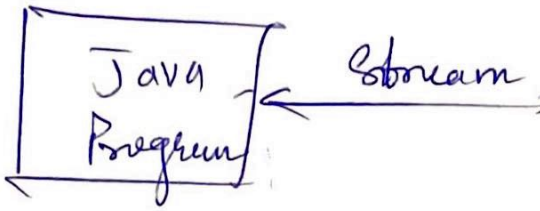
Stream IO

(reading & writing) are handled by streams

Stream is a connection b/w Java Program & destination/source.



Stream is a connection b/w Java Program & data source/sink & it is specific.

- Input stream ⇒ To read data from source we use i/p stream
- Output stream ⇒ To write data to the destination, we use o/p stream.

While working with stream, there are 3 operations involved.

1) Open stream
2) Read/write data depending on open stream if it is input/output.
3) Close stream. (sockets/filehandlers ⇒ OS)

Closing streams free up system resources & this would help in avoiding resource leaks.

Ex:-

```
FileInputStream in = null;
try {
    in = new FileInputStream (filename); // open stream
    // read data.
} catch (FileNotFoundException e) {
    ...
} finally {
    try {
        if (in != null)
            in.close();
    } catch (IOException e) {...}
}
```

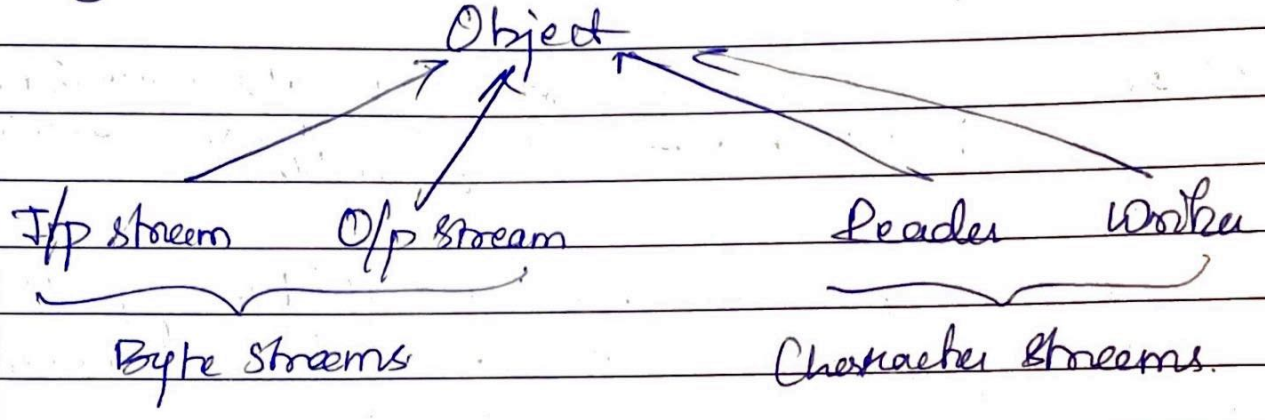
close stream should always be in a finally block to free up system resource

Name of the Experiment ..... Page No.:

Experiment No..... Date :

Shd be AutoClosable.

```
try(FileInputStream in = new FileInputStream(filename)){
    // Read data.
} catch(FileNotFoundException e){
    ...
} catch(IOException e){
    ...
}
```



Byte Streams

* Read/write raw bytes serially
* Character streams are built on Byte Stream.

Ip stream
↳ Base abstract class for all byte input streams
↳ To read data in groups of 8-bit bytes.

Read Operation

abstract int read() throws IOException
↳ Reads 1 bytes & returns b/w 0 & 255 (2^8-1)
↳ Returns -1 if end-of-stream detected
↳ '0' → 97. (implement will done in the Sub Classes)

Convenience method: int read(byte[] b, int offset, int length) throws IOExp
↳ Reads length # bytes into array starting at offset
↳ Returns byte read (or) -1 if end of stream detected
↳ Repeatedly invokes read.

All Read calls are blocking when no data is available.
i.e., the method will wait until the data is available.

Output stream

* Base abstract class for all byte o/p streams.
* To write data in groups of 8-bit bytes.

Write Opn's

abstract void write(int) throws IOExcepton
* Writes only LSByte i.e., because int is 32 bits.
(Least Significant)

Void write(byte[] b, int offset, int length) throws IO Exception
* for array of bytes.
* Writes length # bytes from array starting at offset.

void write(byte[] b) throws IOExcept° → write(b, 0, b.length)
previous version of write method by taking offset index as 0 ⇒ starting index.

Name of the Experiment ..... Page No.:

Experiment No..... Date :

Character streams

* Read / write characters.
* Built on top of byte streams. - Binary in nature.

Reader class & Writer class for character read & write

* Base abstract class for all character i/p streams.
* Read 16-bit char data in UTF-16 format.

read operations

1) int read() throws IOException.
   int read() throws IOException.

→ Reads 1 character & returns as int b/w 0 & $2^{16} - 1$ (65535)

→ Return -1 if end of stream detected.

2) abstract int read(char[] cbuf, int off, int len) throws IOException.

* Reads length # characters into array starting at offset.
* Return # characters read (or) -1 if EOS is detected

All read method would be blocked until some input data is available.

Writers

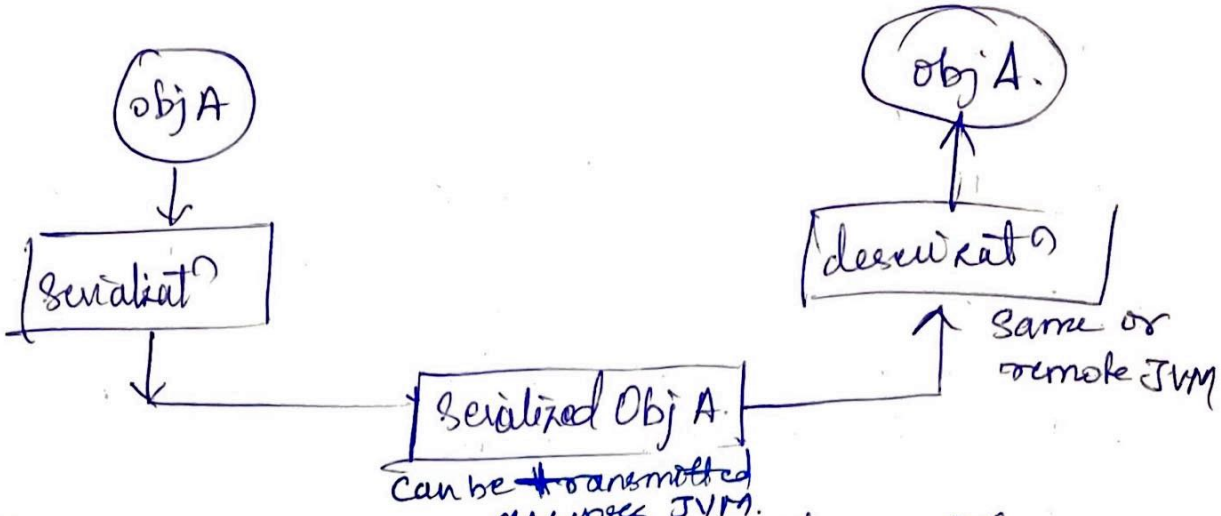
* Base abstract class for all character o/p streams
* Write 16-bit char data to stream.

void write(int c) throws IOException

Writes a single character (lower two bytes)

BIT

Serialization & Deserialization



Serialization - It is process of ~~writing~~ writing java objects to some destination -
Objects get saved as byte ~~strings~~ streams.

Serialization means

* Implements marker java.io.Serilizable
* ObjectOutputStream & ObjInputStream.

for serialization — for deserialization

These classes implement ~~obj~~ ObjectOutput & ObjectInput.

Ex:

```
ObjOutputStream out = new ObjectOutputStream(new
                      BuffOutputStream(new FileOpStream
                                       ("obj.ser")));
out.writeObject("Current Time & Date is");
out.writeObject(new Date());
```

Say, if you don't want an instance variable to be serialized, then use "Transient" keyword - [use at the time of variable declaration]
At this time, the variable's value will be skipped during serialization and during during de-serialization, the variable gets default value, say for an object = null, say if it is int = 0;

Name of the Experiment ..... Page No.:

Experiment No..... Date :

Deserialization:

ObjInputStream is = new ObjectInputStream(new BufferedInput
String str = (String) is.readObject(); // downcast from Object
Date date = (Date) is.readObject.

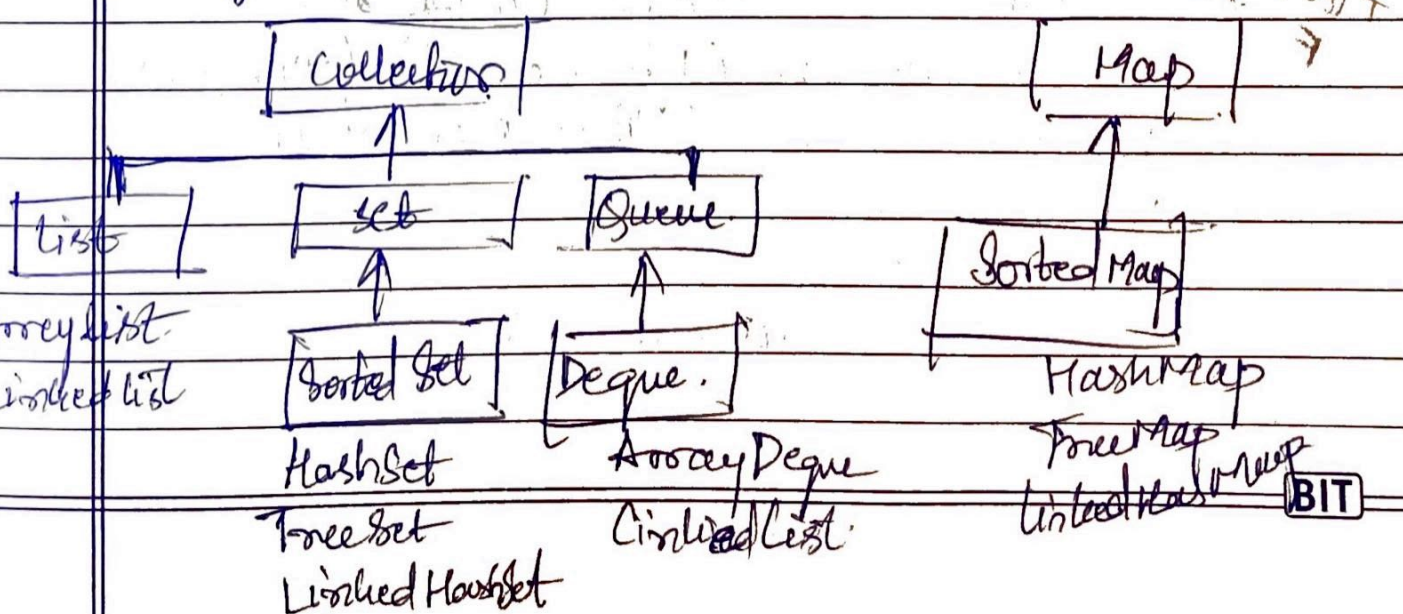
Collections framework – Data structures.

Array limitations
* Fixed no of values
* Not extendible
* Element search is expensive ~ linear.

* Serializable
* clone()
* Most allow null
* Not synchronized

Collections framework includes

* Interfaces
* Implementations, e.g., ArrayList
* Algorithmic classes e.g. Collections.



BIT

Collection Interface

* Collection of objects.
* Polymorphically provides maximum generality
* Some allow duplicates while others do not.
* Some are ordered while others are not.
* AbstractCollection → direct subclass.

public interface Collection<E> extends Iterable<E> {

- Basic Operations
- Bulk Operations
- Array Operations

}.

Basic Operations

→ It is need not be mandatory that the subclass has to be (or) it may not support, at that time the subclass would simply define empty method and would throw an "Unsupported Operation Exception".

T (or) F ← boolean add(E element); [Optional]
boolean remove(Obj element); // optional
boolean contains(Object element);
int size();
boolean isEmpty();
Iterator<E> iterator();

}.

Name of the Experiment ..... Page No.:

Experiment No..... Date :

Bulk Operations

```
public interface Collection <E> extends Iterable <E> {
    boolean addAll (Collection <? extends E> c); // optional
    boolean removeAll (Collection <?> c); // optional
    boolean retainAll (Collection <?> c); // optional
    boolean containsAll (Collection <?> c);
    void clear (); // optional
}
```

Array Operations

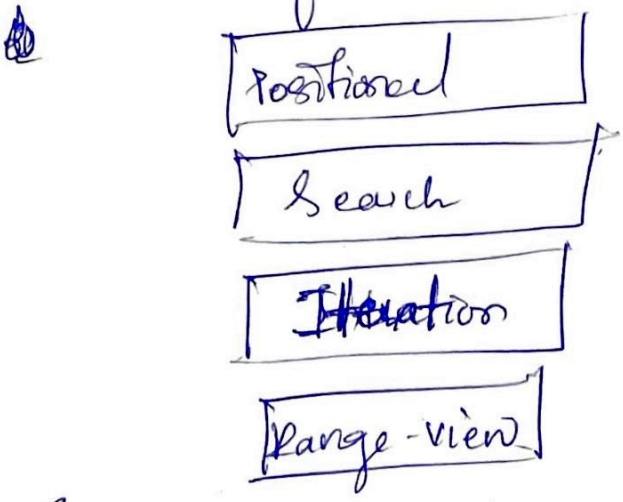
```
public interface Collection <E> extends Iterable <E> {
    Object[] toArray ();
    <T> T[] toArray (T[] a);
    // e.g., String[] a = c.toArray (new String[0]);
}
```

List Interface

* Useful when sequence/positioning matters
* Model resizable linear array with indexed access
* Zero based
* Can have duplicates

BIT

public interface List <E> extends Collection <E> {



}.

Positional Operations

```
{
    E get (int index);
    E set (int index, E element);
    void add (int index, E element);
    boolean add (E element);
    E remove (int index);
    boolean addAll (int index, Collection<? extends E> c);
}
```

Search Operations

```
{
    int indexOf (Object o);
    int lastIndexOf (Object o);
}
```

Name of the Experiment ..... Page No.:

Experiment No..... Date :

Iterator Operations

```
ListIterator<E> listIterator();
ListIterator<E> listIterator(int index);
```

Rangeview Operations

```
List<E> subList(int fromIndex, int toIndex);
```

Array List

A resizable array implementation of a list interface.

~ default capacity = 10, increases by 50%.
~ ArrayList(int initialCapacity) or ensureCapacity(int)
~ allows duplicates & nulls

Typical Uses

* simple iteration of elements
* fast random access ~ O(1)
* appending elements or deleting last element ~ O(1)

add & remove methods

* add (index, element)
  → following elements shifted right by one position
  → O(n)
* remove (index)
  → following elements shifted left by one position
  → O(n).

Search ~ contains() & indexOf()
* O(n)
* Use equals for checking and comparison -
* frequence search - consider Set implementation -

removeAll(collection)
* Collection . contains() on every element
  → if element is present → remove it
* Can be coorse than O(n²) → Quadratic.
  → O(n) ~ transsing current list
  → O(n) ~ contains() call (if collect, is List impl)
  → O(n) ~ element shift on remove().
* retainAll() ~ same performance issue.

O(1) Methods
* size()
* isEmpty()
* set() & get()
* iterator() & listIterator().

Name of the Experiment ..... Page No.:

Experiment No..... Date :

Iterable

```
public interface Iterable<T> {
    Iterator<T> iterator();
    default void forEach(Consumer<? super T> action);
    default Spliterator<T> spliterator();
}
```

// Allows elements removal during Iteration //

List Iterator

We can additionally add & replace elements while iteration — bidirectional.

```
public interface ListIterator<E> extends Iterator<E> {
    void add(E e);
    void set(E e);
    void remove();
```

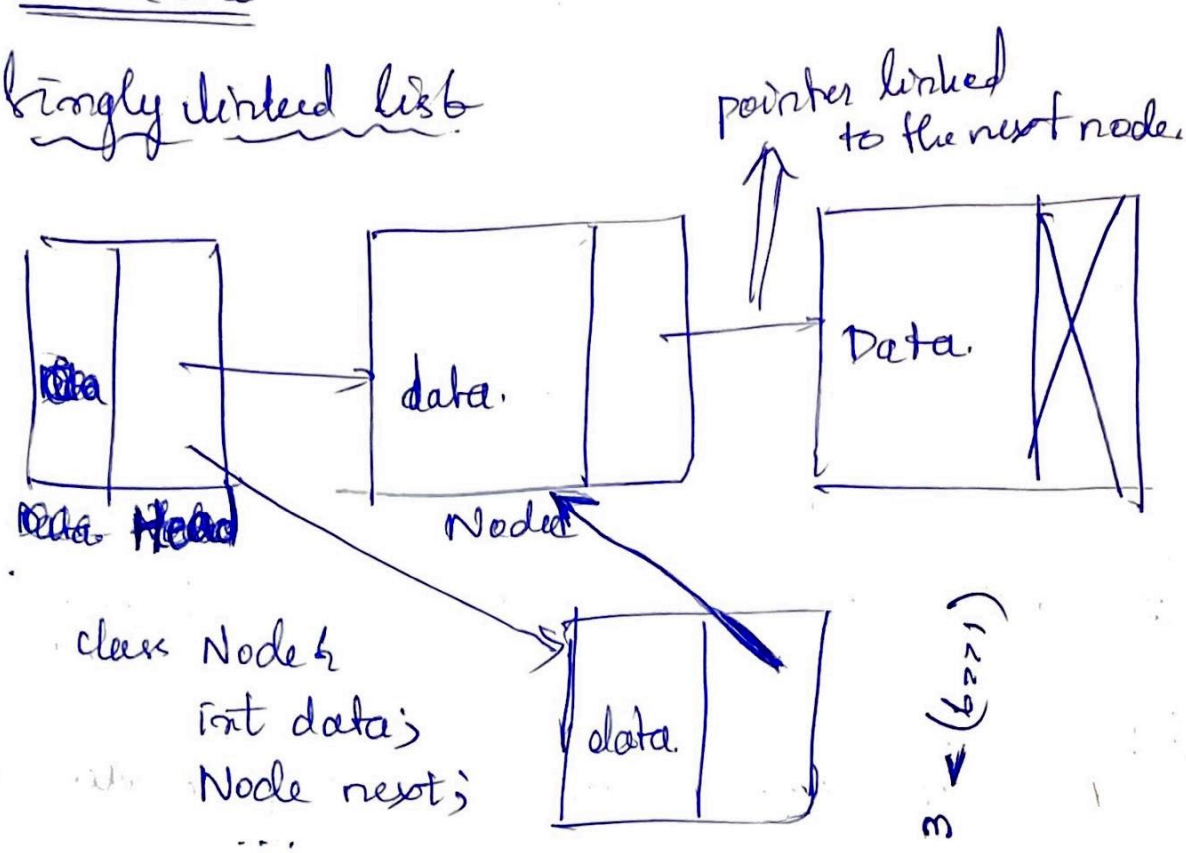
bidirectional access.

```
    boolean hasNext();
    E next();
    boolean hasPrevious();
    E previous();
    int nextIndex();
    int previousIndex();
}
```

BIT

Linked list

Singly linked list



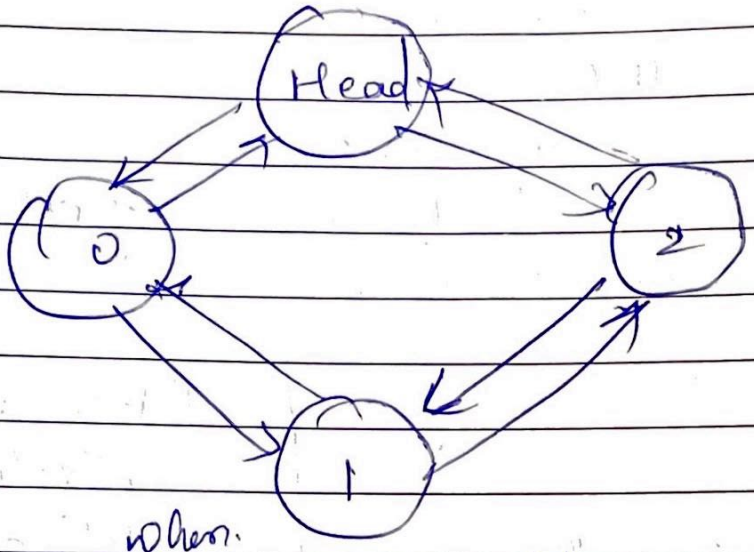

```
class Node {
    int data;
    Node next;
    ...
}

class LinkedList {
    Node head = new Node();
    void add (int data) {
        Node newNode = new Node(data);
        newNode.next = head.next;
        Head.next = newNode;
    }
}
```

Name of the Experiment ..... Page No.:

Experiment No..... Date :

Doubly linked list



When.
Typically used ~~in~~ frequent add/remove during iterat?

O(n) methods.

get(i)
add(i,e)
remove(i)
indexOf(object)
lastIndexOf(object)

} n/2 operat^ns

∴ java.util.LinkedList is a doubly linked list implementation of List & Deque interface.

Queue Interface

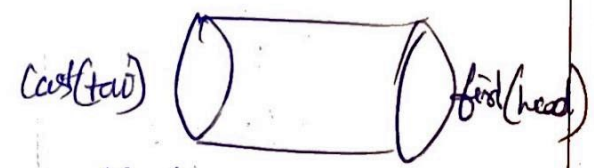
1st element, last element (most recently added).

* Useful when manipulate head & tail.
* tail ~ add
* head ~ remove/retrive
* Models FIFO.
* Can have duplicates.
* Can have nulls, but generally not supported.
* No indexed access.

	Throws except°	Returns special value
Insert	add(e)	offer(e) ~ false
Remove	remove()	poll() ~ null if que is empty
Inspect	element()	peek() ~ null if queue is empty

Deque ~ Double-Ended Queue

* Extends Queue
* Model FIFO & LIFO.
* Deque Implementaters
  - → ArrayDeque
  - → LinkedList
  - → ConcurrentLinkedDeque
  - → LinkedBlockingDeque.



methods are same as mentioned in the table with the suffix ~~last~~ (or) first.

getLast/ peekLast

Name of the Experiment ..... Page No.:

Experiment No..... Date :

Array Deque

a resizable array implementation of a Deque interface.

- models FIFO & LIFO.
- ArrayDeque() or ArrayDeque(int) or ArrayDeque(collection).

* Nulls are prohibited
* faster than linked list as a queue

Concurrency - Multithreading

* Browsers displaying images in Web pages
* Websites displaying ads
* Web crawlers.

Threads.

* Single sequential flow of control within a process.
* Light weight process
* Process ~ multiple threads
* Threads share process-wide resources, e.g. Memory
* Thread ~ has own PC, stack and local variables.

Benefits

* Exploiting multiple processors
* Allow loosely coupled designes
* Better throughput even in single CPU machines.

Types

→ Daemon
  * Background threades for tasks such as garbage collⁿ
→ Non-Daemon.
  * Created within application
  * JVM creates a main thread to run main()
  * JVM will not terminate if at least one non-daemon thread is running.

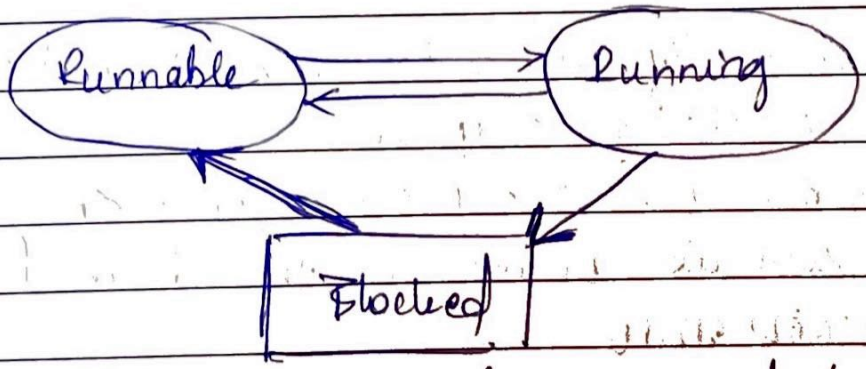
Name of the Experiment ..... Page No.:

Experiment No..... Date :

Lauching a thread.

* Create Task
  * Runnable task = new MyRunnable();
  * Runnable has exactly one method run();
* Create thread with Task.
  * Thread thread = new Thread (task); //NEW.
* Start Thread.
  * thread.start();
  * New call stack with run() is created for thread.

Thread Scheduler



TS

* Makes decisions on who runs and how long they run.
* No guarantee! ~ Never rely on scheduler behaviour

Default priority = 5 //

- Race Condition → Not thread safe

Synchronizat^n → Basically to Protect the mutable thread

The Synchronized block has 2 parts:

* Lock - objref.
* Code that needs to be guarded by lock.

```
Synchronized (lock) {
    // shared state
}.
```

LOCKING Mech

* Automatically acquired by thread entering Synchronized block (locking)
* Automatically release lock when thread exits synchronized block or due to exception
* At most one thread can acquire an object's lock.
* If lock is unavailable, then thread goes into BLOCKED STATE

locks ~ intrinsic (or) monitor locks.

lock ~ object on which method is invoked.

Name of the Experiment ..... Page No.:

Experiment No..... Date :

Locks & Synchronized methods

Locks are per object and not per method.

* If lock is Unavailable → Can't access ANY of objects
* ~~Only one thread can access~~ synchronized methods
* No restriction on unsynchronized methods
* Static synchronized methods uses class obj for lock

* Co-ordinating Access to a variable,
  Synchronize wherever the variable is accessed.

Synchronization = mutual exclusion + reliable thread communication

* Guard same shared mutable variable with same lock.

Synchronized (in code)

↓

[JVM]

↓ memory barriers are put by JVM in order to make a co-ordination between the threads.

BIT

Volatile Variables

* Alternative way to ensure memory visibility
* Always stored in main memory – not register / cache.
* Used with shared mutable variables.

// Locking can guarantee both visibility and mutual exclusion while variables can only guarantee visibility //

Volatile : One thread writes & other read → no race condition –

Synchronized : race condition, e.g., check-then-act.

Automatic Variables

```
import java.util.concurrent.atomic.AtomicLong;

public class IDGenerator {
    private AtomicLong id = new AtomicLong();

    public long getNewId() {
        return id.incrementAndGet();
    }
}
```

Memory Visibility
Atomicity

* There is no lock system in here instead,
* They use compare & swap method.

Name of the Experiment ..... Page No.: ☐

Experiment No..... Date : ☐☐☐☐☐☐

→
```
public final long incrementAndGet(){
  for(;;){
    long current = get();
    long next = current+1;
    if(compareAndSet(current, next))
      return next;
  }
}
```

* non-blocking
* optimistic locking

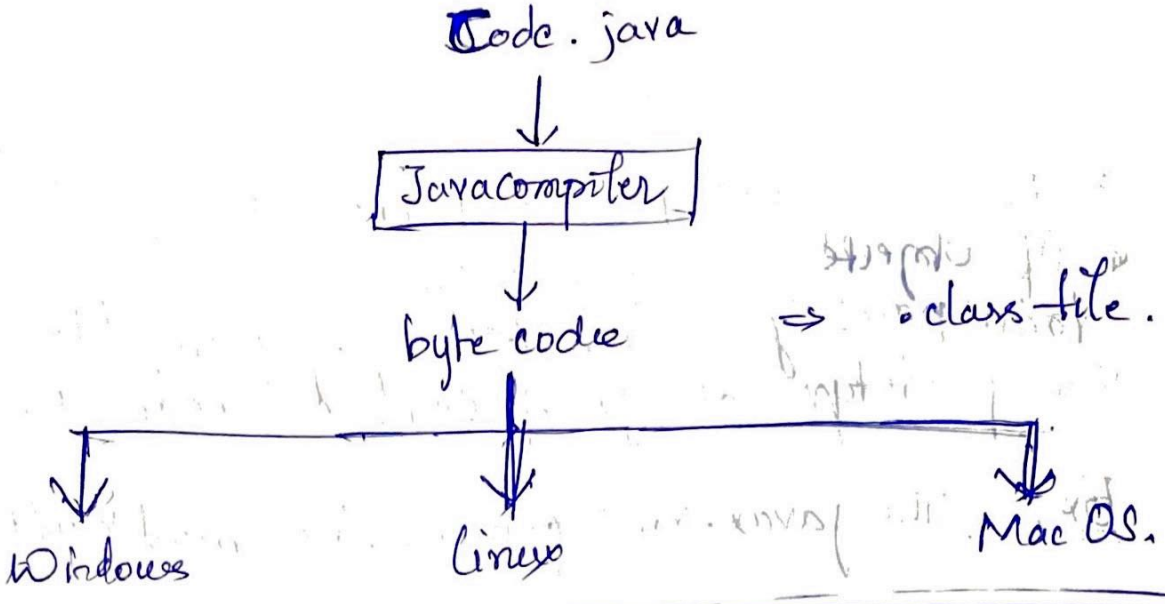
Atomic variable classes. (A → Atomic)(I - Integer) (B - Boolean) (L → Long)(R - Reference)

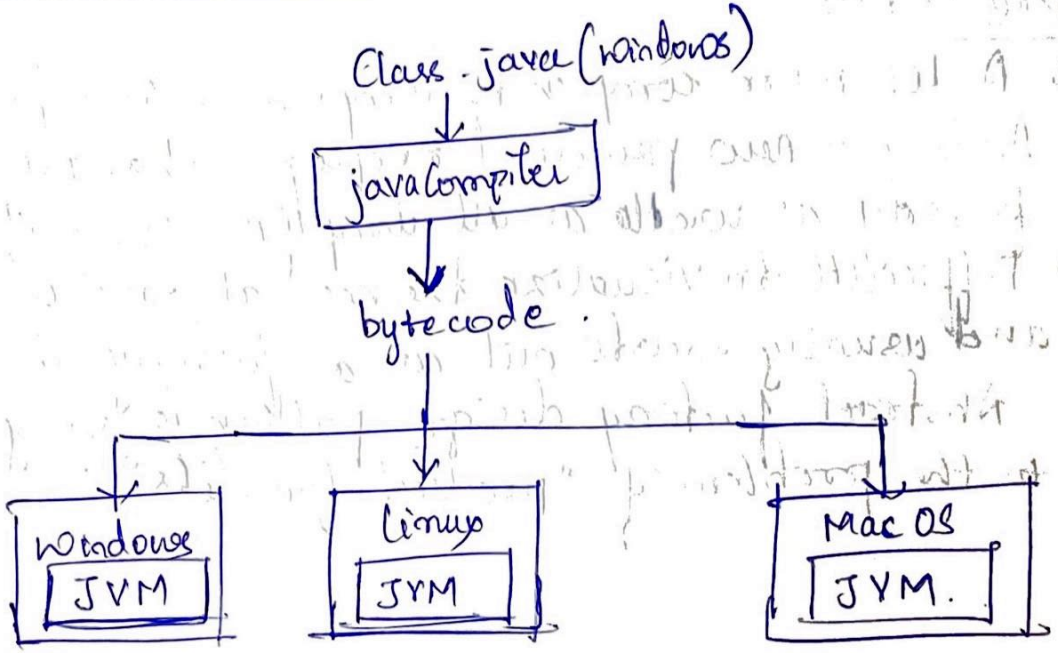
* Scalers
  → AtomicLong, AtomicIntegers, AtomicBoolean, AtomicReference.
* Arrays
  → AtomicLongArray, AtomicIntegerArray, AtomicReferenceArray.
* FieldUpdaters (FU) ⇒ ALFU, AIFU, ARFU
* Compound Variables.
  → AtomicMarkableRef
  → Atomic StampedRef.

BIT

SOP ("%-15s%03d%n", s1, x);

% = used as formattor
- = used for left indentation of the string
15s = denotes the string's minimum field width 15.
0 = pads the extra 0s in the integer
3d = 3 denotes integers minimum field width 3.
%n = newline


Code.java
JavaCompiler
byte code
=> .class file.
Windows
Linux
Mac OS.


Class.java (windows)
javaCompiler
bytecode.
Windows
JVM
Linux
JVM
Mac OS
JVM.

Concurrency / MultiThreading

Doing / performing multiple tasks at the same time is called as multi-threading.

* At there time, we were just doing the Sequential programming, most of the cases it never use to block, but in certain cases like read() in JavaIO, the tasks get blocked when data is not available to read.
* Blocking halts other tasks thus wasting CPU time.
* That is where the concurrent programming come into role, which helps us continue the execution, even when the exec' is blocked.
* In pre-OS era, Computers ran a single program as it was a very inefficient way of using a processor.
* Later on OS was evolved where-in, it allowed multiple processes to run concurrently
* Here multiple process as in
  - → processor get own memory, file handles etc;
  - → Concurrency is achieved through multi-tasking
  - → Multi-tasking, although it seems like parallelism, but it is actually the switching of CPU from one process to another. But that fast switching seems like the parallelism.

Thread

* Thread is nothing but the single sequential flow of control within a process.
* The process as in the light-weight process.
* A process can have multiple threads.
* Threads share process-wide resources, Ex - memory
* Threads have their own PCs, own stack and local variables.

advantages →

* Exploiting multiple processors thus achiving parallelism
* Allow loosely-coupled designed
* Better throughput even in single CPU machines in certain scenarios like when the process is BLOCKED.

thread types

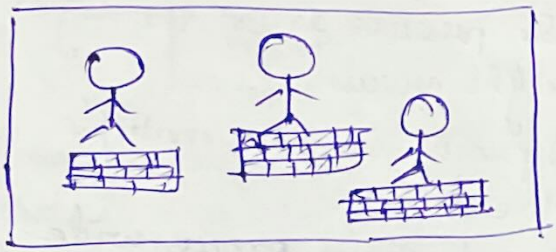
1) Daemon Thread

Daemon thread is basically the background thread that is used for tasks such as GC - Garbage Collections.

2) Non-Daemon:

* Non-Daemon threads are the one that are created from within the applicat°.
* Usually, if there are no threads, then there will always be a main thread that is created by JVM to run main() method.

* JVM will not terminate even if there is 1 non-daemon thread is running.



* Assuming that the box is a site and the workers are the process to complete 1 main process i.e, building.
* Basically, till we give some task or process, the workers will remain unused. Here workers are assumed as threads.
* So, when the work is given, the would go for it to complete 1 main process, parallely.
* In Java, thread is simply the instance of a class "Thread" which belongs to java.lang.Thread package. [or one of its subclasses].
* And the task is defined by a class that implement the interface "Runnable" which belongs to java.lang.Runnable package.

Thread ⇒ Worker
Runnable ⇒ Work.

Launch a thread.

1) Create task.

```
Runnable task = new MyRunnable();
```

Runnable: Interface type (just instantiation) | task: ref obj | new MyRunnable(): user-defined class that implements Runnable Int

* Here Runnable has got only 1 method i.e, run();
* Here we have to define the task in the run method.

2) Then create thread with Task

* `Thread thread = new Thread(task);`

Now, the thread is in a new state. "NEW" state

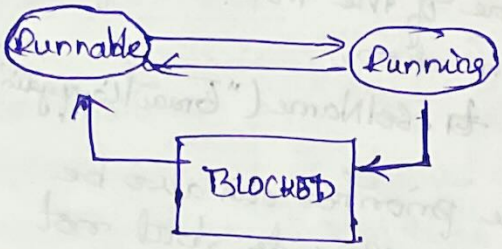
3) Next we start the Thread.

* thread.start();
* Nothing happens until we start a thread.
* When we start it, a new call stack will be created for this, and the run() method will be pushed on to the stack.

[Every thread will get its own call stack]

* And now the thread is said to be in a state "RUNNABLE" state (thread is ready to run)
* Runnable doesn't mean that the run() method is getting executed and for that, thread need to get into a state called "RUNNING".
* There is something called Thread scheduler, that moves the thread from Runnable to Running state.

Thread Scheduler



* Thread Scheduler is the component that is responsible for switching the thread from Runnable to Running state.
* Once the thread is into running state, the call stack of that specific thread becomes active and the method on top of the stack will get executed, say if it is run() method, it gets executed.
* Similarly, the Thread Scheduler is also responsible for switching the thread from Running to Runnable state, so that it can move some other thread from runnable state to running state.
* Basically, "Thread Scheduler" is the component that is responsible for switching b/w the threads.
* There is 1 more Blocle i.e., "BLOCKED" [The thread will usually get into RUNNABLE phase when the data is to be fetched say for example during Java IO when a thread is meant to read a file and the file has got no data into it to read]

* Usually a thread can get into Blocked state only from a Running state, similarly from Blocked state, it cannot get back to running state, instead, it has to go to Runnable state to get into running state again. Again it cannot get into Blocked state from Runnable state.
* Thread Scheduler is the one that makes decisions on who runs and how long they run
* There is no guarantee, how long the thread gonna run and when they gonna switch. Never relay on scheduler's behaviour.
* For a thread to enter a synchronised block, it needs to acquire a LOCK. If LOCK is not available, then the thread moves into BLOCKED state and waits until the locle is available.
[Synchronised block is just a special block of code]

"SLEEP" sleep()

⇒ To put the Thread on sleep mode, we just have to invoke sleep method which is a static method of class Thread.

* Thread.sleep();

More likely, when we invoke sleep method, it would throw an "except" called "InterruptedException", and we have to put it into try catch-block.

* There is an other enum class for Thread sleep ie.,
TimeUnit.SECONDS.sleep(3)

* Once the thread completes its all execution, it goes to "DEAD" state and gets terminated and we cannot restart by thread by using .start() method, if done, it throws an except called "IllegalThreadStateException".

Thread priority

How a thread can get access for its own object (?)

That is achieved by using
Thread.currentThread();
↳ This would give reference to the thread that is been currently running. → main() thread

Ex:- pub stat void main (Str[] args) {

SOP (Thread.currentThread());

}

related to security [recommended not to use]

O/p ⇒ Thread[main, 5, main]

- main → name of the thread
- 5 → thread priority. [default priority = 5]
- main → thread group

[Avoid thread group → To group threads and perform a funct in group.]

* you can actually set the name of the threads for information.
say t1.setName("BroadCastingJio");

* The priorities can also be given to threads, but not sure, if it gonna work as per the priority but there are high changes of a maximum priority thread can run.
The max priority = 10
It always depends on Thread scheduler how it gonna run threads with priority.
The Norm/default priority = 5
Minimum priority = 1

yield()

* This methods, stops currently executing thread for some time and will give chance to other thread that holds priority.
* It basically gives hint to CPU that it is willing to yield.

join()

This method allows the joined thread to complete the process or execution meanwhile other threads will wait until they complete their execution.

Ex - t2.join()

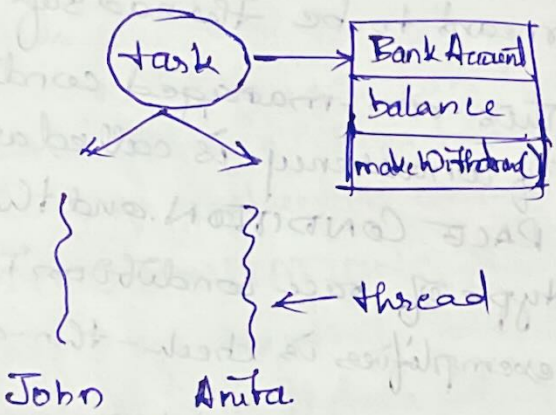
It means, other thread waits until the t2 dies. It has void type & throws InterruptedExcept.

Concurrency Hazard

Race condition.

The main complexity of a concurrency comes in when same object is shared around multiple threads and that object has some state i.e., data which can be modified by the threads.

Example



Here, let us consider John and Anita are couples and they have a joint account and let's say they have balance of $100.

Whenever people access bank application they usually get represented as threads. as shown in the image:

Here Bank Account is a task. and since it is a task it implements runnable interface. and Bank Account class has 1 instance variable called balance and also a method called makeWithdrawal(); which is invoked when a user is making a withdrawal.

```
void makeWithdrawal (int amount){
    if (balance >= amount){
        balance = balance - amount;
    }
}
```


This method checks if there is a balance in the account and if only yes, it gonna let to the withdraw so that there shouldn't be any over withdrawal of the money.

Now,
When a user tries accessing the bank application, at that time a bank account object will be created for that user's account and will be passed as an input to the thread the represents user.

Now, let us assume that both John and Anitha access the application at the same time. Since they both represent same bank account, a single object is created and is shared among two threads, i.e., 1 is for John and other 1 is for Anitha as input.

Now say, both John and Anitha wants to withdraw money of $75 at the same time i.e., they are using banking system at the same time hence here 2 threads gets created and say Thread scheduler picks John's thread at first.

John

Enters makeWithdrawal();
checks balance >= amount;
Since John wants $75 and is lesser than $100, the condition is passed.
Now, lets assume that Thread scheduler gets the John's thread to RUNNABLE state before the new balance is computed.

Anita

Now, the Thread Scheduler will bring Anita's thread into RUNNING state, now
Enters makeWithdrawal();
checks balance >= amount;
Since John has not withdrawn any money, the condition is passed and then the amount is withdrawn and new balance is computed.
balance = balance - amount;
balance = $25;

John

Now, John's thread has brought to running state as Anita's thread has completed execution and assuming that the balance is still $100, John will overdraw, which brings balance to -$50. So now the system is in bad state and the data got corrupted. Hence, the BankAccount obj is not thread safe

* BankAccount obj was not thread safe

↳ i.e., it was mutable state
↳ obj was shared b/w threads for accessing the methods that can mutate data.
↓
Since the mutable state was not well managed it is not meant to be thread safe.

This non-managed condition of concurrency is called as RACE CONDITION. and the type of race condition it exemplifies is check-then-act.

Sol? :- To avoid this, we have to make withdrawal() method as 1 unit called ATOMIC UNIT. i.e., if 1 thread enters make withdrawal(), it must be allowed to complete the process before any other thread enter the method. This behaviour is called as MUTUAL EXCLUSION. - Only 1 thread is able to enter a method, at a given instance of time.

Non-thread safe Obj → incorrect program → data corrupt.

Synchronization

As we know about the Race condition, to avoid that we have to make the whole method as 1 atomic unit.

```
void makeWithdrawal (int amount){
  if(balance >= amount){
    balance -= amount;
  }
}
```

atomic unit.

It happed because, the class was not thread safe.

To avoid this, we can use "inbuild Java block locking mechanism called "SYNCHRONIZATION". The duty of synchronization is to protect critical data which is shared and also mutable such as "balance variable".

Protect critical (shared mutable) data

Basically, the synchronized block has 2 parts :-

1) Lock  ⇒ which is basically the obj reference.

2) code that needs to be guarded by Lock.

```
synchronized (obj ref/Lock){
    //shared state
    // block of code.
}
```


Ex :-

```
public void go(){
  non-critical stuff;
  synchronized(this){
    critical statement 1;
    critical statement 2;
  }
  non-critical stuff;
}
```

this ⇒ ref to current object. serving as the lock.

LOCKING

* Every object will almost get 1 lock.
* Automatically acquired by thread entering synchronized block
* Automatically released when thread exits synchronized block (or) due to exception.
* At most one thread can acquire an object's lock.
* If lock is unavailable, then thread goes to BLOCKED state, once the lock is available gets back to RUNNING state and takes up the lock
* Again, it is decided by the Thread scheduler.

Note :- There can be multiple threads waiting for a lock.

* The built-in locks associated with the objects is called as intrinsic (or) monitor locks.

Hence, the method can be made as

```
synchronised void makeWith
  -drawal(int amount){
    if(balance >= amount){
      balance -= amount;
    }
}.
```

Here Lock ⇒ obj on which method is invoked. Hence there won't be any race condition and is thread safe.

Locks & Synchronised methods

Locks are per object and not per method.

So, it means is that, 1 thread has acquired object's lock, then no other thread can enter any of the object's synchronised methods.

* Lock Unavailable → Can't access ANY of object's synchronised methods.
* But they have no restriction on unsynchronised methods.
* Static Variables synchronised methods use class object for lock.
* The main reason we are using synchronisat° is to co-ordinate access to a shared variable.
* Synchronise everywhere where the shared variables are accessed

Ex:- getMethod()

```
public private int balance;
public synchronized int
getBalance(){
    return balance;
}.
```

Synchronizat° = mutual exclusion + reliable thread communicat°.

* Guard same shared mutable variable with same lock.

Java Memory Model.

As, out, there were multiple unpredictable problems in concurrency which actually lead to one of the concurrency hazard i.e., Race Condition. And then, we came up with Synchronizat° to overcome this issue. Here we made things in a more ordered way. Here JMM is one of the specificat° of Java that defines ordering. They are basically bunch of rules. - Related to multi-threading.

Reasons

1) Out-of-order actions.

We know about the race condition as it is, please go through the code -

```
public class BankAccount
implements Runnable {
    private int balance;
    public void int getBalance() {
        return balance;
    }
    public void run() {
        makeWithdrawal(75);
    }
    public synchronized void
    makeWithdrawal(int amount) {
        if (balance >= amount)
            balance = balance - amount;
    }
            // Race Condition
}
```

Here ~~reason~~ say both Anita and John tries to access the bank applicat' at the same time, and in here say, John's thread gets the access at first, meanwhile John's thread gets into withdrawal method and before updating the balance, the thread would get into RUNNABLE state.

Now when Anitha's thread tries to get the balance, it would obviously know the old balance since the John's thread went into RUNNABLE state before updating the balance and this leads to

the race condition and this is what actually known as out-of-order actions.

⇒ Multi-processor with Shared Memory.

This is the problem that are happening in the modern days. Consider that there are 2 processors and they have their own local cache for ~~for~~ periodically ~~pre~~ synced. each that is shared with a common memory.

Processor 1 (John)

* Executes makeWithdrawal() completely
* Flush new balance to local cash & to memory.

Processor 2 (Anita)

* Reads old balance from local cache as it would have not got updated updating at the time of getting a cache memory.

~~The~~ * The main reason for all this is lack of thread co-ordination

* Basically, No thread coordination is needed for performance (non-synchronised)
* Typically, threads do their own stuff

(19)

* Thread co-ordination / thread communication
every write is very expensive as we have to update the local cash and memory more often.

* Thread co-ordination is needed only when there is a shared and mutable data.

* Thread coordination usually becomes the program's responsibility when there are shared variables, not JVM's.

* Co-ordinating access to a variable is done by synchronizing everywhere where variable is accessed i.e., to make getMethod as synchronized as well.

```
public synchronized int getBalance() {
    return balance;
}
```

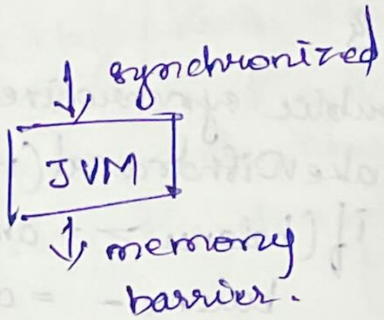
Here 2 actions i.e., John makeWithdrawal() and Anita's getBalance() have a "happens - before" relat'ship

i.e., John's threads of makeWithdrawal() happens before, Anita's getBalance().

Hence Anita's thread sees the latest value.

(20)

Hence this happens-before relationship is achieved via synchronization and this notion of happens - before is part of JMM. and here JVM implement JMM when it sees the synchronized keyword.



when JVM encounters keyword synchronizat°, it gets to know that, there is a need of thread - co-ordination and for this it inserts special instructions called "memory barriers" - i.e., co-ordinat° b/w cache and shared memory of threads.

happens - before ordering

with the happens - before ordering we can be sure that, the results of a write action by 1 thread are guaranteed to be visible to a read action by another thread only if the write operation happens - before the read operation.

Note that JMM specifies few scenarios or rules when "happens-before ordering" happens where synchronization is one among them.

1) Program order

This rule says that, each action in a given thread happens-before every other action that comes later in the thread.

2) Lock - got to be same lock!

Releasing a lock happens-before every subsequent acquisition of that same lock.

3) Volatile variable

Volatile is basically a keyword that can be used while declaring the variable, and write to these kind of variable happens-before every subsequent read of that variable regardless of the thread doing read operation.

4) start()

What it means is, a call to a thread start() on a given thread happens-before every other action that happens within that thread.

5) join()

What it means is, all actions in a particular thread happens-before, the thread that successfully returns from a join on that thread.

6) interrupt() - when forceful termination/stop the process of thread

The act of calling interrupt() by 1 thread on the another thread happens-before, the interrupted thread detects the interrupt itself.

7) Object.finalize()

finalize() is usually invoked by GC when it detects that there are no more references to the objects. Sub-classes of Object can override finalize() to perform any clean-up operations before the object is garbage collected. and with regards to happens-before ordering, the rule is that, the end of the constructor of an object happens-before the start of the finalizer of that object.

8) Transitivity.

Say, if action A happens-before action B and if action B happens-before action C, then action A happens-before action C.

Volatile variables

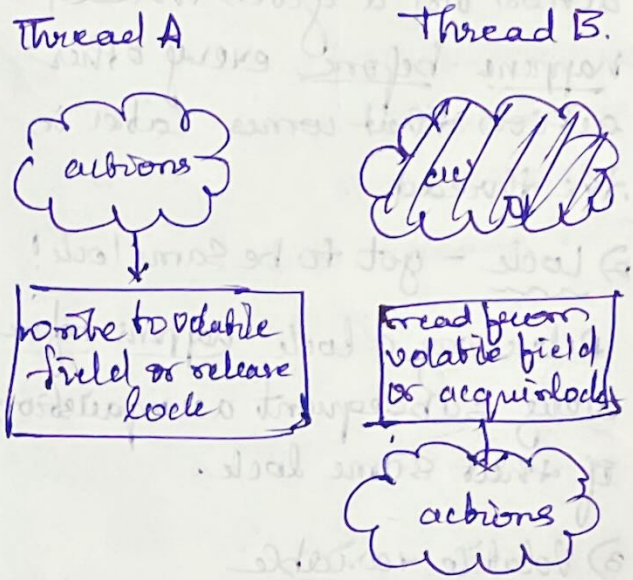
* Volatile is simply the modifier that is used with variables which helps in ensuring the memory visibility.
* Technically, once a particular thread writes a value to a volatile variable, it is guaranteed that, that particular value would be visible to all subsequent reads of that variable from all threads
* Always the data is stored in main-memory - not in registers (or) cache memory.
* Volatile keywords can be used with shared mutable variables.
* Note - volatile keywords cannot be associated with the local variables as the local variables are constrained within the functional block.

[Basically, locking can guarantee both visibility and mutual exclusion while volatile variables can only guarantee visibility].

Volatile - One thread writes and others read → no race condition.

Synchronized - used in case of race condition, eg. check-then-act.

All Actions preceding a write to a volatile field or release of a lock are visible to all actions that follow a "subsequent" read of same volatile variable (or) acquisition of same lock.



Consider having 2 threads A & B performing set of actions and the visibility of memory to read and write of a volatile variable. (T ⇒ time duration)

Thread A — Thread B.

T1 : x = 10, v = 15

T2 : Launched

T3 : x = 11

T4 : reads x.
sees x = 10

T5 : v = 16

T6 : Reads v then x
sees v = 16,
& x = 11

Hence, with volatile keyword thread B gets to see most recent

updated value of both v and x.

Hence, we got to know that volatile keyword guarantees memory visibility.

Automic variables

Automic variables are basically are reffered as better volatile variables, as they offer same memory visibility guarantees as volatile variables, but at the same time, it also supports automicity locks but they are constrained, inlike synchronized - i.e., they cannot block a set of code, instead, it is specific to a single variable.

say for example, as we knew with id generation code snippet i.e.,

```
public class IDGenerator{
  private long id;
  Long getNewId(){
    return id++;
  }
}
```

→ readid
→ increment - id
→ writeid.

as you can see that 3 processes are happening at 1 shot with return id++. So, in this case there is huge possibility of occuring Race condition.

considering this, we can actually use synchronized method i.e.,

```
Public class IDGenerator {
  private long id;
  synchronized long getNewId(){
    return id++;
  }
}
```

Here making it synchronized for just a small operation is not worth it, this because, it makes other threads to wait by getting into the BLOCKED state just to get the lock. After getting into Blocked state, it has to come to RUNNABLE and then to RUNNING State.

Hence it becomes an expensive process.

Hence we wanted something that is more compatible, so here we can use atomic variable in here.

The term Atomic in a atomic variable tells us that the variable has something to do with atomicity.

Atomic variables are represented by atomic variable classes which are from the package, java.util.concurrent.atomic.

Here the classes support atomic operations on a single variable.

so, the code looks like.

(27)

```
public class IDgenerator{
    private AtomicLong id =
    new AtomicLong();
    public long getNewId(){
    return id.incrementAndGet();
}
}.
```

Here in incrementAndGet() 3 sub-process will happen and return the value. Hence when multiple thread access, it ensures that the id is unique.

Through this, we can achieve,
* Memory visibility.
* Atomicity benefit of synchronization.

Here AtomicLong has this memory visibility because it internally uses volatile variable.

It is very much restricted to 1 variable holds the disadv over synchronization where the whole block of code can be protected.

Since there are no locks, hence no thread suspension & resumption and they do this by using an approach called compare and swap.

(28)

```
public final long incrementAndGet()
{
    for(;;){
        long current = get();
        long next = current + 1;
        if(compareAndSet(current, next))
            return next;
    }
}.
```

Here we have an infinite loop and in here, we are first fetching the current value of the atomic variable and we are storing it in a current variable.

After that we are invoking 1 more method i.e, compare And set method in Atomic Long and this methods invokes a native method that works automatically implements an instruction compare&swap

compare&swap 1st compares the current value which we pass, to the current value in the atomic variable at that instance of time. If the values match, then it implies that the current value has not changed by any other thread since it was read by

the current thread and so the current atomic value will be swapped with the value in the next variable.

So, compareAndSet invocation returns "true" and then the next value is returned by the method.

But, if the current value passed here is different from the current value in the atomic variable, that would imply that, the current value in atomic variable was updated by some other thread since the time it was read by this particular thread and due to that swap will not happen and compareAndSet would return a false. With that, the thread would loop again to see if the value can be updated in the next iteration.

This way threads keep trying to increment the value.

Hence this algorithm is called as non-blocking algorithm unlike synchronized algorithm and the technique used here is also refered to as optimistic locking - [Here the value of next variable is completed 1st, based on the optimistic assumption that, no other thread would have already set the same value. When the optimistic

assumption is true, the shared memory is updated with next value without locking. If the assumption is false, the work is lost but still there is no-locking involved.

The traditional locking with synchronized is sometimes refered to as - pessimistic locking.

Atomic Variable classes.

Basically, there are 12 Atomic variable classes. and are categorised into 4 fields i.e.,

1) Scalars
2) Arrays
3) Filed Updaters
4) Compound variables.

Scalars
- AtomicLong
- AtomicInteger
- AtomicBoolean
- AtomicReference

→ subclass of java.lang.Number rest are subclass of Object class.

Arrays
- AtomicLongArray
- AtomicIntegerArray
- AtomicReferenceArray

Field Updators
- AtomicLongFieldUpdator.
- AtomicIntegerFieldUpdator.
- AtomicReferenceFieldUpdator.

These are reflection based utility classes that wrap around corresponding volatile variables and can perform

atomic operation on them.

Compared variables

* AtomicMarkableReference
* AtomicStampedReference.

These classes are used if we want to couple together an obj refernce with either a boolean flag or with an integer.

If it is boolean flag it is AtomicMarkableReference, else it will be AtomicStampedReference

Synchronizat' Best Practices.

Tips

1) Synchronize everywhere where the variable is accessed.
2) Guard same shared mutable variable with same lock.
3) If a method modifies a static field, you MUST synchronize access to this field even if the method is typically used only by a single thread.
4) Synchronize only what is needed - keep blocks small.
5) Try to move time consuming activities out of synchronized blocks without affecting thread safety.
6) Item 67: Avoid excessive synchronizat'.

Costs

Costs ⇒ * Thread suspension/resumption.

* Lock acquisition/release
* Ensuring consistant memory visibility.
* Limits JVM's ability to optimize code execution.
* Lost opportunities for parallelism in multi-core system.

7) If you not sure of synchro-nizing then it is good not to synchronized but with that document that the class is not thread-safe.

NOTE

STRING BUFFER found that string buffer instances were mostly used by single threads i.e., they are not shared across multiple threads but here the class uses synchronizat' internally wherever it is needed and that affected the performance. and this was the reason in java 5, STRING BUILDER was introduced as a dropping replacement for string buffer. STRING BUILDER dosen't use any synchronizat' hence it is faster. Hence when synchro-nizat' is needed then we shall go for STRING BUFFER.

Till now, we were actually concerned about 2 things mainly i.e.,

Race condition - and this was overcome by Atomicity via synchronizat°, Atomic variables

Memory visibility - Synchronizat° Atomic Variables, Volatile.

Here are 2 more things (or) requirements that we have to mainly focus on as well.

Thread Co-operat° - Usually we know that with the synchronizat° we can attain upto some level of thread co-operation where the threads not interfere with each other. However, in certain scenarios we need more than this. Say, for instance, if one thread is inside the synchronizat° region may not be able to perform its task until some other thread completes its task which is relavent. And for that, the 1st thread has to let the other thread take over & perform its task befor the 1st thread can continue.

Note that, the 1st thread needs to know, whether or not the other thread has performed the task or not. and this information is (34) captured using some shared state and if the shared state meet certain criteria which is basically a condition only then the 1st thread will continue its actions.

Hence here we see real interaction b/w threads and we have 3 approaches.

* Naive (via volatile variables)
* wait() & notify()
* Java's concurrency utilities.

Thread Management

It is important, when we are dealing with large-scale applications where we will have lots & lots threads running concurrently. for this we should have good thread creat° and managing facilities. Here comes the role of EXECUTOR FRAMEWORK.

Example Usecase

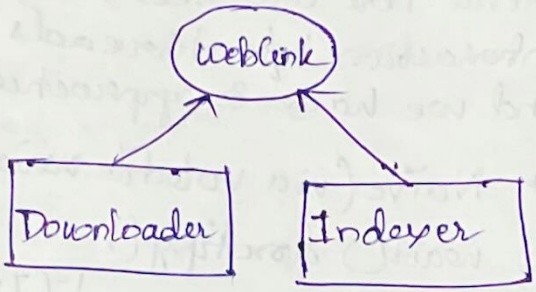
Web page Indexing.

It is a process done by search engines like Google & its a complex process.

1) The first step is downloading web pages and this is done by web crawler

2) Once all the recursive web pages are downloaded they

have to be indexed, for this search engines, they have built something called inverted index which is a datastructure that allows us to search for documents in a efficient way.



* Here in our demo project we will 3 classes as shown above. i.e., Downloader, Indexer and Weblink.
* An instance of weblink represents a URL of a webpage and this weblink object is shared b/w Downloader and Indexer objs basically the 2 threads.
* So, for each weblink we are basically creating downloader & indexer threads and they share the same weblink object.
* Downloader thread is responsible for downloading the webpage referenced by weblink and Indexer thread is responsible for indexing the downloaded webpage.
* So, there will be thread co-operation here, as the indexer should be able to index only after the downloader thread downloads the web page.

* Say, even if the indexer page is ready to index, and the downloader has not yet downloaded the webpage, then indexer needs to wait for the downloader to download the web page.
* Condition - To index the webpage, the downloading of the web page has to be completed.

Naive Approach

* As we were discussing about the web crawler, hence here looking in the perspective of Naive approach * Here, the only problem is that the Indexer has to wait until the webpage gets downloaded
* Hence the limitation is CPU cycles are wasted ~~as~~ in Indexer as it is actually waiting for the page to be downloaded.
* Note :- Say, if there are N weblinks, this approach creates 2*N threads i.e., 1 for weblink downloads and 2nd is for Indexer -
* Here - htmlPage is made volatile.

Now, we try to eliminate the limitation of Naive approach by using some other approach.

Wait And Notify Approach

Here in this approach wherever we use wait & notify method make sure that they are made atomic unit block i.e. they have to be synchronised.

Reason :- say for example, let us assume that, the indexer thread has started running 1st and say it has stopped by thread scheduler befor wait() is executed. Say now the downloader thread is given a chance to perform and say the weblink is downloaded and has invoked notify method.

Now, say the indexer thread started processing and will invoke the wait() and will be in a wait state forever even if the webpage has already downloaded.

Hence, it has to be made as an atomic unit.

If it is synchronised, then once after it invokes wait() it will also release the lock, but we have to make sure that even notify() is synchronised and uses the same lock. (refer code).

Hence, by this method, we not actually wasting CPU cycle unstead we are suspending the thread function - activity.

Here the wait() should always be used with while condition. It is the std idium

Reason - There is something called spurious wakeup, case where the thread which wakes up and moves from wait state to blocked state without notify() invoked.

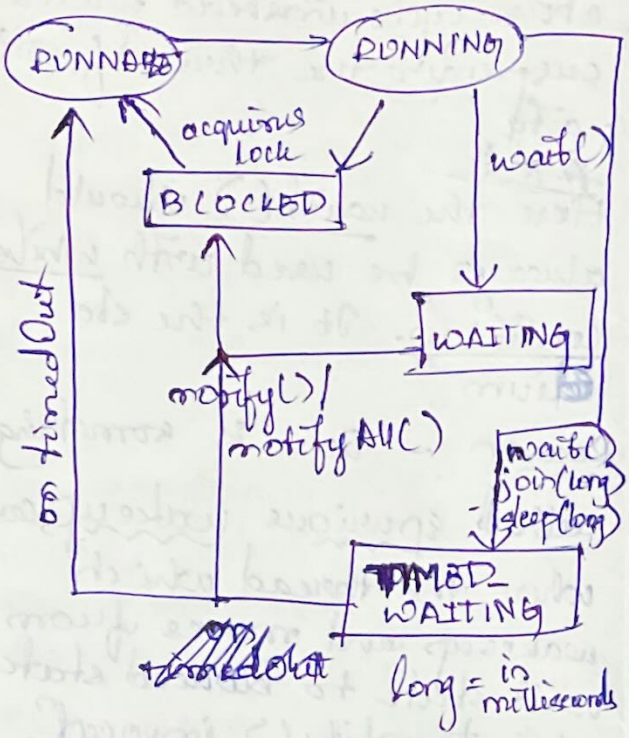
Benefits of using wait() & notify() & Limitations

Bene - Better co-ordinal of control flow b/w threads. Relinquishes lock on wait()

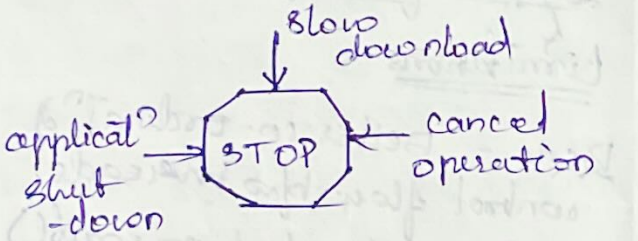
limitat? - Solves task co-operat in low-level fashion because synchronized blocks are needed.

Note :- htmlPage is NOT declared volatile in Weblink as unlock on a monitor/lock 'happens before' every subse-quent lock on that same monitor.

Thread schedules



Thread Interruption



There can be the reasons for the thread interruption where the situation comes for us to forcefully stop the thread.

But, Java usually doesn't allow forcibly stopping a thread.

The Thread class has a method stop() to forcibly stop the thread but it has been depricated.

stop() } or depricated
suspend() }
resume() }

i.e., Java, instead of stoping thread forcefully, it provides co-operative mechanism for requesting a thread to stop what it is doing. (40)

do any clean-up / stop gracefully

Why co-operative and not forcefull stop ?

↓

can leave shared objects in inconsistent state.

If we take example of stopThread where when the condition stop = true then the thread is stopped. Say for instance, if we have invoked some other methods like wait() or timed_wait() and other related methods like join(), sleep() that blocks or takes thread into BLOCKED state, post which we never know when they gonna come back to RUNNABLE state, at this time, the interruption can be used.

Interruption

* Every thread has a boolean interrupted status and i.e., set to "false" by default.

* interrupt() :-
sets interrupted status to true.

* This means that the thread is been insisted to stop at its own convenience gracefully. Technically, it is upto the thread how it responds to the request.

Blocking methods (ex, sleep(), wait())

* Blocking methods like sleep() or wait() detect when a thread was interrupted and they try to return early essentially they respond by clearing the interrupted status. i.e., resetting the interrupted status to false and also throwing an InterruptedException.

* WKT, the blocking methods are enclosed by try-catch block where the exception handling code has an interrupted except°. So, if the interrupted status has been set to true and target method executing a blocking method, then it would throw an interrupted except° after reseting the status flag to false.

* Ideally even, if a thread is already in a state like waiting (or) time waiting (or) blocked, it will respond by reseting the interrupt° status & by throwing Int Exp°.

indicating that it has been interrupted and the catch block handling except° can include any cleaning code

* Say, if the program of ours does not take into any blocking methods, we can use isInterrupted()

```
public void run(){
  while(!Thread.currentThread().
        isInterrupted()){
    ...
  }                    → To terminate.
}                      // from Thread class.
```

* Other method,

```
public void run(){
  while(!Thread.currentThread().
        interrupted()){
    ...
  }
}
```

↓

Clears interrupted status & returns previous value of the status flag. If the method returns true, the loop would terminate.

[This is the only method that can be used for reseting the interrupted status flag.]

[Note - If the thread is blocked due to stream IO - then that thread is uninterruptible].

i.e., when the thread is waiting for some data from the source - [to interrupt - close the stream

[Ugly when a thread is blocked due to intrinsic lock like Synchronized locks - Even they are uninterruptable.]
→ Use explicit locks instead.

Explicit locks

Threads waiting on explicit locks are interruptable.

Common Template

```
import java.util.concurrent.locks.*;
...
Lock lock = new ReentrantLock();
...
lock.lock();
try{
  // critical section
}finally{
   lock.unlock();
}
```

Here we have lock() through which lock is acquired and we then we have some critical section guarded by the block. At last, we release the lock in the finally block.