t1 enters into waiting state.

→ t2 → Starts d2(), Since d2() is synchronized and a part of 'B' class to t2 applies lock of (B) and starts the execution, while executing it encounter Thread.Sleep(), So TS give chance again for t1 thread.

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

Note: Synchronized is the only reason why there is a deadlock, so we should be careful when we use synchronized keyword, if we remove atleast one synchronized word then the program would enter into "deadlock".

Deadlock vs Starvation

Long waiting of a thread, where waiting never ends is termed "deadlock".

Long waiting of a thread, where waiting ends at a certain point is called "Starvation".

eg:: Assume we have 1cr threads, where all 1cr threads have priority of 10, but one thread is of priority 1 and it has to wait for long time, this scenario is called "Starvation".

Note: Low priority thread has to wait until completing all priority threads but ends at certain point which is nothing but Starvation.

Daemon Threads: The thread which is executing in background is called "Daemon Thread". eg: AttachListener, SignalDispatcher ...

Main Objective of Daemon Thread:

* To provide support for Non-Daemon threads (main thread).

eg:: if main thread runs with low memory then jvm will call Garbage Collector thread to destroy the useless objects, so that no. of bytes of free memory will be improved, with this free memory main thread can continue it's execution

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

Jvm → Creates 2 threads

a. Daemon thread (priority 1, priority 10)

b. main (priority = 5)

While executing the main code, if there is a shortage of memory then immediately Jvm will change the priority of Daemon thread to 10, so Garbage Collector activates Daemon

thread and it frees the memory after doing it immediately it changes the priority to 1, so main thread will continue.

Q1 How to check weather the Thread is Daemon or not?

Ans * public boolean isDaemon() → To check weather the thread is Daemon.

* public boolean setDaemon(boolean b) throws IllegalThreadStateException

b → true, means the thread will become Daemon, before starting the thread we need to make the thread as "Daemon" otherwise it would result in "IllegalThreadStateExcept."

Q2 What is the default nature of the Thread?

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

Q3 It is possible to change the NonDaemon nature of main thread?

Ans No, because the main thread's starting is not in our handle, it will be started by JVM.

```
eg:: class MyThread extends Thread { }
     public class Test{
  public static void main(String... args){
    System.(Thread.currentThread().isDaemon());// false
  Thread.currentThread().setDaemon(true);// RE

      MyThread t = new MyThread();
        System.(t.isDaemon());// false
        t.setDaemon(true);
```

```
    t.start();
    System.out.(t.isDaemon); } // true
}
```

Note: Whenever last daemon threads terminate, automatically all daemon threads will be terminated irrespective of their position.

eg:: - makeup man in Shooting is a daemon thread
- hero is main thread
- if hero role is over, then automatically the makeup role is also over.

```
eg:: Class MyThread extends Thread{
        public void run(){
            for(int i=1; i<=10; i++){
                System.out.("child thread");
            try{ Thread.sleep(2000); }
        Catch (InterruptedException e){ System.out.(e); }
        }
    }
    public Class Test{
    public static void main(String...arg(){
        MyThread t = new MyThread();
        t.setDaemon(true); // Stmt-1
        t.start(); System.out.("end of main"); }
    }
```

Output: if we comment Stmt-1, both threads are non daemon threads, so it would continue execut.

end of main-thread
Child thread
⋮ ⋮

Output: In the above code, main thread is non daemon and user defined thread is daemon, if main thread finishes execution, automatically daemon thread will also finish the execution.

Date 8/3/23

Collections in Java

The main 7 classes in collections are

1. ArrayList
2. LinkedList
3. ArrayDeque
4. PriorityQueue
5. TreeSet
6. HashSet
7. LinkedHashSet

Divided into 3 major interfaces:
1. List
2. Queue
3. Set

1 ArrayList → List(I)

* Internally dynamic array data structure
* Size of the arraylist grows automatically
* Heterogenous data elements can be stored with homogenous.
* Index based accessing allowed
* We are able to add elements at any given index

Syntax:

```
ArrayList al1 = new ArrayList();
al1.add(30);
al1.add(20);
Sysout(al1); // [30, 20]
```

```
ArrayList al2 = new ArrayList();
al2.add("vivek");
al2.add(45.9);
Sysout(al2); // ["vivek", 45.9]

ArrayList al3 = new ArrayList();
al3.add(al2); // Can add one complete arraylist.
```

```
al3.add(55);
// adds at rear
al3.add(0,7);
// adds at first
al3.add(7,9);
// adds at 7th index
```

But adding at index is inefficient.

(2) LinkedList → List(I) and Deque(I)

* Doubly linked list data structure.
* Homogenous data can be stored, duplicate are allowed.
* Heterogenous data can be stored.
* Data is stored as an object, index based accessing allowed.

Syntax:
```
LinkedList ll1 = new LinkedList();
ll1.add(50);
ll1.add("ineuron");
Sysout(ll1); // [50, ineuron]

ll1.addFirst("Hyder"); // adds at first
ll1.add(3, 99); // adding at index 3
ll1.addLast("Bangalore"); // add at last
```

Arraylist and LinkedList have almost same methods but LinkedList has some extra methods and adding at particular index is very fast compared to Arraylist because of doubly ll. data structure

Q1. When to use array over Arraylist?

Ans Whenever the size of the data is known and you are sure that the data is homogenous then you must go with array. Array is faster than arraylist because in Arraylist data is stored as an object which is little time consuming, for primitive data types also get converted to object, but array store as it is.

Other methods in Arraylist

* al4.get(5); // used to fetch element from position 5.
* al4.contains(44); // returns true if 44 is available
* al4.indexOf(22); // returns index of 22
* al4.isEmpty(); // Checks if empty
* al4.size(); // returns number of elements
* al4.trimToSize(); // Clear the extra space
* al4.clear() // erases all data from arraylist.

(and many more are available)

Other methods in LinkedList

* list.clear(); // removes all element from list.
* list.getFirst(); // gives you first item
* list.getLast(); // gives you the last item
* list.indexOf(40); // returns index of 40
* list.lastIndexOf(40); // returns last index of 40
* list.addFirst(50); // adds element at first
* list.addLast(90); // adds element at last
* list.peekFirst(); // returns first item
* list.pollFirst(); // gives first item, and item is removed from list.

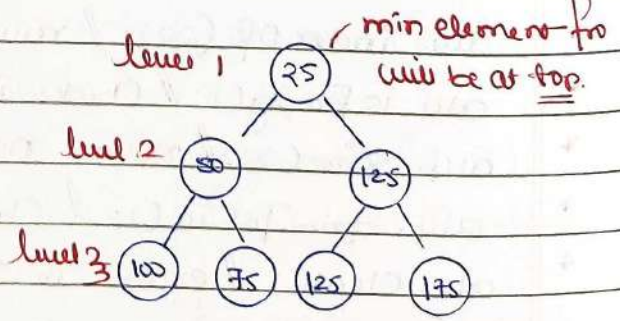
(and many more available)

(3) ArrayDeque → Deque (I)

* Double ended queue data structure.
* Index based accessing is not allowed.
* Insertion and deletion at front and back are allowed but not in middle
* Duplicates allowed.

Syntax:
```
ArrayDeque ad = new ArrayDeque();
ad.add(10); // adds element
ad.add(10);
ad.addFirst(30); // adds at first
ad.addLast(70); // adds at last.
```

(4) PriorityQueue → Queue (I)

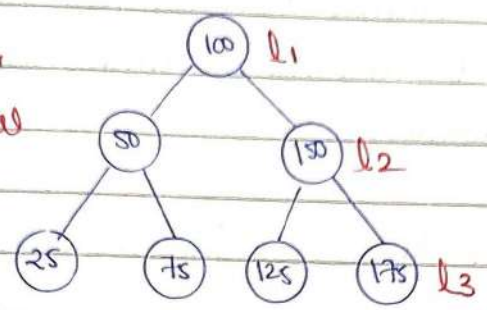


* Min heap data structure
* Indexing is not allowed
* Duplicates are allowed
* Element will be stored in form of min heap data structure - higher priority element will be at the top.

(5) TreeSet → Set(I)

* Binary search tree data structure.
  {100, 50, 150, 25, 75, 125, 175}

in-order traversal for Search



* duplicate not allowed.
* if an element is less than root node goes to left, if higher than right.
* Searching for key(200) takes 3 comparisions in this example. ie. O(log n) time complexity.

* When the data is in sorted order, it forms a "squeud tree" and searching takes 'n' comparisions i.e O(n) TC

Syntax:
```
TreeSet ts = new TreeSet();
    ts.add(100); //adds element
    ts.add(75);
    ts.add(50);
ts.ceiling(50); // returns 50 if available, or the next higher value
ts.higher(50); // returns next higher element than 50.
ts.floor(40); // returns 40 if available, or next lower element
ts.floor(40); //  "  "  "  "  "  "
ts.lower(40); // returns the first lower element than 40
```

(6) HashSet → Set(I)

* It internally has a hash function (hashing algorithm) and associated HashTable with load factor (75%)
* Duplicate values are not allowed
* On an average the contains() of HashSet runs in O(1) time.
* Order of insertion is not maintained

Syntax:
```
HashSet hs = new HashSet();
    hs.add(100);
```

7) LinkedHashSet

* It is a sub class (child class) of HashSet.
* It also follows hashing algorithm behind the scene to store data because of which searching becomes very fast
* The order of insertion is maintained unlike HashSet.
* Duplicates are not allowed.

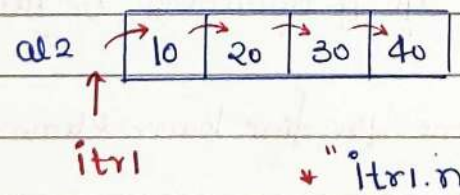
Syntax :
```
LinkedHashSet lhs = new LinkedHashSet();
lhs.add(100);
lhs.add(50);
```

→ Different ways to access data from collection

1>
```
for (Object obj : al2)
{ Sysout(obj +" "); }
```
// only for classes that allow index based accessing. (normal for).
// for-each is for all classes.

* Iterator itr1 = al2.iterator();

// Iterator is common to all classes in collection
iterator points to the first element (just before it)



* "itr1.next()" gives the next object value from the array list.

2>
```
while (itr1.hasNext())
{ Sysout (itr1.next()); }
```
// hasNext() returns boolean value if there is an item next.

* List Iterator litr = al2.listIterator(al2.size());

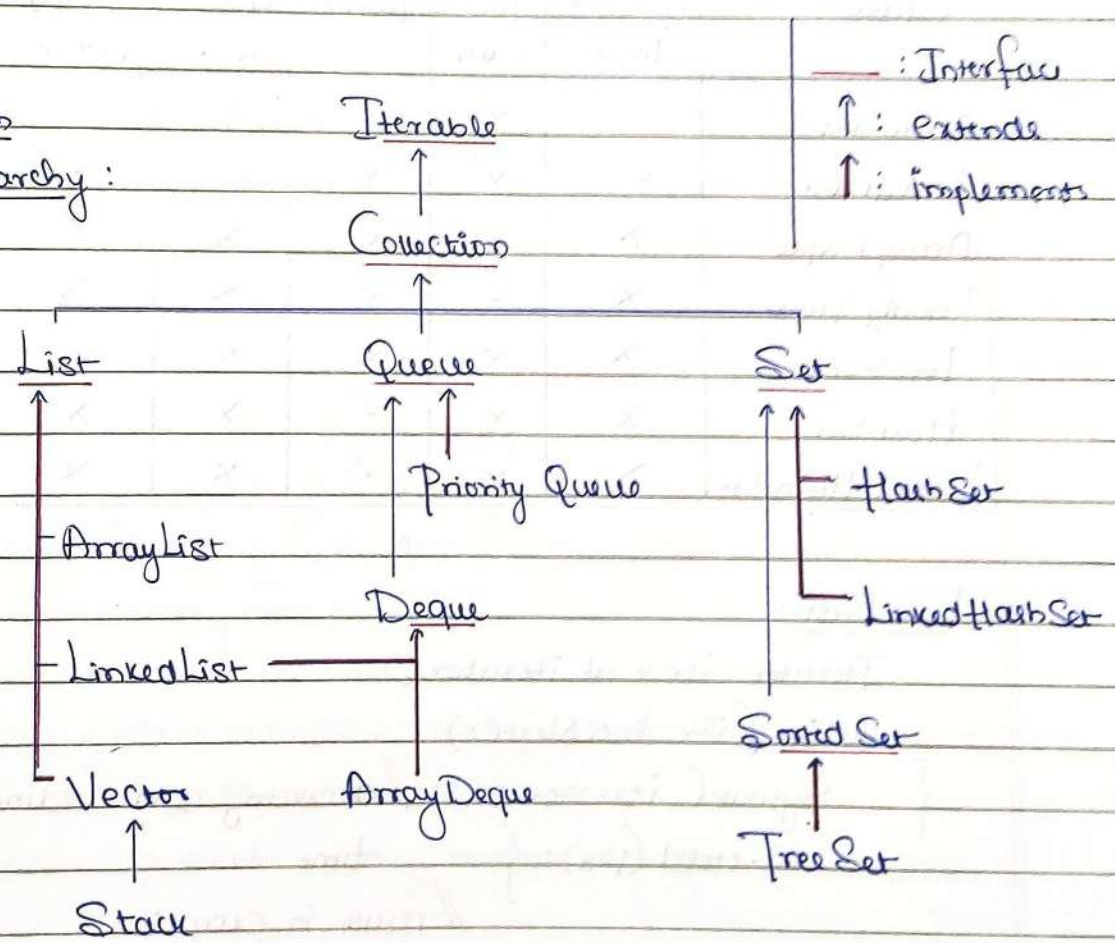
// List Iterator is applicable only for ArrayList and LinkedList

3> while (litr.hasPrevious())
{ System.out.(litr.previous()); } // traverse in reverse direction using list iterator.

* Iterator dtr = list.descendingIterator();
// descending iterator is only in LinkedList, ArrayDeque, TreeSet.
// points to last item and when next() is called goes to previous next (next in reverse order)

4> while (dtr.hasNext()); // traversing in reverse order
{ System.out(dtr.next() + " "); }

Collection Hierarchy:



Summary

Class	Internal DS	Preserve order of insert.	Allow null value.	Duplicate allowed
ArrayList	Dynamic array	✓	✓	✓
LinkedList	Doubly linked list	✓	✓	✓
Array Deque	Double ended queue	✓	✗	✓
Priority Queue	min-heap ds	✗	✗	✓
Tree Set	binary search tree	✗	✗	✗
HashSet	hash table	✗	✓	✗
LinkedHashSet	hash table	✓	✓	✗

Class	for loop()	for each	Iterator	List Iterator	descending iterator	Enumeration
ArrayList	✓	✓	✓	✓	✗	✗
LinkedList	✓	✓	✓	✓	✓	✗
Array Deque	✗	✓	✓	✗	✓	✗
Priority Queue	✗	✓	✓	✗	✗	✗
Tree Set	✗	✓	✓	✗	✓	✗
HashSet	✗	✓	✓	✗	✗	✗
LinkedHashSet	✗	✓	✓	✗	✗	✗

Fail Fast:

```
Iterator itr = al.iterator();
while (itr.hasNext())
{   System.out.(itr.next());   // accessing and adding at the same
    al.add(123);}              time
                               // results in exception
```

Fail Safe:

```
CopyOnWriteArrayList Cal = new CopyOnWriteArrayList();
Cal.add(1000);
```

```
cal.add(2000);
cal.add(3000);
cal.add(4000);
Iterator itr = cal.iterator();
while (itr.hasNext())
{   System.out.(itr.next());  // no exception, fail safe
    cal.add(12345); }
```

* Fail fast means whenever we are attempting concurrent or structural modification while accessing the data then it results in exception and fail fast.
* If you are attempting concurrent modification, your modification should fail without exception then the concept of fail safe comes into picture.
* You can perform fail safe on classes which are available in concurrent package (subpackage of util).

Inbuilt methods in Collections.

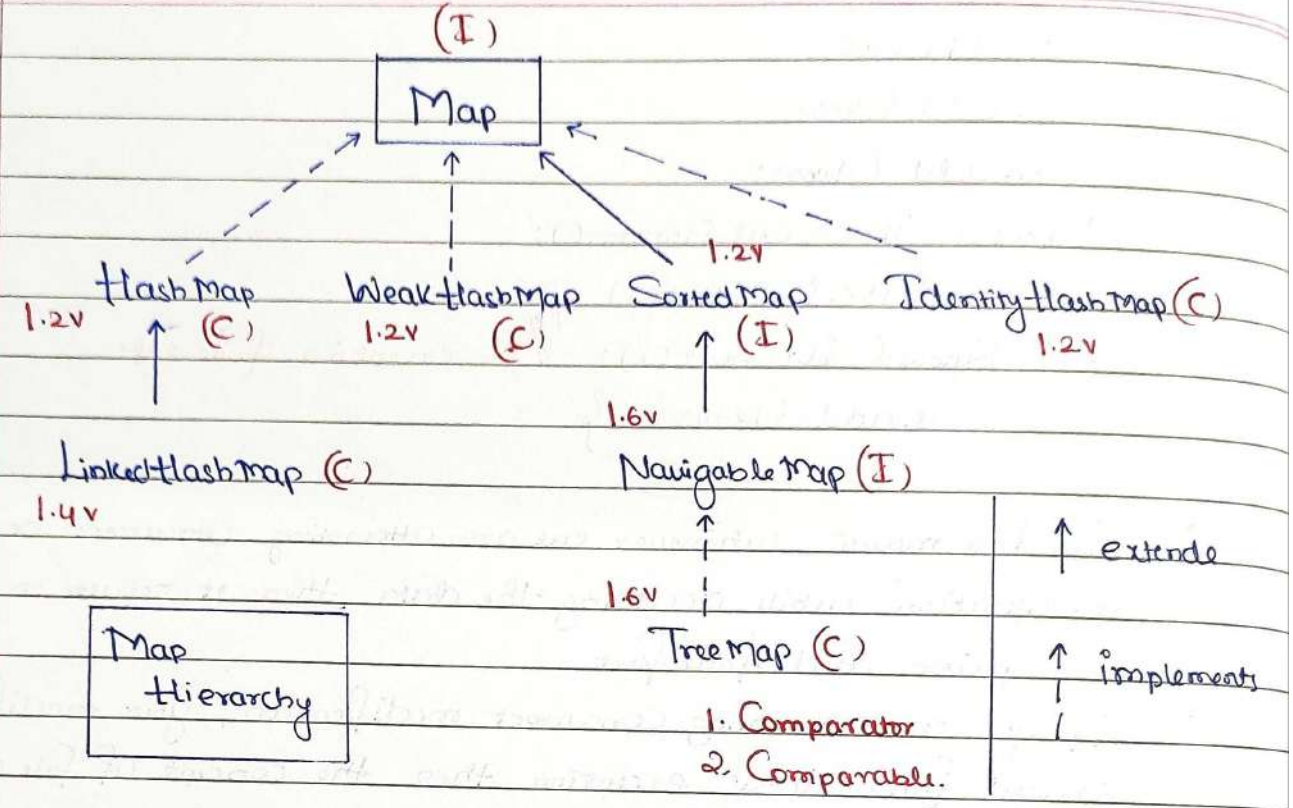
1. Collections.sort(cal); // sorts only if data is homogenous
2. Collections.binarySearch(cal, 40); // returns index of 40 in array list
3. Collections.frequency(cal, 40); // returns frequency of 40 in cal

(and many more).

* All classes under collection API/framework has it's own speciality (features, internal data structures).
* On objects present using method we perform actions.

Map in Java

The map interface is present in java.util package represents a mapping between a key and a value. The map interface is not a subtype of Collection Interface.



* Maps are perfect to use for key-value association mapping, such as dictionaries.
* Since Map is an interface, Objects cannot be created of type map. We always need a class that extends this map.
* A map cannot contain duplicate keys and each key can map to atmost one value. Some implementations allow null key and null value like the HashMap, LinkedHashMap but some do not like the TreeMap.
* The order of map depend upon implementations, TreeMap and LinkedHashMap have predictable orders, while HashMap does not.
* There are two interfaces for implementing map in java. They are Map and SortedMap, and three classes : HashMap, TreeMap and LinkedHashMap.

<u>Syntax</u> :
```
HashMap hm = new HashMap();
hm.put(10, "Sachin");
hm.put(7, "MSD");
```

```
hm.put(18, "Kohli");
System.out.println(hm); // 18=Kohli, 7=MSD, 10=Sachin
                        // Order of insertion not maintained

LinkedHashMap lhm = new LinkedHashMap();
lhm.put(10, "Sachin");
lhm.put(7, "MSD");
lhm.put(18, "Kohli");
Sysout(lhm); // { 10=sachin, 7=MSD, 18=Kohli }
             // order of insertion is maintained

Collection c = hm.values();
Iterator itr = c.iterator();
while (itr.hasNext())
{ System.out.println(itr.next()); } // Kohli MSD Sachin

  Sysout(hm.get(10)); // Sachin

Set s = hm.keySet();
Iterator itr2 = s.iterator();
while (itr2.hasNext())
{ System.out.println(itr2.next()); } // 18 7 10

Set es = hm.entrySet();
Iterator itr3 = es.iterator();          // 18=Kohli
while (itr3.hasNext())                  7=MSD
{ System.out.println(itr3.next()); }    10=Sachin

while (itr3.hasNext())
{ Map.Entry data = (Entry) itr3.next();          // getting
  Sysout(data.getKey()+":"+data.getValue()); }   entries
```

classmate
Date 9/3/23
Page

Passport code to show Map functions:

```
import java.util.*;
import java.util.Map.Entry;
Class Information
{   private String name;
    private int age;
    private String fatherName;
    private String city;
  Public Information(String name, int age, String fatherName,
                String city)
  {  this.name = name;
     this.age = age;
     this.fatherName = fatherName;
     this.City = City; }

Public String getName() { return name; }

public int getAge() { return age; }

Public String getFatherName() { return fatherName; }

public String getCity() { return city; }

  @Override
  Public String toString()
{  return name+" "+age+" "+fatherName+" "city; }

}   // System.out.print Call toString() method internally
    so when we override it we get the desired
    Output instead of hashcode.
```

```
public class LaunchPassport
{
  public static void main (String[] args)
{   Information info1 = new Information ("Rohan", 18, "Sharmaji",
                                "Delhi");
Information info2 = new Information ("Nitin", 27, "M", "Bengaluru");
Information info3 = new Information ("Hyder", 28, "H", "Bengaluru");

    HashMap hm = new HashMap();
     hm.put (000, info1);
     hm.put (111, info2);
     hm.put (333, info3);

  Set set = hm.entrySet();
  Iterator itr = set.iterator();
  while (itr.hasNext())
  {  Map.Entry passport = (Entry) itr.next();
     System.out ("Passport Number :" + passport.getKey() + "Info :"
     + passport.getValue() ); }
    }
}
```

Output:

Passport Number : 0 : Info : Rohan 18 Sharmaji Delhi
Passport Number : 333 : Info Hyder 28 H Bengaluru
Passport Number : 111 : Info Nitin 27 M Bengaluru.

	HashMap	LinkedHashMap
1.	Came in 1.2v	Came in 1.4v
2.	DSA : HashTable	DSA: HashTable + LinkedList
3.	Order of insertion not preserved	Order of insertion maintained
4.	Parent of LinkedHashMap	Sub Class of HashMap

eg1.

```
import java.util.*;
Class Employee
{
    Private int ~~get~~ empid;
    Private String name;
    Private String empAddr;
  @Override
    public String toString(){ return "hyder"; }

  @Override
    public void finalize(){
      Sysout("Garbage Collector Collected the object"); }
  }

public Class LaunchGC
{ public Static void main(String[] args) throws Exception
  {
    Employee e = new Employee();

      e = null; // eligible for Garbage Collection
    System.gc(); // Call for garbage collector
      // internally call finalize method
  }
}
```

Output: Garbage Collector Collected the object

eg2.

```
public Static void main(String[] args)
{   Employee e = new Employee();
    HashMap hm = new HashMap();
    hm.put(e, "Hyder");
  e = null; // eligible for garbage Collection
  System.gc(); // Call to GC but won't execute finalize
                because Hashmap dominates Garbage Collector.
}
```

eg3.
```
public static void main(String[] args)
{   Employee e = new Employee();
    WeakHashMap hm = new WeakHashMap();
    hm.put(e, "Hyder");
    e = null; // eligible for gc
    System.gc(); // Call for GC, Since GC "dominates WeakHashMap"
                    we get the output from finalize block
}
```
* This is the only diff b/w HashMap and WeakHashMap

HashTable Syntax:
```
HashTable ht = new HashTable();
ht.put(11, "Hyder");
ht.put(12, "Nitin");
System.out(ht); // { 11 = Hyder , 12 = Nitin }
```

	HashMap	HashTable
1.	All methods are not Synchronized	All methods are Synchronized
2.	At a time multiple threads can Operate on object. So not ThreadSafe.	At a time only one thread can operate on an object, So it is ThreadSafe.
3.	Performance is high	Performance is low
4.	Null is allowed for both keys and values (key's only once)	Null is not allowed for both. results in NullPointerException.
5.	Introduced in 1.2V	Introduced in 1.0V

TreeMap Syntax:
```
TreeMap tm = new TreeMap();
tm.put(14, "Rohan");
tm.put(12, "Vivek");
System.out(tm); // 12 = Vivek , 14 = Rohan
```

Map Keypoints:

* It is not a child interface of collection.
* If we want to represent group of objects as key-value
* pair then we need to go for map.
* Both Keys and values are objects only
* Duplicate keys are not allowed but values are allowed.
* Key-value pair is called as "Entry".

Map Methods (common for all implementation Map objects)

1. Object put (Object key, Object value) // add entry
2. void putAll (Map m) // to add another map
3. Object get (Object key) // get value based on key
4. Object remove (Object key) // remove object based on key
5. boolean containKey (Object key) // checks for key in Map
6. boolean containValue (Object value) // checks for value in map
7. boolean isEmpty () // check weather map is empty
8. int size() // returns size of the map
9. void clear() // remove all entries.

Views of a map

10. j. Set keySet() // convert keys of map into set.
11. k. Collection values () // convert values of map into collection

Purpose

12. l. Set entrySet() // convert whole entry into set.

Entry (I)

* Each key-value pair is called Entry.
* Without existance of map, there can't be existance of Entry object
* Interface entry is defined inside Map interface.

```
interface Map{
    interface Entry{
        Object getKey();
        Object getValue();
        Object setValue(Object newValue);
    }
}
```

Constructors

1. HashMap hm = new HashMap(); // default capacity 16, loadfactor - 0.75
2. HashMap hm = new HashMap(int capacity);
3. HashMap hm = new HashMap(int capacity, float fillratio);
4. HashMap hm = new HashMap(Map m);

HashTable

* The underlying data structure for HashTable is HashTable only.
* Duplicate keys not allowed but duplicate values allowed.
* Order of insertion is not preserved and is based on hashcode of keys.
* Heterogenous objects are allowed for both keys and values.
* 'null' insertion is not possible for both keys and values, otherwise results in NullPointerException.
* It implements Serializable and Cloneable, but not RandomAccess.
* Every method inside it is synchronized and Thread safe.

Constructors of HashTable:

1. HashTable h = new HashTable(); // initial capacity - 11, load factor - 0.75
2. HashTable h = new HashTable(int initialCapacity);
3. HashTable h = new HashTable(int initialCapacity, float fillRatio);
4. HashTable h = new HashTable(Map m)

* equals(); → HashMap } to compare
* == → IdentityHashMap } content.

Interface new features:

* From Java 8 we can have a method in an interface which has a body (implementation).
* That method is by nature public and abstract, if there is a body for a method then "default" keyword is mandatory.
* We can override that method in implemented classes.
* Normal interface methods need to be overridden but default methods no compulsion of overriding
* From Java 8 we can have a static method with a body in an interface, it will not be inherited in implementing classes
* To invoke static method of an interface we can use interface name.
* From Java 9 we can write private methods with body in an interface, they will not be inherited and cannot be called by class name but they can be used within the interface.

eg:
```
interface Hyder
{
    void teaches();      // public and abstract.
    void writeCode();
    default void disp()
    { System.out.("Normal method allowed"); }

    static void disp2()
    { System.out.("Interface Special method"); }

    private void disp3()
    { System.out.("Interface private method"); }

    private void disp4()
    { System.out.("Interface private method"); } }
```

```
Class Student implements Hyder
{
    public void teacher()
    { System.out.("Hyder teaches java"); }

    public void writeCode()
    { System.out.("Hyder write code"); }
}

public Class LaunchSpecialJava{
    public Static void main (String [] args)
{   Student s = new Student();
        s.teacher();
        s.writeCode();
        s.disp();
        Hyder.disp2(); }
}
```

Introduction to Generics:

Generics was introduced in Jdk 1.5v to provide Type-Safety and to resolve Type-Casting problems.

Type-Safety:

Arrays are always type safe that is we can give the gurantee for the type of elements present inside the array. For ex if our programming requirement is to hold String type of Object, it is recommended to use String array.
In case of String array we can add only String type of objects, if we try to add any other type of objects, it would result in compile time error.

eg : String names[] = new String[500];
name[0] = "Navin Reddy";
name[1] = "Haider";
name[2] = new Integer(100); // CE

That is we can always provide gurantee for the type of elements present inside array and hence arrays are safe to use with respect to type i.e arrays are type safe.

But collections are not type-safe i.e we can't provide any gurantee for the type of elements present inside collections. For example if our programming requirement is to hold only String type of Objects, it is never recommended to go for arraylist.
If we are trying to add any other type we won't get any compile time error but program may fail at runtime.

eg:: ArrayList al = new ArrayList();
al.add("Navin Reddy");
al.add("Haider");
al.add(new Integer(10));
String name = (String) al.get(2); // Exception in thread "main"

Hence we can't provide gurantee for the type of elements present inside collections i.e. collections are not safe to use with respect to type.

Type-Casting (issue)

In case of array at the time of retrival it is not required to perform any type casting
eg:: String name = new String[500];
name[0] = "Navin";
String name = name[0]; // type casting not required.

But in the case of collection at the time of retrival compulsory we should perform type casting otherwise we get compile time error.
eg:

```
ArrayList al = new ArrayList();
        al.add("Nawin Reddy");
String name = al.get(0); // CE
String name = (String) al.get(0); // typecasting is necessary
```

To over come these problems of collections, Generics was introduced in 1.5V. The main objectives of generics are:

1. To provide type-safety to the collections
2. To resolve type-casting problems

To hold only String type of objects we can create a generic version of ArrayList as follows:

→ `ArrayList<String> al = new ArrayList<String>();`
`al.add("Nawin Reddy");`
`al.add(10); // CE`

If we add int type to above example results in "CE" ie through generics we are getting type-safety. At retrival it is not required to perform any type casting.

→ `String name = al.get(0); // type casting not required`

Conclusions:

1. Polymorphism concept is applicable only for the base type but not for parameter type
   eg:: `ArrayList<String> al = new ArrayList<String>();`
   `List<String> al = new ArrayList<String>();`
   `Collection<String> al = new ArrayList<String>();`
   `Collection<Object> al = new ArrayList<String>(); // CE`
2. Collections concept only applicable for objects, parameters can be class or interface but not primitive. eg: `ArrayList<int> // CE`

Enum:

An enum type is a special data type that enables for a variable to be a set of predefined constants. The variable must be equal to one of the values that have been predefined for it.

* Because they are constants, the names of an enum type's field are in uppercase letters.
* We can write an enum within the class or outside of the class.

eg:: enum Result
{ PASS, FAIL, NR; } (internally a class is created for enum)

* All the predefined constants (uppercase words) are static and final by default.
* An enum can have methods and constructors within it.

It can have fields ——→ instance variables ——→ properties

```
eg:: enum Result
     { PASS, FAIL, NR; // static final
       // PASS → public static final Result Pass = new Result();
       // FAIL → public static final Result FAIL = new Result();
       // NR → public static final Result NR = new Result();
       Result()
       { System.out("Constructor is called"); }
     }

public class LaunchEnum
{ public static void main (String[] args)
  { Result res = Result.PASS;
    System.out.println(res);

    Result resArr[] = Result.FAIL.values();
```

```
    for (Result hader: resArr)
    { System.out.println (hyder.ordinal()+":" + hader.name()); }
  }
}
```

Output: Constructor is called
PASS
0: PASS
1: FAIL
2: NR.

eg::
```
enum Course
{ JAVA, JEE, SPRINGBOOT;
  int courseId;
  Course()
  { Sysout ("Constructor"); }

  void SetCourseId (int courseId)
  { this.courseId = courseId; }

  int getCourseId () { return courseId; }
}

public class Launch
{ public static void main (String[] args)
  { Course.JAVA.SetCourseId (10);
    int cid = Course.JAVA.getCourseId();
    System.out.println (cid); }
}
```

Annotations

Annotations are used to provide supplimental information about a program.

* Annotations start with '@'
* Annotations do not change the action of a compiled program.
* Annotations help to associate metadata (information) to the program elements i.e instance variables, constructors, methods, classes etc.
* Annotations are not pure comments as they can change the way a program is treated by the compiler.

eg:: @FunctionalInterface
interface Trial // Compiler will get to know this is a functional interface, any changes would result in CE!
{ int getNum(); }

eg: class Java
{ public void disp1()
{ Sysout ("Parent display"); }
}

Class Focus extends Java
{
@Override
public void disp1() // any changes in method signature would result in compile time error.
{ Sysout ("Focus is key"); }
}

* Annotation → Annotation → parent of all Annotations
* Annotations can be custom (our own)
* Annotations can be used for class, interface, method,

instance variables, local variables, constructors, parameters, enum.

eg:: Custom annotation:

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@interface CricketPlayer
{ //@ → it's not an interface, but it's annotation being created.

    String country();  // String country() default "India"; To give
    int runs();        // int runs() default 20000;  initial values.
}

// Target: to specify for what the annotation can be used, ex.
   ElementType.CLASS - can be used for class only (TYPE - for all)
// Retention: until what the annotation should be active if
   RetentionPolicy.CLASS - it will go till JVM.

@CricketPlayer(country="India", runs=20000) // to assign values
class ViratKohli                              to annotations.
{ . . . . }
                // multiple ElementType can be added in Target.
```

Note: Every inbuilt annotation has a target like @Override can be applied to overridden methods only and cannot be used for class or interface.

eg::
```
{ ViratKholi vk = new ViratKholi();  // Creating ViratKholi object
    Class c = vk.getClass();  // getting the class
  Annotation an = c.getAnnotation(CricketPlayer.class) // get annot.
    CricketPlayer cp = (CricketPlayer) an;  // typecasting
    int runs = cp.runs();  // getting the 'runs' from annotation
```

```
System.out.println(runs); // 20000
String cn = cp.country(); // giving country from annotations
System.out.println(cn); // India
}
```

Note: At a time single annotation can be applied to multiple places by making change at "Target".

→ @Target({ElementType.METHOD, ElementType.LOCAL_VARIABLE})

Generic Classes:

* Until 1.4v a non generic version of Arraylist class is declared as follows.

```
Class Arraylist{
        add(Object o);
        Object get(int index); }
```

add() method can take object as the argument and hence we can add any type of object to the arraylist. Due to this we are not getting type-safety. The return type of get() method is 'object' hence at time of retrieval compulsorily we should perform type casting.

* But in 1.5v a generic version of Arraylist class is declared as:

```
                ↓ Type parameter.
Class Arraylist <T>
{   add(T t);
    T get(int index); }
```

Based on our requirement T will be replaced with our provided type. For example to hold only String type of objects we

can Create ArrayList object as:

```
ArrayList <String> l = new ArrayList <String> ();
```

Internally:

```
Class ArrayList <String>
{   add (String e);
    String get (int index); }
```

add() method can take only String type as argument, hence we can add only String type objects to it, if we try to add other it would result in Compile Time Error.

eg::
```
ArrayList <String> al = new ArrayList <String> ();
al.add ("Nani Reddy");
String name = al.get(0); // typecasting not required.
```

* In Generics we are associating a type parameter to the class, such type of parameterized classes are nothing but Generic Classes.

Generic Class: class with type-parameter.

* Based on our requirement we can Create our own generic classes also.

eg::
```
Class Account <T> { }
Account <Gold> g1 = new Account <Gold> ();
```

eg::
```
Class Gen<T>
{   T obj;
    Gen (T obj) { this.obj=obj; }

    public void Show()
{   System.out ("Type of object is " + obj.getClass().getName()); }
```

```
        public T getObject(){ return obj; }
    }

    Class GenericsDemo{
      public static void main(String[] args)
    {   Gen<Integer> g1 = new Gen<Integer>(10);
            g1.show();
        Sysout(g1.getObject());

        Gen<String> g2 = new Gen<String>("iNeuron");
            g2.show();
        Sysout(g2.getObject()); }
    }
```

Output: The type of object is java.lang.Integer
10
The type of object is java.lang.String
iNeuron.

Note: To get the underlying object of any reference type we use a method called ref.getClass().getName()

eg::

```
Interface Calculator{ }
Class Casio implements Calculator{ }
Class Quartz implements Calculator{ }

Calculator c1 = new Casio();
Sysout(c1.getClass().getName()); // Casio
Calculator c2 = new Quartz();
Sysout(c2.getClass().getName()); // Quartz
```

Bounded types

We can bound the type parameter for a particular range by using extends keyword such types are called bounded types.

```
eg:: Class Test <T> { }
     Test <Integer> t1 = new Test <Integer> ();
     Test <String> t2 = new Test <String> ();
```

there as the type parameter we can pass any type and there are no restrictions hence it is unbounded type.

```
eg:: Class Test <T extends x> { }
```

* If x is a class then as the type parameter we can pass either x or its child classes.
* If x is an interface then as the type parameter we can pass either x or its implementation classes

```
eg:: Class Test <T extends Number> { }
     Class Demo {
          public static void main (String [] args)
     { Test <Integer> t1 = new Test <Integer> ();
       Test <String> t2 = new Test <String> (); // CE
     }
}
```

For interface also we use extends keyword

```
eg:: Class Test <T extends Runnable> { }
     Class Demo {
          public static void main (String [] args)
     { Test <Thread> t1 = new Test <Thread> ();
       Test <String> t2 = new Test <String> (); // CE
     }
}
```

Keypoints about bounded types

* We can't define bounded types by using implements and super keyword
* But implements keyword's purpose we can replace with extends keyword.

  eg:: class Test <T implements Runnable> { } // invalid
  class Test <T super String> { } // invalid.

* As the type parameter we can use any valid java identifier but it convention to use T always.

  eg: class Test<T> { }
  class Test <iNteger> { }

* We can pass any no. of type parameters, need not to be one.

  eg: class HashMap <K, V> { }
  HashMap <Integer, String> h = new HashMap <Integer, String>();

Which of the following are valid?

1. class Test <T extends Number & Runnable> { } // valid
   Number - class, Runnable - interface
2. class Test <T extends Number & Runnable & Comparable> { } // valid
   Number - class, Runnable, Comparable - interface.
3. class Test <T extends Number & String> { } // invalid
   We can't extend more than one class at a time.
4. class Test <T extends Runnable & Comparable> { } // valid
   Both are interfaces so valid
5. class Test <T extends Runnable & Number> { } // invalid
   First class should be there then interface so invalid

Q. Can we apply Type parameter at Method level?

Ans Yes, it is possible.

Generic methode and wild-card character (?)

1. methodOne (ArrayList<String> al):
   This method is applicable for Arraylist of only String type.

```
methodOne (Arraylist<String> al)
{   al.add ("sachin");
    al.add (new Integer(10));); // invalid
}
```

Within this method we can add only String type of object and null to the list.

2. methodOne (ArrayList<?> l):
   We can use this method for Arraylist of any type but within the method we can't add anything to the list except null.

```
eg:  l.add (null); // valid
     l.add (10); // invalid
     l.add ("String"); // invalid
```

This method is useful whenever we are performing read operation.

3. methodOne (Arraylist<? extends X> al)
   * X → class, we can make a call to method by passing Arraylist of X type or it's Child type.
   * X → interface, we can make a call to method by passing Arraylist of X type or it's implementation Class.
   * You can't add anything except null and hence best suited for read operation.

4. methodOne (Arraylist<? Super X> al)
   * X → Class, we can make a call to method by passing Arraylist of X type or it's Super Class.
   * X → interface, we can make a call to method by passing Arraylist

of x type or its Super Class of implementation Class of x.

```
method One (ArrayList<? Super X> al)
{   al.add(x); // becau of only x type, you can add
    al.add(null); }
```

Which of the following declarations are allowed?

1. ArrayList<String> l1 = new ArrayList<String>(); // valid
2. ArrayList<?> l2 = new ArrayList<String>(); // valid
3. ArrayList<?> l3 = new ArrayList<Integer>(); // valid
4. ArrayList<? extends Number> l4 = new ArrayList<Integer>(); // valid
5. ArrayList<? extends Number> l5 = new ArrayList<String>(); // invalid
6. ArrayList<?> l4 = new ArrayList<? extends Number>(); // invalid
7. ArrayList<?> l6 = new ArrayList<?>(); // invalid

Type-parameter at method level.

```
Class Demo<T>{
                ↓ Type parameter defined just b4 return type.
    public <T> void m1(T t){...}
}
```

Which of the following declarations are allowed?

1. public <T> void m1(T t) // valid
2. public <T extends Number> void m1(T t) // valid
3. public <T extends Number & Comparable> void m3(T t) // valid
4. public <T extends Number & Comparable & Runnable> void m4(T t) // valid
5. public <T extends Number & Thread> void m5(T t) // invalid

6. public <T extends Runnable & Number> void m6(T t) // invalid
7. public <T extends Number & Runnable> void m7(T t) // valid

Communication with non generic code:
To provide compatability with older versions Sun ms people compromised the concept of generics in few areas.
eg:

```
Class Test{
    public static void main(String[] args)
{   ArrayList<String> l = new ArrayList<String>();
    l.add("sachin");
    l.add(10); // CE

    m1(l);
    l.add(10.5); // CE
    Sysout(l); } // Sachin, 10, dhoni, true

  public static void m1(ArrayList l)
{     l.add(10);
      l.add("dhoni"); }
}
```

Conclusions: Generic concept is applicable only at compile time, at runtime there is no such concept.
At the time of compilation, as the last step generics concept is removed, hence for JVM generics syntax won't be available.
Hence the following declarations are equal:

```
ArrayList l = new ArrayList<String>();
ArrayList l = new ArrayList<Integer>();
ArrayList l = new ArrayList<Double>();
```

All are equal at runtime, because compiler will remove these generics syntax. → ArrayList l = new ArrayList();

eg: Arraylist l = new Arraylist<String>();
l.add(10);
l.add(true);
Sysout(l); // 10, true

RHS part is ignored, Since LHS is generic, so no error.

eg:: class Test{

public void m1(Arraylist<String> l){ }
public void m1(Arraylist<Integer> l){ }
} // CE: duplicate methode found.

Behind the Scenes by the Compiler:
1. Compiler will Scan the code
2. Check the argument type
3. If generice found in argument type then remove generice Syntax.
4. Compiler will check again the Syntax.

eg:: The following two declarations are equal
Arraylist<String> l1 = new Arraylist();
Arraylist<String> l2 = new Arraylist<String>();
For these objects we can add only String type of objects:
l1.add("A"); // valid
l1.add(10); // invalid

Comparable vs Comparator

Public TreeSet();
When we use the above constructor, JVM will internally use Comparable interface method to sort the objects based on default natural sorting order.

Q1 What is comparable interface?

Ans. It is a functional interface present in java.lang. package. This interface is internally used by TreeSet object during sorting process of the object.

```
@FunctionalInterface
public interface java.lang.Comparable<T>{
    public abstract int CompareTo(T);
}
```

eg::
```
TreeSet ts = new TreeSet();
ts.add("A");
ts.add("z");
ts.add("B");
ts.add(null); // Null pointer exception
ts.add(10); // Class Cast exception
System.out.println(ts); // [A, B, L, Z]
// Sorting of object will happen based on default natural
Sorting Order.
```

Note: If we are keeping the data inside TreeSet object, then data should be:

a. Homogenous → because it uses CompareTo() to sort the object.

b. The object should compulsory implement an interface called "Comparable", if we fail to do so it would result in "Class Cast Except."

eg::
```
TreeSet ts = new TreeSet();
ts.add(new StringBuffer("A"));
ts.add(new StringBuffer("z"));
ts.add(new StringBuffer("Y"));
System.out.println(ts); // Class Cast exception.
```

Note: All wrapper classes and String classes has implemented "Comparable" interface. StringBuffer class has not implement Comparable interface, so above program would result in "ClassCastException".

* If we are depending on default natural sorting order compulsor-ily object should be Homogenous and Comparable, otherwise results in "RE".
* An object is said to be comparable if and only if corresponding class implements comparable interface.
* All wrapper classes, String class already implements Comparable interface. But StringBuffer class doesn't implement Comparable interface.

```
@FunctionalInterface
public interface Comparable<T>{
    public abstract int CompareTo(T);}
```

→ Obj1.CompareTo(Obj2)
- returns -ve value, if obj1 has to come before obj2
- returns +ve value, if obj2 has to come before obj1.
- returns 0 if both are equal.

eg::
```
Class Test{
    public static void main(String[] args)
  { TreeSet ts = new TreeSet();
      ts.add("A");
      ts.add("Z");
      ts.add("L");
      ts.add("B");
    Sysout(ts); }
}
```

// A, B, L, Z

Rules in binary tree
-ve means node should be at left
+ve means node should be at right
zero means nodes are duplicated

Comparable (I)

Comparable interface present in java.lang package and it contains only one method compareTo().

→ obj1.compareTo(obj2)

returns -ve if obj1 has to come before obj2.

returns +ve if obj1 has to come after obj2.

returns 0 if and only if both are equal.

eg:: Sysout("A".compareTo("z")); // -ve value

Sysout("z".compareTo("k")); // +ve value

Sysout("k".compareTo("k")); // zero

Sysout("R".compareTo(null)); // NullPointerException

* Whenever we are depending on default natural sorting order and if we are trying to insert elements then internally JVM will call compareTo() to identify sorting order.
* For 'String' and 'Number' natural sorting order is ascending order.

Comparator (I)

If we are not satisfied with default sorting order or if default natural sorting order is not already available then we can define our own sorting by using comparator object.

→ Public interface java.util.Comparator<T>{

Public abstract int compare(T, T);

Public abstract boolean equals(java.lang.Object);}

* Comparator (I) is present in java.util. Package. It contains two methods - compare() and equals().

→ public int compare (Object obj1, Object obj2)
returns -ve if obj1 comes before obj2
returns +ve if obj1 comes after obj1.
returns 0 if obj1 and obj2 are equal

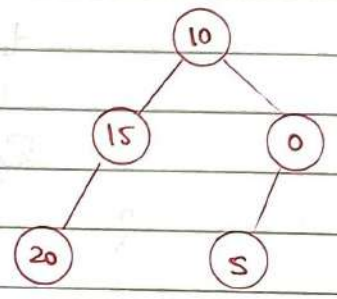
→ public boolean equals (Object o):
Whenever we are implementing Comparator Interface compulsory we should provide implementation for compare().
Implementing equals() is optional because it is already available to our class from Object class through Inheritance.

```
eg:: Class TreeSetDemo{
        public static void main (String [] args)
    { TreeSet t = new TreeSet (new MyComparator ());// Line 1.
            t.add (10);
            t.add (0);
            t.add (5);
            t.add (15);
            t.add (20);          // 20, 15, 10, 5, 0
        System.out.println (t); }
    }

Class MyComparator implements Comparator {
    public int compare (Object obj1, Object obj2)
    {       Integer i1 = (Integer) obj1;
            Integer i2 = (Integer) obj2;
            if (i1 < i2)
                    return +1;
    else if (i1 > i2) return -1;
                else return 0; }
    }
```

* If we are not passing comparator object as an argument then internally JVM will call compareTo(), in that case output will be [0, 5, 10, 15, 20].

* At line1 if we are passing comparator object then JVM will call compare() instead of compareTo() which is meant for customized sorting.

```
TreeSet ts = new TreeSet(new MyComparator());
    ts.add(10);
    ts.add(0);   Compare(0,10)
    ts.add(15);  Compare(15,10)
    ts.add(5);   Compare(5,10)
                 Compare(5,0)
ts.add(20);      Compare(20,10)
                 Compare(20,15)
    → 20 15 10 5 0
```



<u>Various possible implementations of compare()</u>

```
public int compare(Object obj1, Object obj2)
{   Integer i1 = (Integer) obj1;
    Integer i2 = (Integer) obj2;
      return i1.compareTo(i2); // ascending
      return -i1.compareTo(i2); // descending
      return i2.compareTo(i1); // descending
      return -i2.compareTo(i1); // ascending
      return +1; // insertion order
      return -1; // reverse of order of insertion
  } return 0; // only first element.
```

* W.A.P. to insert String Objects into the TreeSet where the Sorting order is of reverse of alphabetical order.

→
```
import java.util.*;
Class TreeSetDemo{
    public static void main(String[] args){
        TreeSet t = new TreeSet(new MyComparator());
            t.add("sachin");
            t.add("pointing");
            t.add("sangakara");
            t.add("fleming");
            t.add("lara");
            System.out.println(t); }
    }
Class MyComparator implements Comparator{
    public int Compare(Object obj1, Object obj2)
    {       String s1 = (String) obj1;
            String s2 = obj1.toString();

            return s2.compareTo(s1); }
}
```

* W.A.P to insert StringBuffer Objects into the TreeSet where Sorting order is alphabetical order.

→
```
import java.util.*;
Class TreeSetDemo{
    public static void main(String [] args)
    {       TreeSet t = new TreeSet(new MyComparator1());
```

```
t.add (new StringBuffer("A"));
t.add (new StringBuffer("Z"));
t.add (new StringBuffer("K"));
System.out.println(t); }
}

Class MyComparator1 implements Comparator{
    public int Compare (Object obj1, Object obj2)
{       String s1= Obj1.toString();
        String s2= Obj2.toString();
        return s1.CompareTo(s2); }  // A, K, Z
}
```

★ W.A.P to insert String and StringBuffer Objects into the TreeSet where sorting order is increasing length order. If two objects have same length then consider their alphabetical order

→

```
import java.util.*;
Class TreeSetDemo{
    public static void main (String [] args)
{   TreeSet t= new TreeSet (new MyComp());
    t.add("A");
    t.add (new StringBuffer("ABC"));
    t.add (new StringBuffer("AA"));
    t.add ("XX");
    t.add ("A");
    System.out.println(t); }
}

Class MyComp implements Comparator{
    public int Compare (Object obj1, Object obj2){
```

```
String s1 = obj1.toString();
String s2 = obj2.toString();
if int i1 = s1.length();
   int i2 = s2.length();
if (i1<i2) return -1;
else if (i1>i2) return 1;
else return s1.compareTo(s2); }
}
```

Note: * if we use TreeSet(), then condition is:

a. Object should be homogenous

b. Object should be comparable (class should implement Comparable (I)).

* If we use TreeSet (Comparator c) then:

a. Object need not be homogenous

b. Object need not implement Comparable (I).

Q.1. When to go for Comparable (I) and when to go for Comparator (I)?

Ans Predefined comparable classes like String, Wrapper class for which default natural sorting is available, If we are not satisfied with natural sorting order or we need to modify then we need to go for Comparator (I).

* For predefined Non-Comparable class like StringBuffer, Comparator is used for both natural sorting order and customized sorting order.

* For user defined class like Employee, Student, the developer if he comes up with own logic of sorting, then he should implement Comparable (I) and give it as a ready made logic.

```
class Employee implements Comparable
{   int id;
    String name;
    int age;

    public int compareTo (Object obj){
        // Sorting is done based on "id"
        ;;
    }
}
```

If the developer is using Employee class, if he is not interested in sorting based on "id" given by the api then he can use "Comparator".

Comparable v/s Comparator (usage)

→ For predefined comparable classes (like String) default natural sorting order is already available. We can customize it by defining our own sorting by comparator object.

→ For pre-defined non-comparable classes (like StringBuffer) default natural sorting order is not available, if we want to define our own sorting we can use comparator object.

→ For our own classes (like Employee) the person writing Employee class is responsible to define (default) sorting order by implementing comparable interface

→ The person who is using our (Employee) class wants to modify the order of sorting then he can define his own sorting order by using comparator object.

★ W.A.P to insert Employee objects into the tree set where DNSO is based on ascending order of Employee Id and customized sorting order is based on alphabetical order of names.

→ import java.util.*;

```
Class Employee implements Comparable{
    String name;
    int eid;
    Employee (String name, int eid)
    {   this.name = name;
        this.eid = eid; }

  Public String toString(){return name+" "+eid; }
  public int CompareTo (Object Obj)
  {     int eid1 = this.eid;
        Employee e = (Employee) Obj;
        int eid2 = e.eid;
        if (eid1 < eid2) return -1;
        else if (eid1 > eid2) return 1;
        else return 0; }
  }

Class Test{
  public static void main (String [] args)
  {     Employee e1 = new Employee ("sachin", 10);
        Employee e2 = new Employee ("pointing", 14);
        Employee e3 = new Employee ("lara", 9);
        Employee e4 = new Employee ("anwar", 23);
        Employee e5 = new Employee ("flintoff", 17);

        TreeSet t = new TreeSet();
        t.add (e1); t.add (e2);
        t.add (e3); t.add (e4);
        t.add (e5);
        System.out.println (t);
```

Output:

lara 9
Sachin 10
Pointing 14
flintoff 17
anwar 23

```
TreeSet t1 = new TreeSet(new MyComparator());
    t1.add(e1);  t1.add(e2);
    t1.add(e3);  t1.add(e4);
    t1.add(e5);
    System.out.println(t1); }
}
```

```
anwar 23
flintoff 17
lara 9
pointing 14
sachin 10
```

Output.

```
Class MyComparator implements Comparator{
    public int compare(Object obj1, Object obj2)
{   Employee e1 = (Employee) obj1;
    Employee e2 = (Employee) obj2;
    String s1 = e1.name;
    String s2 = e2.name;
    return s1.compareTo(s2); }
}
```

	Comparable (I)	Comparator (I)
1.	Present in java.lang package	Present in java.util package
2.	It is meant for default natural Sorting order	It is meant for Customized Sorting order.
3.	Define only one method: compareTo()	Define two methods: compare() and equals()
4.	All wrapper class and String class implements comparable interface	The only implemented classes of Comparator are Collator and RuleBaseCollator.

Functional Interfaces

If an interface contains only one abstract method then such interfaces are called as "Functional Interface".

```
public interface. java. util. function. Predicate <T>
{   public abstract boolean test (T);
    // default methods
  Public java.util.function. Predicate <T> and (java.util.function.
      Predicate < ? Super T> );
  public java.util. function. Predicate <T> negate ();
  public java. util. function Predicate <T> or (java. util function.
      Predicate < ? Super T.> );

    // Static method
  public Static <T> java. util. function. Predicate <T> isEqual
      (java.lang. Object);
}
```

Use of Predicate:

```
Class MyPredicate implements Predicate <Integer>
{   @Override
    public boolean test (Integer i)
  {  if (i>10)
          return true;
     else return false; }
}
```

* Instead of writing a Seperate Class we can write lambda expression:

```
→ Predicate <Integer> p = i → i>10;
      Sypout (p.test (10)); // false
      Sypout (p. test (100)); // true.
```

* Write a predicate to check weather the given String length is >=3

```
Class MyPredicate implements Predicate<String>
{   @Override
    public boolean test (String name)
    {   if (name.length() >=3)
            return true;
        else return false; }
}
```

<u>Lambda Expression:</u>

```
Predicate<String> p = name -> name.length()>=3;
Sysout (p.test ("PWC"));  // true
Sysout (p.test ("CS"));  // false
```

<u>Default methods available as utility methods:</u>

```
public default Predicate<T> and (Predicate p);
public default Predicate<T> negate ();
public default Predicate<T> or (Predicate p);
```

```
import java.util.function.*;
public class Test{
    public static void main (String [] args)
    {
        int [] arr = { 0, 5, 10, 15, 20, 25, 30};
        Predicate<Integer> p1 = i -> i>10;
        Sysout ("Elements greater than 10 are:");
        m1 (p1, arr);  // 15, 20, 25, 30

        Predicate<Integer> p2 = i -> i%2==0;
        Sysout ("Even numbers are:");
        m1 (p2, arr);  // 0 10 20 30
```

```
Sysout(" Elements greater-than 10 and even are :");
m1(p1.and(p2), arr); // 20 30

Sysout(" Elements greater-than 10 or even are :");
m1(p1.or(p2), arr); // 0 10 15 20 25 30

Sysout(" Elements not even are :");
m1(p2.negate(), arr); } // 5 15 25

public static void m1(Predicate<Integer> p, int[] x)
{   for (int ele : x)
        if (p.test(ele))
            Sysout(ele); }
}
```

Function (I)

```
                                                Input type
                                                    ↑
public interface java.util.function.Function<T, R>{
    // 1 abstract method                            ↓
    public abstract R apply(T);                 Return type.

    // default methode
    public <V> java.util.function.Function<V, R>
        compose(java.util.function.Function<? super V, ? extends T>);
    public <V> java.util.function.Function<T, V> andThen
        (java.util.function.Function<? super R, ? extends V>);

    // static method
    public static <T> java.util.function.Function<T, T> identity();
}
```

Writing a code using Implementation Class

```
Class MyFunction implements Function<String, Integer>
{ @Override
    public Integer apply (String name)
    { return name.length(); }
}

Public Class Test{
    public static void main (String[] args)
{   Function<String, Integer> f = new MyFunction();
        int output = f.apply("Sachin");
        Sysout (output);
        Sysout ("sachin".length()); }
}
```

Above Code with Lambda Expression:

```
public Class Test{
    public static void main (String args[])
{   Function<String, Integer> f = name -> name.length();
        int output = f.apply ("sachin");
        Sysout (output); }
}
```

Note: When to go for Predicate and when for Function?

* Predicate: To implement some conditional checks we should go for Predicate.
* Function: To perform some operation and to return some result we should go for Function.

Method Reference (::) and Constructor reference (::)

:: → Scope resolution operator.

Syntax for method reference

1. Static method

   Class Name :: methodName

2. instance method

   Object :: methodName.

eg::
```
public class Test{
    public static void m1(){
        for (int i=1; i<=10; i++)
        { Sysout (" Child thread"); }
    }
public static void main(String [] args) throws Exception {

// using method reference binded the method call of run() of
   interface 'Runnable'
  Runnable r = Test :: m1;
  Thread t = new Thread(r);
     t.start();
  for (int i=0; i<10; i++)
   { Sysout (" main thread"); }
  }
}
```

Output:

```
Child thread
Child thread
  :    :
main thread
main thread
  :    :
```

eg:: interface Interf
{ public void m1(int i) }

public class Test{
public void logic(int i){ System.out("method ref."); }

public static void main(String [] args)
{ Interf i = x → System.out("lambda expression");
i.m1(100);

Interf i1 = new Test() :: logic;
i1.m1(10); }
}

Output:
"lambda expression"
"method ref."

eg:: Constructor reference

Class Sample
{ private String s;
Sample (String s) { this.s=s;
System.out("Constructor executed"); }
}

@FunctionalInterface
interface Interf
{ public Sample get(String s); }

public class Test{
public static void main (String [] a)
{
Interf i = s → new Sample (s);
i.get ("from lambda expression...");

```
        // Constructor refrence
        Interf i1 = Sample :: new;
        i1.get("from constructor reference"); }
    }
```

Usage of "forEach()" to print elements of ArrayList.

```
import java.util.*;
import java.util.function.*;
// public void forEach (java.util.function.Consumer<? super E>);
// public abstract void accept (T t)
Class MyConsumer implemments Consumer<String>
{   public void accept (String name){
        Sysout (" accept method call");
        Sysout (name); }
    }

Public Class Test{
    public static void main (String [] args){
        ArrayList<String> names = new ArrayList<String>();
    names.add ("sachin"); names.add ("dhoni");
        // Traditional approach
    Consumer<String> consumer = new MyConsumer();
    names.forEach (consumer);

    // lambda expression
    names.forEach (name -> Sysout(name));

    // method refrence
    names.forEach (System.out :: println); }
}
```

Stream API

* **Stream**: Channel through which there is a free flow movement of data.
* **Streams**: To process objects of the collection, in 1.8v Streams Concept was introduced.

Q1. Difference between Java.util.Streams and java.io Streams.

Ans Java.util Streams meant for processing objects from the collection. That is it represents a Stream of objects from the collection. But java.io Streams meant for processing binary and Character data with respect to file that is it represents stream of binary data or character data from the file, hence both Java.io Streams and Java.util Streams both are different.

Q2. Difference between Collection and Stream?

Ans
* If we want to represent a group of individual objects as a single entity then we should go for collection.
* If we want to process a group of objects from the collection then we should go for Streams
* We can create a Stream object to the collection by using Stream() method of Collection interface. Stream() method is a default method added to Collection in 1.8v

```
import java.util.*;
import java.util.Stream.*;
public class Test{
    public static void main (String [] args){
    ArrayList <Integer> al = new ArrayList <Integer> ();
        al.add(0);  al.add(5);  al.add(10);
        al.add(15);  al.add(20);  al.add(25);
            Sysout (al);  // [0, 5, 10, 15, 20, 25]
```

```
// Till JDK 1.7v
ArrayList<Integer> evenlist = new ArrayList<Integer>();
for (Integer i1 : al)
    if (i1 % 2 == 0)
        evenlist.add(i1);
Sysout(evenlist); // [0, 10, 20]

// From JDK 1.8v we use Streams
// Configuration → al.stream()
// Processing → filter(i → i % 2 == 0).collect(Collectors.toList())
List<Integer> Streamlist = al.stream().filter(i → i % 2 == 0)
        .collect(Collectors.toList());
Sysout(Streamlist); // 0 10 20
Streamlist.forEach(System.out::println); }
}                   // 0
                    // 10
                    // 20
```

eg:: (for the above code, continuation)

```
// Till JDK 1.7v
ArrayList<Integer> doublelist = new ArrayList<Integer>();
for (Integer i1 : al)
    doublelist.add(i1 * 2);
Sysout(doublelist); // 0 10 20 30 40 50

// from JDK 1.8v
// map - for every Object if a new Object has to be created then
    go for map.
List<Integer> Streamlist = al.stream().map(obj → obj * 2)
        .collect(Collectors.toList());
Sysout(Streamlist); // 0 10 20 30 40 50
Streamlist.forEach(i → Sysout(i));
Streamlist.forEach(System.out::println); } 0 10 20 30 40 50
```

* Stream is an interface present in java.util.Stream. Once we got the Stream, by using that we can process objects of that collection.
We can process the objects in 2 phases:
1. Configuration
2. Processing.

1. Configuration:

We can configure either by using filter mechanism or by using map mechanism.

Filtering: We can configure a filter to filter elements from the collection based on some boolean condition by using filter() method of Stream interface.

→ public Stream filter (Predicate<T> t)

Here (Predicate<T> t) can be a boolean valued function/lambda expression.

eg:: Stream S = C.Stream();
Stream S1 = S.filter (i → i%2 == 0);

Hence to filter elements of collection based on some boolean condition we should go for filter method.

Mapping: If we want to create a seperate new object, for every object present in the collection based on our requirement then we should go for map() method of Stream interface.

→ public Stream map (Function f);

It can be lambda expression also.

eg:: Stream S = C.Stream();
Stream S1 = S.map (i → i+10);

Once we perform operations we can process objects by using several methods.

2. Processing:

Processing can be done by multiple methods:

1. Collect () method
2. Count () method
3. Sorted () method
4. min() and max method
5. forEach() method
6. toArray() method
7. Stream.of() method

→ eg::
```
import java.util.*;
import java.util.Stream.*;
public Class Test{
    public Static void main (String [] args)
{   ArrayList<String> names = new ArrayList<String>();
    names.add ("sachin"); names.add ("harbajan");
    names.add ("dhoni"); names.add ("Kohli");

  List<String> result = names.stream().filter (name -> name.
      length()>5).Collect (Collectors.toList());
    Sysout (result.size()); //2

  long count = names.Stream().filter (name -> name.length()>5)
      .count();
    Sysout (count); } //2
}
```

→ eg::
```
import java.util.*;
import java.util.Stream.*;
```
// Comparable (Predefined API for natural Sorting order)

→ compareTo (Object obj)

// Comparator (for user defined class for customized sorting order)
→ compare (Obj1, Obj2)

```
Public class Test{
    public static void main (String [] args)
{   ArrayList <Integer> al = new ArrayList<Integer> ();
        al.add (10); al.add (0); al.add (15);
        al.add (5); al.add (20);
        Sysout ("Before Sorting:" + al); // 10 0 15 5 20

        // Using Stream api
    List<Integer> result = al.stream().Sorted().Collect(Collectors.
                                toList());
        Sysout ("After Sorting ::" + result);
                    // After Sorting 0 5 10 15 20
    List<Integer> custom = al.Stream().Sorted((i1, i2) →
                    i2.compareTo(i1).Collect(Collectors.toList());
        Sysout ("After Sorting ::" + custom); }
                    // After Sorting 20 15 10 5 0
}
```

→ eg::

```
ArrayList <Integer> al = new ArrayList <Integer> ();
    al.add (10); al.add (0); al.add (15);
    al.add (5); al.add (20);
    Sysout (al); // 10 0 15 5 20

Object [] ObjArr = al.Stream().toArray();
for (Object Obj : ObjArr)
        Sysout (obj); // 10 0 15 5 20

Integer [] ObjArr1 = al.stream().toArray(Integer[]::new);
for (Integer i : ObjArr1)  Sysout (i);
                    // 10 0 15 5 20
```

(i) Collect() : This method collects the elements from the stream and adding to the specified to collection indicated by the argument.

```
eg:: { ArrayList<String> namee = new Arraylist<String>();
       al.add("sachin"); al.add("saurav");
       al.add("dhoni"); al.add("yuvi");

  List<String> result = namel.Stream().filter(name -> name
          .length()>5).collect(Collectors.toList());
  Sysout(result);

  List<String> mapR = namee.Stream().map(name ->
          name.toUpperCase()).collect(Collectors.toList());
  Sysout(mapR);
}
```

(ii) Count() : This method returns number of elements present in the stream.

→ public long count()

```
eg:: { ArrayList<String> namel = new Arraylist<String>();
       namee.add("sachin"); namee.add("Kohli");
       namee.add("yuvaraj");

  long count = namee.Stream().filter(name -> name.length()>5).
          count();
  Sysout(count); }
```

(iii) Sorted() : If we want to sort the elements present inside stream then we should go for sorted() method.

Sorted() → default natural sorting order

Sorted(Comparator c) → Customized sorting order

```
eg:: {ArrayList<Integer> al = new ArrayList<Integer>();
      al.add(10); al.add(20); al.add(0);
      al.add(5); al.add(25); al.add(15);

  List<Integer> result = al.stream().sorted().collect(Collectors.
                                toList());
      Sysout(result);

  List<Integer> custom = al.stream().sorted((i1,i2) → i1.compareTo
                                (i2)).collect(Collectors.toList());
      Sysout(custom); }
```

(iv) min() and max() :

min(Comparator c) returns min value according to specified comparator.

max(Comparator c) returns maximum value according to specified comparator.

```
eg:: { ArrayList<Integer> al = new ArrayList<Integer>();
       al.add(75); al.add(15); al.add(3); al.add(0);

  Integer mn = al.stream().min((i1,i2) → i1.compareTo(i2)).get()
      Sysout(mn);

  Integer mx = al.stream().max((i1,i2) → i1.compareTo(i2)).get()
      Sysout(mx); }
```

(v) forEach() : This method will not return anything. It will take lambda expression as argument and apply that lambda expression to each element present in stream.

```
eg:: { ArrayList<String> al = new ArrayList<String>();
        al.add("Sachin");
        al.add("Kohli");

    al.Stream().forEach(al -> System.out(al)); // Sachin Kohli
    al.Stream().forEach(System.out::println); }
                // Sachin
                // Kohli
```

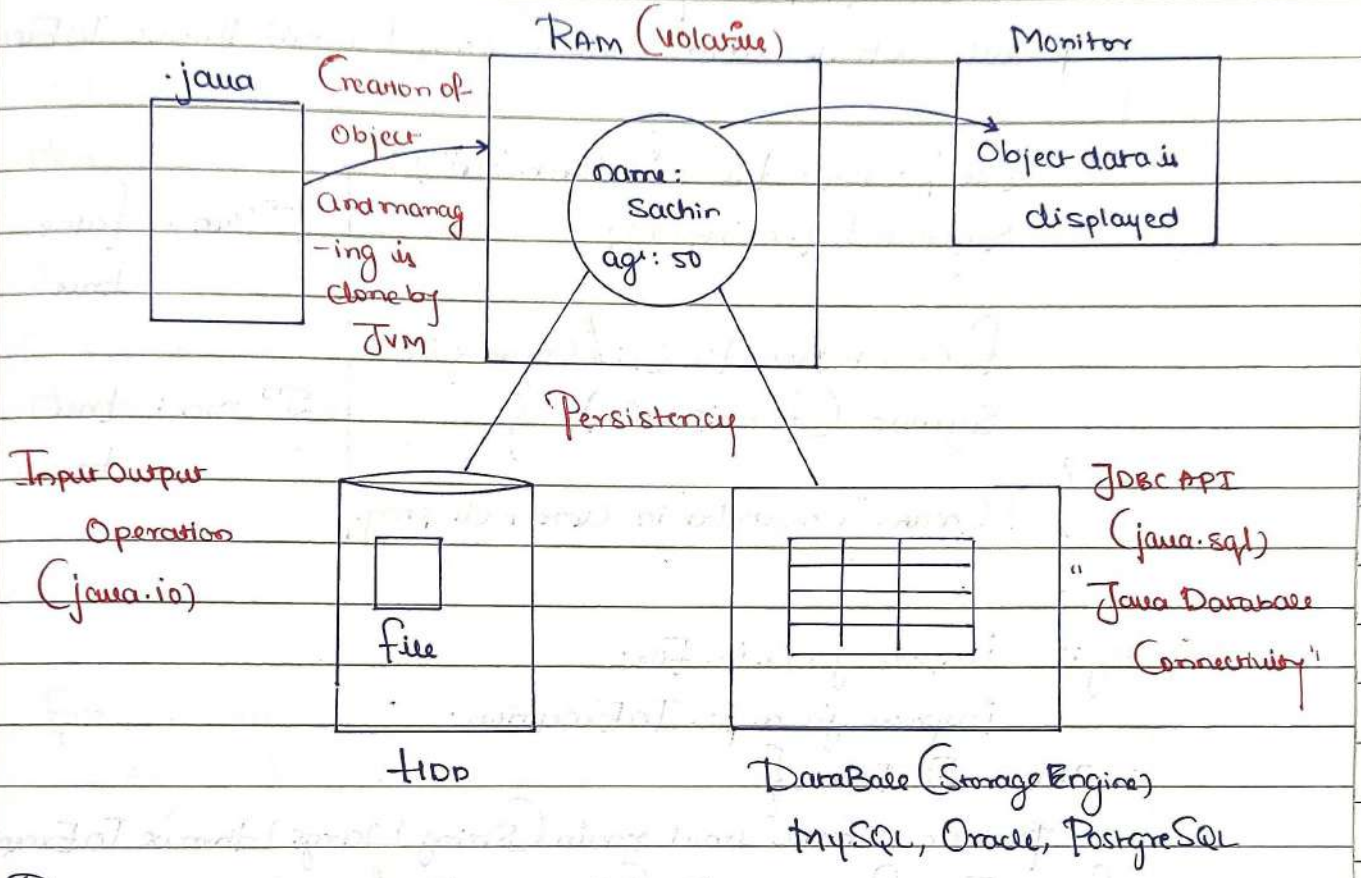
(vi) toArray() : We can make use of toArray() method to copy elements present in the stream into specified array.

```
eg:: { ArrayList<Integer> al = new ArrayList<Integer>();
        al.add(10); al.add(5);

    Integer[] arr = al.Stream().toArray(Integer[]::new);
    for (Integer element : arr)
        System.out(element); } // 10 5
```

(vii) Stream.of() : We can also apply a stream for group of value and for array.

```
eg:: { Stream s = Stream.of(99, 999, 9999);
        s.forEach(System.out::println);

    Double[] d = {10.0, 10.1, 10.2, 10.3}
        Stream s1 = Stream.of(d);
        s1.forEach(System.out::println); }
```

File Operation in Java



Persistency:

It is a mechanism of storing the data permanently on to the file. In java persistency can be achieved through an API's available inside package called "java.io".

File:

File f = new File("abc.txt");

* The line first checks weather abc.text file is already available or not, if it is already available then "f" simply refers that file.
* If it is not already available then it won't create any physical file, just creates a java file object represents the name of the file.
* A java file object can represent a directory also.

→ eg:: import java.io.*;

```
class FileDemo{
  public static void main (String [] args) throws IOException
{
    File f = new File ("cricket.txt");
    Sysout (f.exists());

    f.CreateNewFile(); // Create file
    Sysout (f.exists()); }
} // Creates "Cricket.txt" in current directory.
```

1st Run : false
true

2nd run : true
true

→ eg:: import java.io.File;
import java.io.IOException;

```
Class FileDemo{
  public static void main (String [] args) throws IOException
{   File f = new File ("IPLTeams"); // no extension → directory
    Sysout (f.exists());

    f.mkdir (); // Create directory
    Sysout (f.exists()); }
} // Creates a directory in
  current directory (folder)
```

1st Run : false
true

2nd Run : true
true

<u>Note</u>: In UNIX everything is a file, java "file IO" is based on UNIX operating system hence in java also we can represent both files and directories by file object only.

File Class Constructors:

1. File f = new File(String name);
→ Create a java file object that represents name of the file or directory in current working directory.

2. File f = new File(String subdir name, String name);
→ Create a file object that represents name of the file or directory present is specified sub directory.

```
eg:: File f1 = new File("abc");
     f1.mkdir();
     File f2 = new File("abc", "demo.txt");
```

3. File f = new File(File subdir, String name);

```
eg:: File f1 = new File("abc");
     f1.mkdir();
     File f2 = new File(f1, "demo.txt");
```

→ eg::

```
Class Test{
    public static void main(String [] args)
    {
        File f = new File("iNeuron");
        f.mkdir();
        Sysout(f.isDirectory()); // true

        File f1 = new File(f, "abc.txt");
        f1.CreateNewFile();
        Sysout(f1.isFile()); } // true
    }
```

// Create a "iNeuron" directory in current working directory and a file "abc.text" is created inside it //

// If "iNeuron" directory is not available and you try to create file inside it, it would result in "FileNotFoundException".

eg:: { String location = "C:\\pwskills";
File f = new File (location);
f.mkdir ();

File f1 = new File (f, "java.txt");
f1.CreateNewFile ();
}

// Creates the "pwskills" directory in C drive and then "java.txt" file is created inside it. //

Important methods of file class:

1. boolean exists(): returns true if physical file or directory available.
2. boolean CreateNewFile(); : this first checks if physical file is already available or not, if it is already available then it returns false without creating any physical file
If file is not already available then it creates new file and returns true.
3. boolean mkdir(): this method checks if directory already available or not, if available then it returns false and returns without creating anything. If directory not already available then it will create new directory and returns true.
4. boolean isFile(): returns true if file object represents a physical file
5. boolean isDirectory(): returns true if file object represents a directory.
6. String [] list(): returns names of all files and subdirectories present in specified directory.
7. long length(): returns number of characters present in a file.
8. boolean delete(): to delete a file or a directory.

Write a program to display the names of all files and directories present in "D:\\Java Job Guranteed Batch" and return the number of files and directories present inside it.

```
import java.io*;
Class Test{
  public Static void main (String [] args) throws Exception
{   int dirCount=0;
    int jpgFileCount=0;
    int txtFileCount=0;
    int zipFileCount=0;
  String location = "D:\\ Java Job Guranteed Batch";
    File f= new File(location);

  String [] names = f.list();
  for (String name: names)
{   // iterating over all file in location folder
    File f1 = new File (f,name); // Creating new file for every file to
    if (f1. isDirectory()) dirCount++;  apply 'isFile()' method
                                  // new file not created if its already
    if (f1. isFile())             available.
  {   if (name. endeWith (".png")) jpgFileCount++;
      if (name. endeWith (".txt")) txtFileCount++;
      if (name. endeWith (".zip")) txtFileCount++; }

    System. out. println(name); }
                                  // in loop new file are not created
  Sysout (jpgFileCount);          as they are already available, but
  Sysout (txtFileCount);          it will point to already available
  Sysout (zipFileCount);          file and file methods can be
  Sysout (dirCount); }            applied.
}
```

FileWriter : By using FileWriter Object we can write Character data to the file.

→ FileWriter fw = new FileWriter (String name);

→ FileWriter fw = new FileWriter (File f);

The above two constructors meant for overriding the data to the file.

* Instead of overriding if we want append operation then we should go for:

→ FileWriter fw = new FileWriter (String name, boolean append);

→ FileWriter fw = new FileWriter (File f, boolean append);

* If the specified physical file is not already available then these constructors will create that file.

Methods:

1. write (int ch) : to write a single character to the file.
2. write (char[] ch) : to write an array of characters to the file.
3. write (String s) : to write a string to the file
4. flush () : to give the guarantee the total data includes last character also written to the file.
5. close () : to close the stream.

```
import java.io.*;
Class Test{
  public static void main (String [] args)
{
    File f1 = new File ("abc.txt");
    FileWriter f2 = new FileWriter (f1); // write to f1 file

      // Performing write operation on a file
    f2.write (97); // 'a' (ascii value of 97)
```

```
    f2.write('\n'); // adds a new line | acts like 'Enter' key
    f2.write("Hello World!"); // to add string
  Char [] ch = {'V','I','V','E','K'};
    f2.write(ch); // passing char array
      f2.flush(); // making the data to write to the file.
    f2.close(); } // to close the resourse
}
```

A new "abc.txt" file is created in current working directory.

<u>Note</u>: If "abc.txt" already available and if you run this code then the content will be overridden.
To append change → <u>new FileWriter(f, true)</u>; this will now append all the writings to the available file.

The main problem with FileWriter is we have to insert line seperator manually, which is difficult to the programmer ('\n'), and line seperator varying from system to system.

<u>File Reader</u>: By using FileReader object we can read character data from the file.

* FileReader fr = new FileReader(String name);
* FileReader fr = new FileReader(File f);

<u>Methods</u>:

1. <u>int read()</u>: it attempts to read next character from the file and return it's unicode value. If next character is not available then we will get -1.
   eg:: int i = fr.read();
2. <u>int read(Char[] ch)</u>: it attempts to read enough characters from the file into Char[] array and returns the number of characters copied from the file into Char array.

eg:: import. java. io. FileReader;
import java. io. IoException;

```
Public Class Test{
  Public Static void main (String [] args) throws IoException
{   FileReader fr = new FileReader ("abc.txt");
        int i = fr.read();
    while (i != -1)
  {  Sysout ((Char) i); // typecasting as i is unicode
        i = fr.read(); } // prints all the characters from file.
  }
```

eg:: Class Test{

```
  public Static void main (String [] args) throws Exception
{   File f = new File ("sachin.txt");
    FileReader f1 = new FileReader (f);

    Char [] ch = new Char [ (int) f.length() ];
     // f.length() is long so typecasting to int, making
        char array of the no. of characters in the file.

    f1.read (ch); // reads and stores chars inside array
  for (char data : ch)
        Sysout (data); // accessing the characters from array
    f1.close();
  }          // Closing the resources
}
```

Usage of FileWriter and FileReader is not recommended because of following reasons:

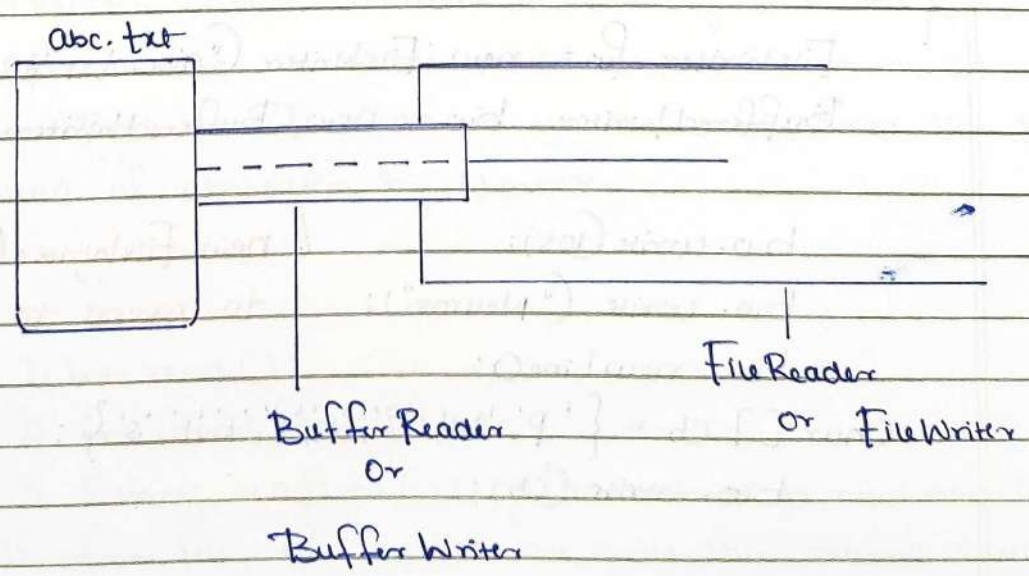
1. While writing data by FileWriter compulsory we should insert line seperator ('\n') manually which is a big headache to programmer.
2. While reading data by FileReader we have to read character by character instead of line by line which is not convinient.
   eg:: we need to search for a 10 digit mobile no. present in file called 'mobile.txt', Since we can read only characters just to search one number 10 searching is required and to search 10,000 mobile numbers we need to read 10cr times, so performance is very low.
3. To overcome these problems we should go for "BufferedReader" and "BufferedWriter."

Buffered Writer:

By using BufferedWriter we can write the character data to the file. It can't communicate with the file directly, it can communicate only with writer object.

Constructors:

* BufferedWriter bw = new BufferedWriter (Writer w);
* BufferedWriter bw = new BufferedWriter (Writer w, int size);



Which of the following declarations are valid?

1. new BufferedWriter ("cricket.txt"); // Invalid
2. new BufferedWriter (new File ("cricket.txt")); // invalid
3. new BufferedWriter (new FileWriter ("cricket.txt")); // valid
4. new BufferedWriter (new BufferedWriter (new FileWriter ("cricket.txt"))); // valid

Methode :

1. write (int ch);
2. write (char [] ch);
3. write (String s);
4. flush();
5. Close();
6. newline(); → for inserting a new line character to the file.

Q1. When compared to FileWriter which of the following capability is available as a method in BufferedWriter.

Ans Inserting new line character → newLine();

eg:: import java.io.*;

```
Class Test{
public static void main (String [] arge)
{
    FileWriter fw = new FileWriter ("abc.txt");
    BufferedWriter bw = new BufferedWriter (fw);

    bw.write (105);          // new FileWriter ("abc.txt", true);
    bw.write ("Neuron");     to append to file.
    bw.newLine();
Char [] ch = { 'P', 'W', 'S', 'K', 'I', 'L', 'L', 's' };
    bw.write (ch);
```

```
bw.newLine();
bw.write("unicorn");

bw.flush(); // make sure the operation is successful on file
bw.close(); // internally fw.close() will happen
    }
}
```

Note:

1. bw.close(); // recommended to use
2. fw.close(); // not recommended
3. bw.close(); fw.close(); // not recommended

Whenever we are closing BufferedWriter automatically underlying writer will be closed and we don't close explicitely

Buffered Reader:

This is the enhanced (better) reader to read character data from the file.

Constructors:

* BufferedReader br = new BufferedReader(Reader r);
* BufferedReader br = new BufferedReader(Reader r, int size);

BufferedReader cannot communicate directly with the file, it should communicate via some reader object. The main advantage of BufferedRead over FileReader is we can read data line by line instead of character by character.

Methods:

1. int read();
2. int read(char[] ch);
3. String readLine(); → It attempts to read next line and return it from file, if next line not avail then returns null.

eg:: import java.io*;

```
public class Test{
public static void main (String [] args) throws IOException
{
    FileReader fr = new FileReader ("abc.txt");
    BufferedReader br = new BufferedReader (fr);

    String line = br.readLine();
    while (Line != null)
    {   System.out.println (line);
        line = br.readLine ();  }

    br.close();  }
}
```

Note:

1. br.close(); // recommended
2. fw.close(); // not recommended
3. br.close(); fw.close(); // not recommended to use both.

Whenever we are closing BufferedReader automatically underlying FileReader will be closed, not required to close explicitly.

PrintWriter:

This is the most enhanced writer to write text data to the file. By using FileWriter and BufferWriter we can write only character data to the file but by using PrintWriter we can write any type of data to the file.

Constructors:

```
PrintWriter pw = new PrintWriter (String name);
PrintWriter pw = new PrintWriter (File f);
PrintWriter pw = new PrintWriter (Writer w);
```

Methods:

1. write (int ch);
2. write (char [] ch);
3. write (String s);
4. flush ();
5. close ();
6. print (char ch);
7. print (int i);
8. print (double d);
9. print (boolean b);
10. print (String s)
11. println (char ch)
12. println (double d)
13. println (int i)
14. println (boolean b)
15. println (String s)
16. Println (String s)

eg:: import java.io.*;

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

FileWriter fw = new FileWriter ("abc.text");
PrintWriter out = new PrintWriter (fw);

Out. write(100); // 100 unicode value will be written to file
Out. write ('\n');
Out. print\n (100); // "100" will be written to the file
Out. print\n (true);
Out. print\n ("Denis");
Out. flush();
Out. close (); }
}
```

In case of print(100), '100' as it is will be written, but write(100) write the unicode value 'd' to the file.

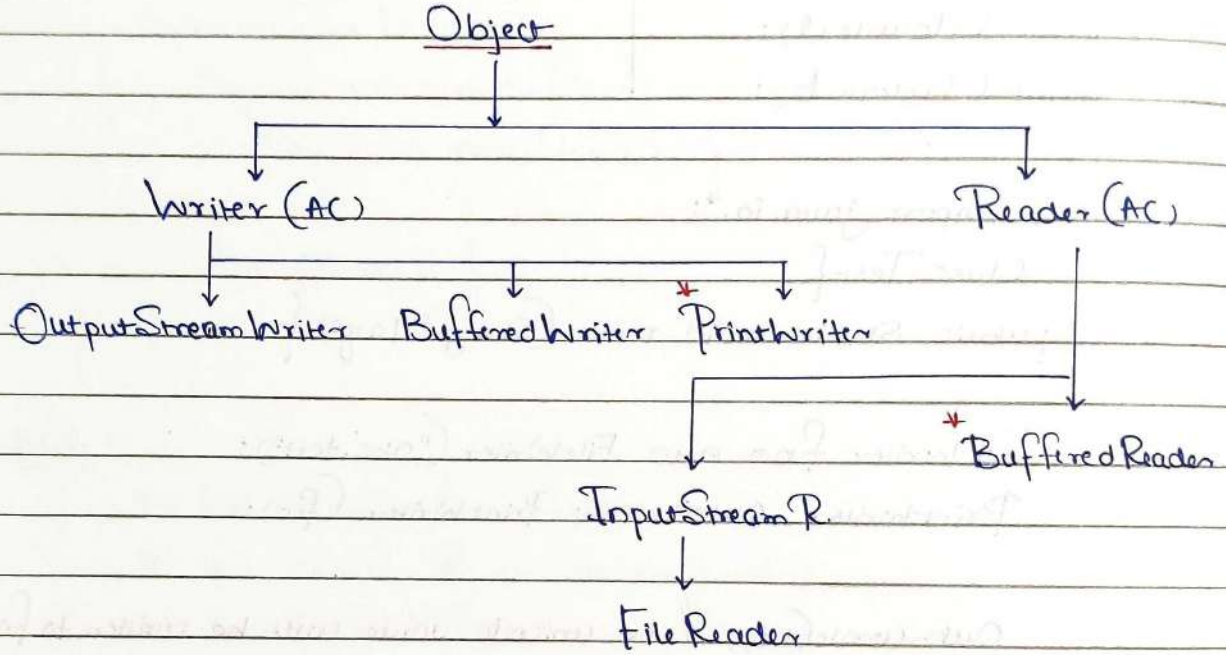
Note: The most enhanced Reader to read Character data from the file is "BufferedReader"

The most enhanced writer to write Character data to the File is "PrintWriter"

* In general we can use Readers and Writers to handle Character data, whereas we use InputStreams and OutputStreams to handle binary data (like images, audio files, video files etc).
* We can use OutputStream to write binary data to the File and we can use InputStream to read binary data from the File.

Character data → Reader and Writer

Binary data → InputStream and OutputStream



Program 1: Copy all contents from file1.txt, file2.txt to file3.txt

```
import java.io.*;
class Test{
  public static void main (String [] args) throws Exception
{
    PrintWriter pw = new PrintWriter ("file3.txt");
      // reading from first file and writing to file3.
    BufferedReader br = new BufferedReader (new FileReader
                                ("file1.txt"));
    String line = br.readLine();
```

```
while (line != null)
{   pw.println(line);
        line = br.readLine();}
// reading from second file and writing to file3
br = new BufferedReader (new FileReader ("file2.txt"));
line = br.readLine();
while (line != null)
{   pw.println(line);
        line = br.readLine();}

pw.flush() // to write all the data to file3.txt.
br.close()
pw.close();
Sysout-(" Open "file3.txt" to see the result");}
}
```

<u>Program 2</u>: Copy one line from file1.txt and from file2.txt to file3.txt.

```
{   PrintWriter pw = new PrintWriter ("file3.txt");
        // reading from first file
    BufferReader br1 = new BufferReader (new FileReader
                                ("file1.txt"));
    String line1 = br1.readLine();
        // reading from second file
BufferReader br2 = new BufferReader (new FileReader ("file2.txt"));
    String line2 = br2.readLine();

    while (line1 != null || line2 != null)
{       if (line1 != null)
    {       pw.println (line1);
            line1 = br1.readLine();}
```

```
    if (line2 != null)
    {   pw.println (line2);
        line2 = br2.readLine();  }
  }

    pw.flush();  // to write all data to file3.txt.
  br1.close();
  br2.close();
  pw.close();  }

}
```

<u>Program 3:</u> W.A.P to remove duplicate from the file

```
{ BufferReader br = new BufferReader (new FileReader ("input.txt"));
  PrintWriter pw = new PrintWriter ("output.txt");

    String target = br.readLine();
    while (target != null)
  {   boolean isAvailable = false;
      BufferReader br1 = new BufferReader (new FileReader
            ("output.txt"));
      String line = br1.readLine();
  // Control comes out of loop in smooth fashion without break.
      while (line != null)
  {  // if matched, control should come out of block with break
      if (line.equals (target))
    {     isAvailable = true;
            break;  }

        line = br1.readLine();  }
```

```
    if (isAvailable == false)
    {   pw.println(target);
        pw.flush();}

    target = br.readLine();}

    br.close();
    pw.close();}
}
```

<u>Program 4</u>: Write a program to perform extraction of mobile no. only if there are no duplicate.

```
{ BufferedReader br = new BufferedReader (new FileReader
                        ("input.txt"));
  PrintWriter pw = new PrintWriter ("output.txt");
    String target = br.readLine();

    while (target != null)
  {   boolean isAvailable = false;
    BufferedReader br1 = new BufferedReader (new FileReader
                        ("data.txt"));
      String line = br1.readLine();

      while (line != null)
    {   if (line.equals(target))
      {   isAvailable = true;
            break;}
          line = br1.readLine();}
```

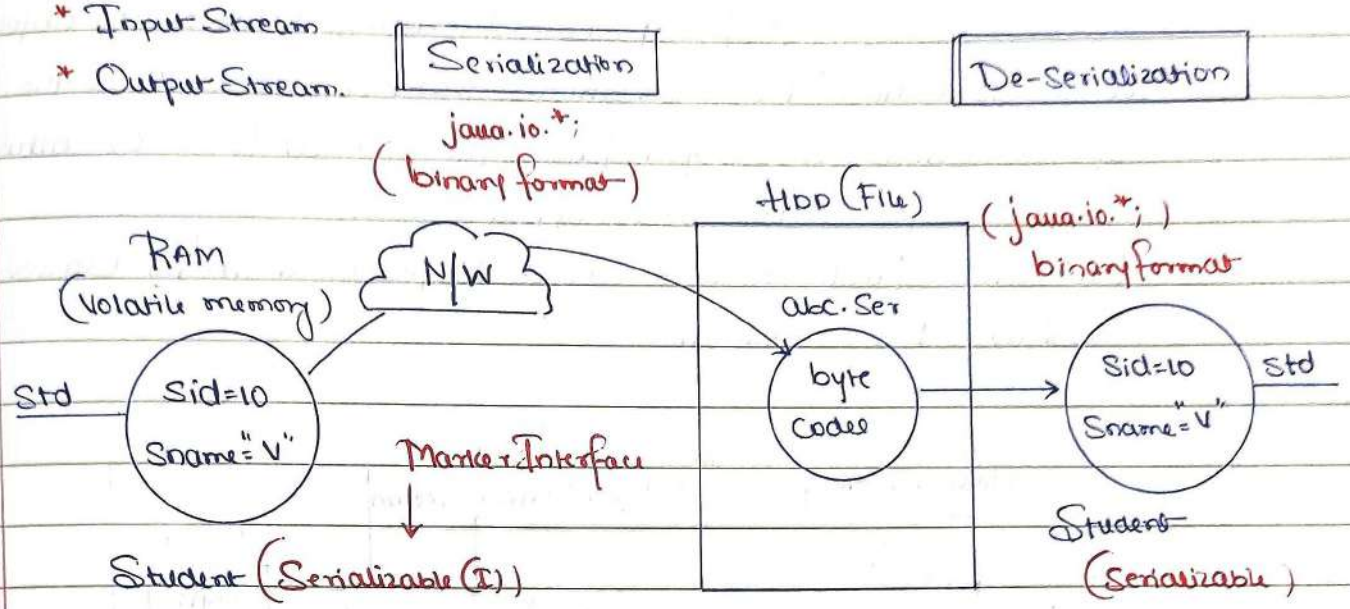
```
if (isAvailable == false)
{ pw.println (target);
  pw.flush(); } // write all the data if false.

target = br.readLine(); }

br.close();
pw.close(); }
}
```

Program 5: W.A.P to read the data from file and identify which data is larger in length. (data is string).

```
{ BufferedReader br = new BufferedReader (new FileReader
                        ("data.txt"));
  String data = br.readLine();
  int maxLength = 0;  String result = " ";
  while (data != null)
  { int resLength = data.length();
    if (maxLength < resLength)
    { maxLength = resLength;
      result = data; }

      data = br.readLine(); }

  System.out.println ("The max Length String is" + result);
  System.out.println ("The max Length of String is" + maxLength);
  }
}
```

Binary Type of data.

* Input Stream
* Output Stream.

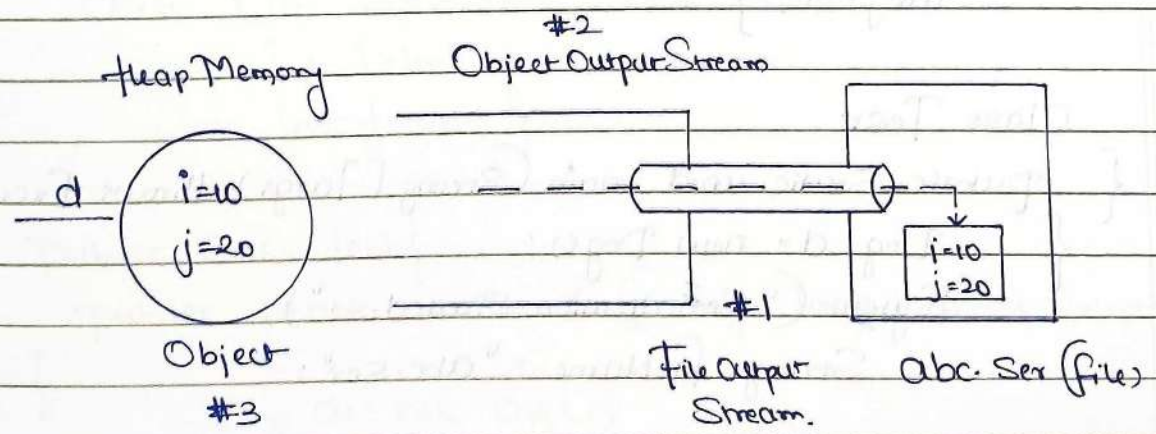


Key Points:

1. Object Should have the facility of transportation.
2. Object Should be Supported to Store inside file System. (Using Streams).

Serialization:

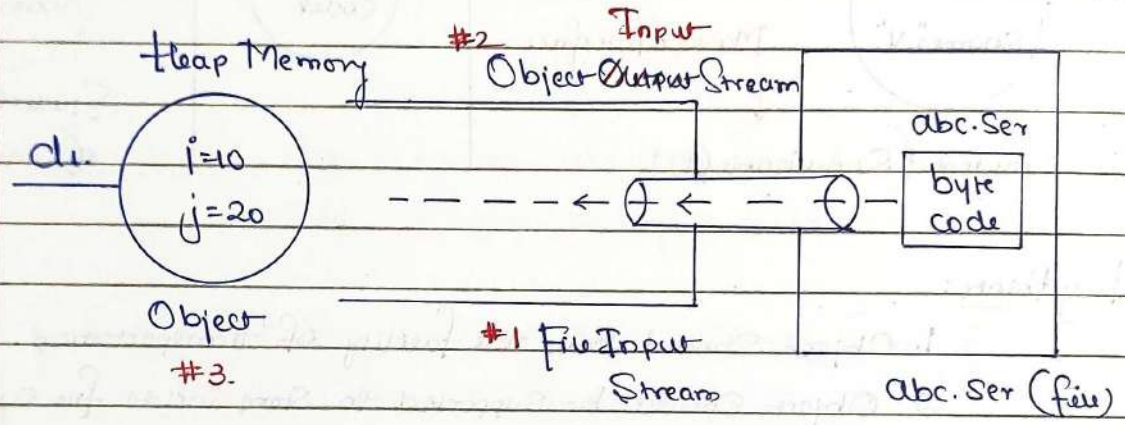
* The process of saving or writing State of an object to a file is called "Serialization" but/or in other words it is a process of converting an object from "java Supported form" to either network supported form or file supported form.
* By using "FileOutputStream" and "ObjectOutputStream" classes we can achieve Serialization process.



De-Serialization:

* The process of reading state of an object from a file is called "DeSerialization", in other words it is the process of converting an object from file supported form or network supported form to java supported form.

* By using "FileInputStream" and "ObjectInputStream" classes we can achieve De-Serialization.



→ eg:: import java.io.*;

```
Class Dog implements Serializable {

    Static { Sysout ("Static block gets executed"); }

    Dog (){ Sysout ("Object is created"); }

        int i=10;
        int j=20; }

Class Test
{   public Static void main (String [] args) throws Exception
    {       Dog d = new Dog();
            Sysout ("Serialization Started...");
                String fileName = "abc.ser";
```

```
FileOutputStream fos = new FileOutputStream (fileName);
ObjectOutputStream oos = new ObjectOutputStream (fos);
  oos.writeObject(d);
Sysout ("Serialized object reference is :"+d); // reference is
Sysout (" Serialization ended...");            "Dog@byk292"

// To pause the execution till we press some key from keyboard
System.in.read();

Sysout ("De-serialization Started...");
FileInputStream fis = new FineInputStream ("abc.ser");
ObjectInputStream ois = new ObjectInputStream (fis);
    Object obj = ois.readObject();
  Dog d1 = (Dog) obj; // Object Created but Constructor not called
    Sysout (" De-Serialised object reference is:" +d1);
    Sysout ("De-Serialized process over"); } // refrence is
}           // Object creation happens internally      "Dog@pk973"
```

→ eg::
```
import java.io.*;
  Class Dog implements Serializable {
        int i=10;
        int j=20; }   // Class whose object to seralize must
                         Implement Serializable

  Class Car implements Serializable {
        int i=100;
        int j=200; }

Public Class Test{
  public static void main (String [] args) throwes Exception
{
        Dog d1 = new Dog();
```

```
    Car c1 = new Car();
FileOutputStream fos = New FileOutputStream("abc.ser");
ObjectOutputStream oos = new ObjectOutputStream (fos);
    oos.writeObject(d1); // the order in which serialized, the
    oos.writeObject(c1); same order it should be de-serialized

FileInputStream fis = new FileInputStream("abc.ser");
ObjectInputStream ois = new ObjectInputStream (fis);
    Dog d2 = (Dog) ois.readObject();
    Car c2 = (Car) ois.readObject(); // order of de serialization
  }                                   should be maintained
}
```

Note:

* We can perform Serialization only for Serializable objects.
* An object is said to be serializable if and only if the corresponding class implements Serializable interface.
* Serializable interface present in java.io package and does not contain any abstract methods. It is a marker interface, the required ability will be provided automatically by JVM
* We can add any no. of objects to the file and we can read all those objects from the file but in which order we wrote objects in the same order only the objects will come back. The order is important.
* If we try to serialize an object we get (non-serializable object) then we get Runtime Exception : "NotSerializableException".

Transient Keyword:

* transient is the modifier applicable only for variables, but not for classes and methods.
* While performing serialization if we don't want to save the value of a particular variable to meet security constraint such type

of variable, then we should declare it "transient".

* At the time of serialization JVM ignores the original value of transient variable and save default value to the file.
* That is transient means "not to serialize".

eg:
```
import java.io.*;
Class Dog implements Serializable {
        int i =10;
        transient int j = 20; }

Public Class Test{
    Public Static void main (String [] args) throws Exception
{
    Dog d1 = new Dog();
    FileOutputStream fos = new FileOutputStream ("abc.ser");
    ObjectOutputStream oos = new ObjectOutputStream (fos);
        oos.writeObject (d1);

    FileInputStream fis = new FileInputStream ("abc.ser");
    ObjectInputStream ois = new ObjectInputStream (fis);
        Dog d2 = (Dog) ois.readObject ();

    Sysout (d2.i + "\t" + d2.j); } // 10 0
}
```

Static v/s transient: Static variable is not part of object state hence they won't participate in serialization because of this declaring a static variable as transient there is no use.

eg:
```
{
    Static transient int i=10;
        int j=20; }
```
Output : 10 20 (after seriaiz. and deserialization

final v/s transient: final variables will be participated into Serialization directly by their values. Hence declaring a final variable as transient there is no use. (The compiler assigns the value to final variable).

eg:: final int x = 10;
int y = 20;
System.out.println(x); System.out.println(y);

eg:: { int i=10; Output: 10 20
transient final int j=20; }

Declaration Outputs:

1. { int i=10; int j=20; } Output: 10 20

2. { transient int i=10; int j=20; } Output: 0 20

3 { transient int i=10; transient static int j=20; } Output: 0 20

4. { transient final int i=10; transient int j=20; } Output: 10 0

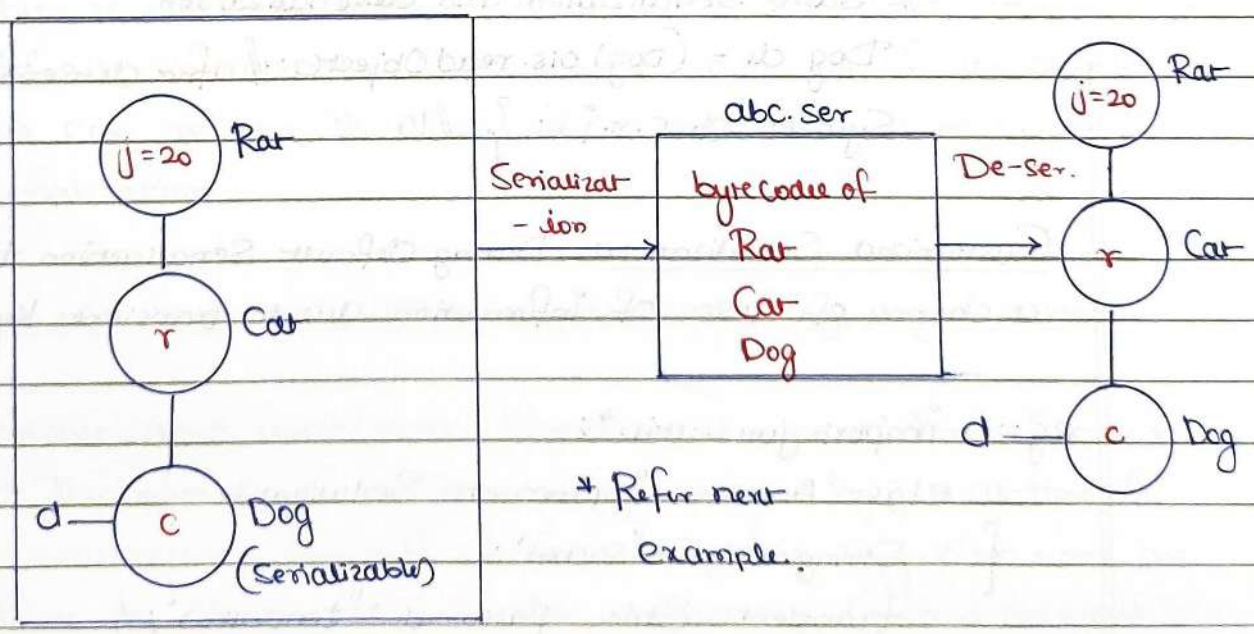
5. { transient final int i=10; transient static int j=20; } Out: 10 20

Note: We can serialize any number of objects to the file but in which order we serialized in the same order only we have to deserialize, if we change the order, it results in "ClassCastException".

eg:: { Dog d1 = new Dog();
Cat c1 = new Cat();
Rat r1 = new Rat();
FileOutputStream fos = new FileOutputStream("abc.ser");
ObjectOutputStream oos = new ObjectOutputStream(fos);

```
oos.writeObject(d1);
oos.writeObject(c1);
oos.writeObject(r1);
// if we don't know the order of serialization then:
FileInputStream fis = new FileInputStream("abc.ser");
ObjectInputStream ois = new ObjectInputStream(fis);
    Object obj = ois.readObject();

if (obj instanceOf Dog)
  { Dog d = (Dog) obj; }

if (obj instanceof Car)
  { Car c = (Car) obj; }

if (obj instance of Rat)
  { Rat r = (Rat) obj; }
}
```

<u>Object graph in Serialization</u>



* Refer next example.

Object Graph

1. Whenever we are serializing an object, the set of all objects which are reachable from that object will be serialized automatically. This group of objects is nothing but object graph in serialization.
2. In the object graph every object should be serializable otherwise we will get runtime exception saying "NotSerializableEx."

```
eg:: Class Dog implements Serializable{
        Cat c = new Cat(); }

     Class Cat implements Serializable{
        Rat r = new Rat(); }

     Class Rat implements Serializable{
        int i=10; }

     Class Test{
        public static void main(String [] args) throws Exception
     {
        Dog d = new Dog();
        // Same dog code as previous examples
        // Same serialization and deserialization.
        Dog d1 = (Dog) ois.readObject(); // after de-serialization.
        System.out.(d1.c.r.i); } //10
```

<u>Customized Serialization</u>: During default serialization there may be a chance of loss of information due to transient keyword.

```
eg:: import java.util.*;
     Class Account implements Serializable
     {  String name = "Sachin";
        transient String password = "tendulkar"; }
```

```
public class Test{
    public static void main(String [] args) throws Exception
{
        Account acc = new Account();
        Sysout(acc.name +" " + acc.password);

        FileOutputStream fos = new FileOutputStream("abc.ser");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
            oos.writeObject(acc);

        FileInputStream fis = new FileInputStream("abc.ser");
        ObjectInputStream ois = new ObjectInputStream(fis);
            acc = (Account) ois.readObject();
        System.out.println(acc.name +" " acc.password);
    }
}
```

→ In the above code example before serialization Account object can provide proper username and password. But after De-serialization Account object can provide only username but not password. This is due to declaring password as transient. Hence doing default serialization there may be a chance of loss of info. We can recover this loss of information by using customized serialization.

We can implement customized serialization by using 2 methods.

1. private void writeObject(ObjectOutputStream os) throws Exception
→ This method will be automatically executed by JVM at time of serialization. Hence if we want to perform any extra work we have to define that in this method only. (prepare encrypted password and write enc. password seperate to the file).

(2.) private void readObject (ObjectInputStream is) throws Exception.
→ This method will be automatically executed by JVM at the time of de-serialization. Hence if we have to perform any extra work, we need to define that in this method only. (read encrypted password, perform decryption and assign decrypted password to the current object password variable.)

```
eg:: Class Account implements Serializable {
        String name = "sachin";
    transient String password = "tendulkar";

    private void writeObject (ObjectOutputStream oos) throws Exception
    { oos.defaultWriteObject(); // performing default serialization
      String epwd = "123" + password; // encryption
      oos.writeObject(epwd); // write data to file (abc.ser)
    }

    public void readObject (ObjectInputStream ois) throws Exception
    { ois.defaultReadObject(); // performing default serialization
      String epwd = (String) ois.readObject(); // decryption
      password = epwd.substring(3); } // writing data to object
    }

public class Test{
  public static void main (String[] args) throws Exception
  {
      Account acc = new Account();
      Sysout (acc.name + " " + acc.password);

    FileOutputStream fos = new FileOutputStream ("abc.ser");
      ObjectOutputStream oos = new ObjectOutputStream (fos);
        oos.writeObject (acc);
```

```
FileInputStream fis = new FileInputStream ("abc.ser");
ObjectInputStream ois = new ObjectInputStream (fis);
    acc = (Account) ois.readObject();
    Sysout(acc.name + " " acc.password); }
}
```

→ At the time of Account object Serialization JVM will check is there any writeObject() method in Account class or not.

→ If it is not available then JVM is responsible to perform Serialization (default serialization).

→ If Account class contains writeObject() method then JVM feels very happy and executes that Account class.

→ The same rule is applicable for readObject method also

eg:: 2. <u>To add another variable (transient) to previous example</u>

```
{   transient int pin = 4444; // loss of information

    public void writeObject (...) ... // Changes to make in
{   ..... Previous example.
        int encypin = 1111 + pin;
        oos.writeInt(encypin); } // for int type data

    public void readObject (...) ...
{ .... int encypin = ois.readInt();
        pin = encypin - 1111; }
}
```

Serialization w.r.t Inheritance

<u>Case 1</u>: If parent class implements Serializable then automatically every child class by default implements Serializable.

classmate
Date 13/3/23
Page

That is Serializable nature is inherting from parent to child. Hence even though child class doesn't implements Serializable, we can Serialize child class object if parent class implements Serializable interface.

```
eg:: Class Animal implements Serializable {
        int i=10; }

     Class Dog extends Animal{
        int j=20; }

     public Class Test{
       public static void main(String [] args)
{        Dog d = new Dog();
         // Serialization process..
         . . . . . .
         // De Serialization process (same as in previous Dog example)
         . . . . . .
         Dog d1 = (Dog) ois.readObject();
         Sysout(d1.i +" " d2.j); }   //10 20
}
```

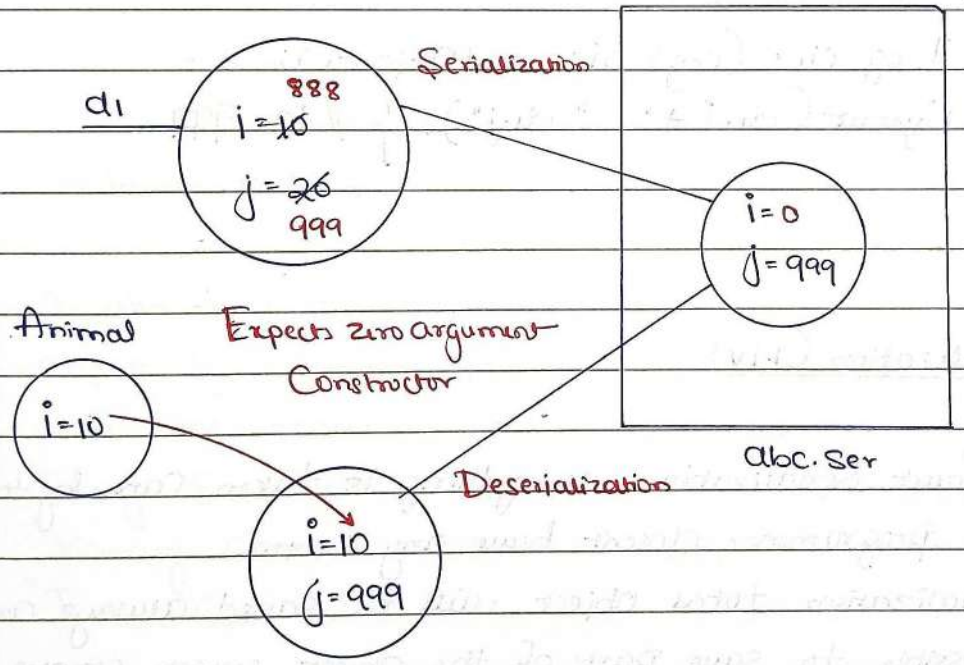
* Eventhough Dog class does not implement Serializable explicitly but we can Serialize Dog object because it's parent class already implements Serializable interface.
* Object class doesn't implement Serializable interface

<u>Case 2</u>:

* Even though parent class does not implement Serializable we can Serialize child object if <u>child class implements Serializable</u> interface.
* At the time of Serialization <u>Jvm ignores the values of instance</u>

variable which are coming from non Serializable parent then instead of original values JVM saves default values for those variable to the file.

* At the time of de Serialization JVM checks weather any parent class is non Serializable or not. If any parent class is non Serializable JVM creates a seperate object for every non serializable parent and shares it's instance variables to the current object.
* To create an object for non-serializable parent JVM always calls no argument constructor of that non-serializable parent hence every parent should contain no-arg constructor otherwise we will get runtime Exception "InvalidClassException".



eg:: import java.io.*;

```
Class Animal{
    int i=10;
    Animal(){ "Sysout("no arg Animal constructor") ; }
}

Class Dog extends Animal implements Serializable {
    int j=20;
```

```
    Dog(){ Sysout ("no arg dog constructor"); }
}

public class Test{
    public static void main (String [] arg) throws Exception
{       Dog d = new Dog();
        d.i = 888;
        d.j = 999;

        // Serialization takes place like previous example..
        .....

        // De-Serialization happens
        .........
        Dog d1 = (Dog) ois.readObject();
        Sysout (d1.i +"  " d1.j"); } // 10 999
}
```

Externalization (I.IV)

1. In default Serialization everything is taken care by the JVM and the programmer doesn't have any control
2. In Serialization total object will be saved always and it is not possible to save part of the object, which create perform-ance problems at certain points.
3. To overcome these problems we should go for externalization where everything is taken care by the programmer and JVM doesn't have any control.
4. The main advantage of externalization over Serialization is we can save either total object or part of the object based on our requirement.

5. To provide externalizable ability for any object compulsory the corresponding class should implements externalizable interface.
6. Externalizable interface is child interface of serializable interface.

Externalizable interface defines 2 methods:

1. public void writeExternal(ObjectOutput out) throws IOException
   This method will execute automatically at the time of serialization with this method, we have to write code to save required variables to the file.

2. public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException
   This method will be executed automatically at the time of de-serialization with this method we have to write code to save read required variable from file and assign to the current object.

→ At the time of deserialization JVM will create a seperate new object by executing public no-arg constructor on that object JVM will call readExternal() method.

→ Every Externalizable class should compulsory contains public no-arg constructor otherwise we get Runtime exception saying "InvalidClassException".

eg:: import java.io.*;
```
Class Demo implements Externalizable {
        String i;
        int k;
        int j;
    Demo(String i, int k, int j){
        this.i=i;
        this.j=j;
```

```
        this.k=k; }
    public Demo()
{   Sysout ("zero arg constructor"); }

    // performing serialization as required
    public void writeExternal (ObjectOutput out) throws IoException
{   Sysout ("call back method used while serialization");
        out.writeObject (i);        // only required objects are
        out.writeObject (j); }      serialized

    // performing de-serialization as per requirement
    public void readExternal (ObjectInput in) throws IOException,
        ClassNotFoundException {
    Sysout ("call back method used while de-serialization");
        i= (String) in.readObject();
        j= in.readInt(); }  ();
}

public class Test {
    public static void main (String [] args) throws Exception
{
    Demo d= new Demo ("nitin", 100, 200);
        // serialization started
    FileOutputStream fos= new FileOutputStream ("abc.ser");
    ObjectOutputStream oos= new ObjectOutputStream (fos);
        oos.writeObject (d);

        // de-serialization started
    FileInputStream fis= new FileInputStream ("abc.ser");
    ObjectInputStream ois= new ObjectInputStream (fis);
            d= (Demo) ois.readObject();
            Sysout (d.i +"  " +d.j +" " d.k); }
```

Output:

Serialization Started
Callback method used while Serialization
Serialization ended

De-Serialization Started
Zero-arg Constructor
Call back method used while de-serialization
nitin 100 0
Deserialization ended.

→ If the Class implements Externalizable interface then only part of the object will be saved.

→ If the class implements Serializable interface then the output is "nitin 100 200".

→ In externalization trancient keyword won't play any role, hence transient keyword not required

	Serialization	Externalization
1.	It is meant for default Serialization	It is meant for customized Serialization
2.	Everything is taken care by JVM, programmer don't have control.	Everything is taken care by programmer JVM does not have any control.
3.	Total object will be saved always.	We can save total object or partial based on our requirement.
4.	Relatively performance is low	Performance is high.
5.	Serializable (I) doesn't contain methods.	Externalizable contains 2 methods. (I).
6.	It is a marker interface	It is not a marker interface.
7.	Serializable class not necessary to contain public no-arg constructor.	Externalizable class should compulsorily contain public no-arg constructor.
8.	transient keyword play role in Serialization.	transient keyword don't play any role in externalization

SerialVersionUID:

* To perform Serialization and de-serialization internally JVM will use a unique identifier, which is SerialVersionUID.
* At the time of Serialization JVM will save SerialVersionUID with object.
* At the time of de-serialization JVM will compare SerialVersionUID and if it matches then only de-serialization can be done or else results in runtime exception saying "InvalidClassException".

Process in depending in default SerialVersionUID are:

→ After Serializing object if we change the class file then we can't perform de-serialization because of mismatch in serialversionUID of local class and serialized object in this case at the time of deserialization we will get runtime exception saying "Invalid Class Exception"

→ Both sender and reciever should use the same version of JVM if there is any incompatability in JVM versions then reciever will be unable to deserialize because of different SerialversionUID, in this case reciever will get "Invalid Class Exception".

→ To generate serialVersionUID internally JVM will use complex Algorithm which may create performance problems

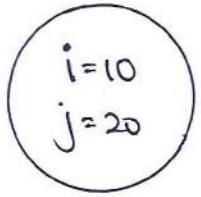
We can solve above problems by configuring our own SerialVersionUID

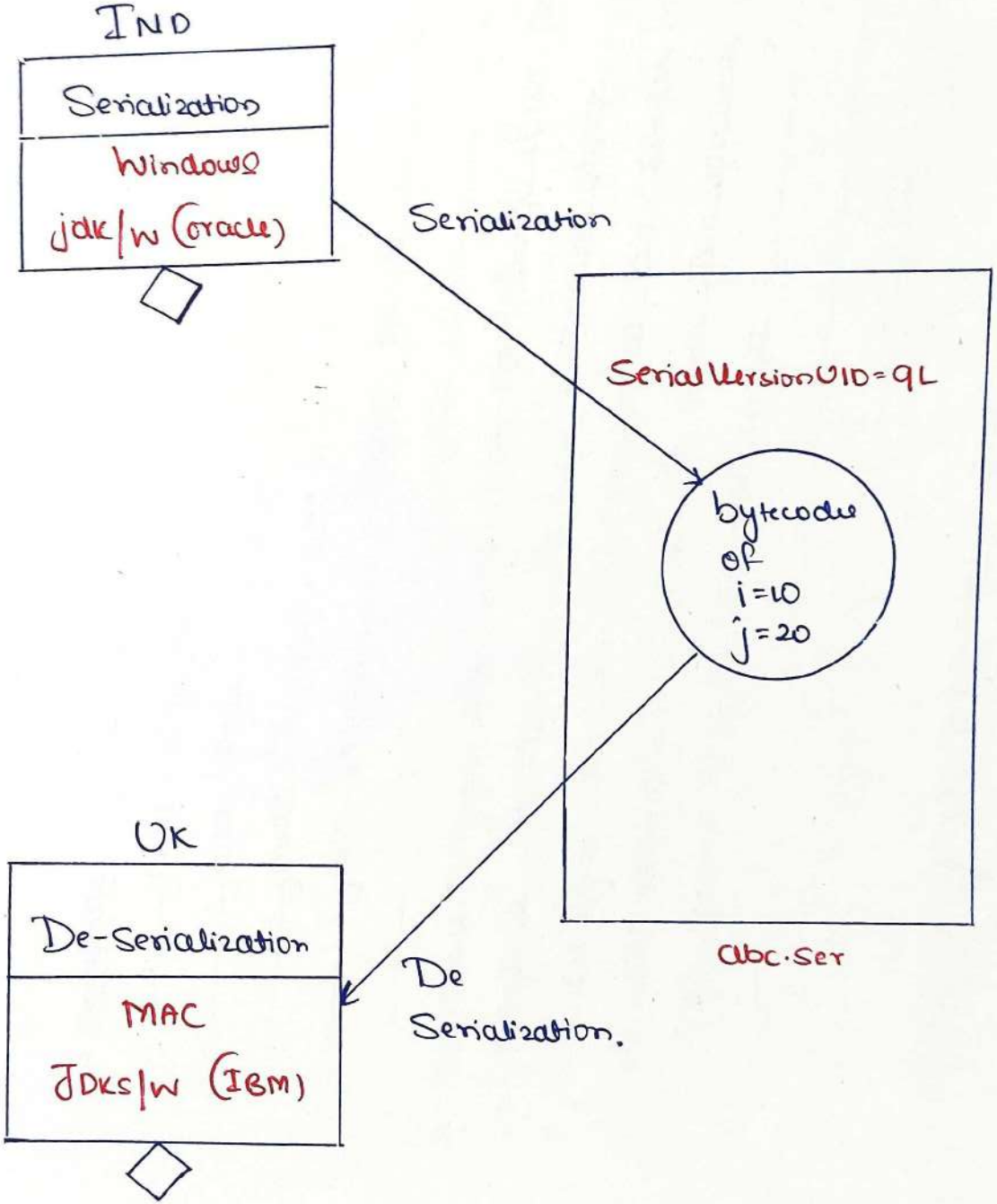
eg::

```
import java.io.Serializable;
Public Class Dog implements Serializable{
    Private static final long SerialVersionUID = 1L;
        int i=10;
        int j=20; }
```

Sender

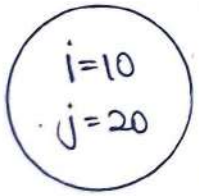
```
Class Dog implements Serializable
{ Private final Static long
    SerialVersion UID = 9L;
        int i=10;
        int j=20;
}
```





Reciever

```
Class Dog implements Serializable
{ Private final Static long
    SerialVersion UID = 9L;
        int i=10;
        int j=20;
}
```



```
import java.io.*;
public class Sender{
  public static void main (String [] args) throws IOException
{   Dog d= new Dog();
    FileOutputStream fos = new FileOutputStream ("abc.ser");
    ObjectOutputStream oos = new ObjectOuputStream (fos);
      oos.writeObject(d); }
  }

import java.io.*;
public class Reciever{
  public static void main (String [] args) throws IOException,
    ClassNotFoundException
{   FileInputStream fis = new FileInputStream ("abc.ser");
    ObjectInputStream ois = new ObjectInputStream (fis);
      Dog d2 = (Dog) ois.readObject();
      System.out.println(d2.i+" " + d2.j); }
}
```

<u>Commands:</u>

```
javac Dog.java
java Sender
javac Dog.java
java Reciever.
```

<u>Output</u>: 10 20

→ In the above program after serialization even though if we perform any change to Dog.class file we can deserialize object.

→ We can configure our own SerialVersionUID, both Sender and reciever not required to maintain the same JVM versions.

→ Some IDES generate explicit SerialVersionUID

String Tokenizer

* It is a part of java.util package
* It is used to split the entire string into multiple tokens based on the delimiter we supply.

```
eg:  String data = "Sachin ramesh tendulkar";
     StringTokenizer stk = new StringTokenizer(data);
                    // splits the string at empty spaces
     StringTokenizer stk = new StringTokenizer(data, "@");
                    // splits at '@'.
```

* Other Methods:
  - → public boolean hasMoreTokens();
  - → public String nextToken()

```
eg:: import java.util.*;
     class Test{
   public static void main(String [] args)
{
   StringTokenizer stk = new StringTokenizer("Sachin $ ramesh $
                        tendulkar", "$");
     System.out.println(stk);
     int tokens = stk.countTokens();
     System.out.println(tokens);
   while (stk.hasMoreTokens())
   {  String data = stk.nextToken();
        System.out.println(data); }
   }
}
```

Output: StringTokenizer @79sb29
3
Sachin
ramesh
tendulkar