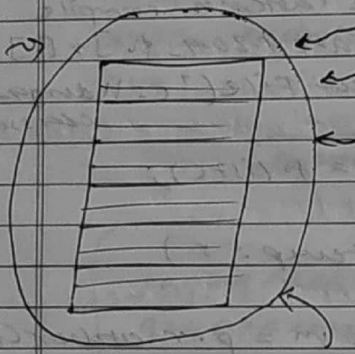
Concurrent Collections

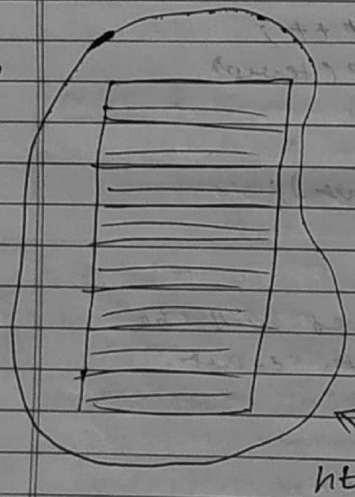
ConcurrentHashMap (C)

→ If we compare ConcurrentHashMap with HashMap & Hashtable,



HashMap Object

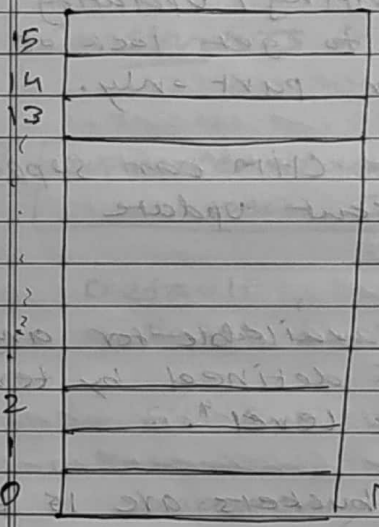
- HashMap is not thread-safe, so simultaneously any no. of thread can operate on same object with may lead to Data Inconsistency Problem.



Hashtable Object

- Hashtable is Thread-safe, but performance wise not up to the mark because even for read operation Whole Hashtable object's lock is required.

[Both uses the Hashtable data structure & bucketing only]



- ConcurrentHashMap also uses "hashtable" as underlying data structure.
- CHM also uses bucketing concept.

[By default 16 buckets are created]

- CHM allows any number of concurrent Read operations, means for read operation no lock is required.
- CHM uses different Locking Mechanism called "**Segment Locking**". which means thread will have to lock only part of CHM.
- CHM also allows concurrent ~~operation~~ write/update operations in safe manner.

- For ex. for writing / updating thread needs to get lock of that particular part only.
- so, by default CHM can support upto 16 concurrent update operations.
- No. of locks available for any CHM object is defined by term "Concurrency level".
- so if no. of buckets are 16 & concurrency level is also 16 then 1 lock per every bucket is available

  & if No. of buckets are 16 & concurrency level is 32 then 2 locks per every bucket is available that means that CHM object can support upto 32 concurrent update operations.
- In CHM, null is not allowed for both keys & values just like "Hash table".
- CHM never throw with ConcurrentModificationException

⇒ Constructors :-

(1) `ConcurrentHashMap chm = new ConcurrentHashMap();`

Default,
No. of buckets (Initial capacity) = 16

Fill Ratio = 0.75

Concurrency level = 16

(2) `CHM chm = new CHM (int initialCapacity);`

(3) `CHM chm = new CHM (int initialCapacity, float fillRatio);`

(4) `CHM chm = new CHM (int initialCapacity, float fillRatio, int concurrencyLevel);`

(5) `CHM chm = new CHM (Map m);`

→ Ex1.

```
import java.util.concurrent.
        ConcurrentHashMap;

class CHMDemo1
{
    p s v main (Str[] )
    {
        ConcurrentHashMap chm =
            new ConcurrentHashMap();

        chm.put(101, "A");
        chm.put(102, "B");
        chm.putIfAbsent(103, "C");
        chm.putIfAbsent(101, "D");

        chm.remove(101, "D");
        chm.replace(102, "B", "E");

        S.O.p(chm);

    }
}
```

O/P :- {103=C, 102=E, 101=A}

[We can't give the guarantee for order of ~~o/p~~ o/p]

→ Ex2.

```
import java.util.concurrent.
        ConcurrentHashMap;

class MyThread extends Thread
{
    static ConcurrentHashMap chm =
            new CHM();

    public void run()
    {
        try
        {
            Thread.sleep(2000);
        }
        catch(InterruptedException e)
        {}
            S.O.p("Child Thread
                Updating Map");
        m.put(103, "C");
    }
    p s v main(Str[] ) throws
        InterruptedException
    {
        m.put(101, "A");
        m.put(102, "B");

        MyThread t = new
            MyThread();
```

```
t.start();

Set s = m.keySet();
Iterator itr = s.iterator();

while(itr.hasNext()).
{
    Integer I1 = (Integer)
                 itr.next();

    s.o.p("Main Thread
          iterating &
          current Entry is :"+
          I1 +"----"+ m.get(I1));
  }
  s.o.p(m);
 }
}
```

O/P :-

[If we replace CHM with HM then we will get ConcurrentModificationException.]

Main Thread Iterating & current Entry is : 102.....B

Child Thread updating Map

Main Thread iterating & current Entry is : 101.....A

{103=C, 102=B, 101=A}

Note:- While getting O/P, Iterator might not get the updated value by another thread bcz Iterator might already visited the bucket which second Thread updated & Iterator iterates in forward direction only.

	HashMap	ConcurrentHashMap
(1)	It is not Thread safe	It is Thread-safe
(2)	Relatively performance is high because Threads are not required to wait to operate on HashMap	Relatively Performance is Low bcz some times Threads are required to wait to operate on ConcurrentHMap.
(3)	While one Thread iterating HashMap the other Threads are not allowed to modify Map objects otherwise we will get Runtime Exception saying ConcurrentModificationException.	While one Thread iterating CHM the other Threads are allowed to modify Map Objects in safe Manner & it won't throw ConcurrentModificationException.

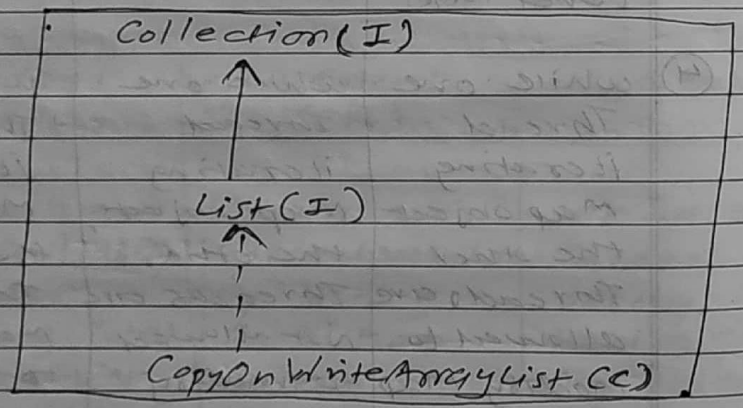
④	Iterator of HM is Fail-Fast bcz it throws CME	Iterator of CHM is Fail-Safe bcz it won't throws CME.
⑤	null is allowed for Both keys & values.	null is not allowed for Both keys & values, otherwise we will get NPE.
⑥	Introduced in 1.2 version	Introduced in 1.5 version.

→

	Concurrent HashMap	synchronized Map()	Hashtable
①	We will get Thread safety without locking Total Map object Just with Bucket level lock / segment loking.	We will get Thread safety by locking Total Map object	We will get Thread safety by locking Total Map object.
②	At a time multiple Threads are allowed to operate on Map objects in safe Manner.	At a time only one Thread is allowed to perform any operation on Map object.	At a time only one Thread is allowed to operate on Map object.
③	Read operation can be performed without lock but write operation can be performed with Bucket level lock	Every Read & write Operation require total Map object	Every Read & write operation require total Map object.
④	while one Thread iterating Map object the other Threads are allowed to modify Map & we don't get CME.	while one Thread iterating Map object, the other Threads are Not allowed to modify Map. otherwise we will get CME	while one Thread iterating Map object the other Threads are Not allowed to modify map otherwise we will get CME
⑤	Iterator of CHM is Fail-safe	Iterator of synchronized Map() is Fail-Fast	Iterator of Hashtable is also Fail-Fast.

6	null is not allowed for Both keys & values	null is allowed for Both keys & values	null is not allowed for Both keys & values
7	Introduced in 1.5 version	Introduced in 1.2 version	Introduced in 1.0 version

✱ CopyOnWriteArrayList(C):



- It is Thread-safe version of ArrayList.
- As the name suggest CopyOnWrite ArrayList creates a separate cloned copy of underlying ArrayList for Every Update Operation.
- At certain point Both (The original AL & cloned copy) will be synchronized Automatically which is taken care by JVM internally.
- As Update operation will be performed on cloned copy there is no effect for the Threads which performs Read operations.
- It is costly to use because for every update Operation a cloned copy will be created. Hence CopyOnWriteArrayList is the best choice if several Read operations & less number of write operations are required to perform.
- Insertion order is preserved.
- Duplicate objects are allowed.
- Heterogeneous objects are allowed.

→ null Insertion is possible.

→ It implements serializable, Cloneable & RandomAccess Interface.

→ While One Thread iterating CopyOnWriteArrayList, the other Threads are allowed to modify & won't get concurrentModification Exception. That is iterator is Fail-safe.

→ Iterator of ArrayList can perform Remove operation but Iterator of CopyOnWriteArrayList can't perform Remove operation. Otherwise we will get R.E. saying UnsupportedOperationException.

⇒ Constructors :-

① `CopyOnWriteArrayList cowAL = new CopyOnWriteArrayList();`

② `CopyOnWriteArrayList cowAL = new CopyOnWriteArrayList(Collection c);`

③ `CopyOnWriteArrayList cowAL = new CopyOnWriteArrayList(Object[] a)`

⇒ Methods :- (Additional)

① `boolean addIfAbsent(Object o)`

→ The element will be Added if and only if list doesn't contain this element.

→ Ex:

```
COWAL l = new COWAL();

l.add("A");
l.add("A");

l.addIfAbsent("B");
l.addIfAbsent("B");

S.O.P(l);
```

O/P :- [A, A, B]

Note :- These methods of cowAL are in addition to the methods inherited by it from parent Interfaces List(I) & Collection(I)

(2) int addAllAbsent(Collection c)

→ The elements of collection will be added to the list if elements are absent & Returns number of elements Added.

→ Ex.

```
ArrayList l = new ArrayList();
l.add("A");
l.add("B");

COWAL cl = new COWAL();
cl.add("A");
cl.add("C");
s.o.p(cl);          // [A,C]
cl.addAll(l);
s.o.p(cl);          // [A,C,A,B]

ArrayList l2 = new ArrayList();
l2.add("A");
l2.add("D");

cl.addAllAbsent(l2);
s.o.p(cl); // [A,C,A,B,D]
```

⇒ Ex 1.

```
import java.util.concurrent.*;
import java.util.*;

class MyThread extends Thread
{
  static COWAL cl = new COWAL();

  public void run()
  {
    try
    {
      Thread.sleep(2000);
    }
    catch(InterruptedException e)
    {
      s.o.p("catch Block");
    }
    .
    s.o.p("child Thread
          ~~Iterating~~ Updating
          List");
    l.add("C");
  }

  p s v main(sn) throws
                       IE
  {
    cl.add("A");
    cl.add("B");
```

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

Iterator itr = cl.iterator();

while ( itr.hasNext())
{
    String s1 = (String) itr.next();

    S.O.P("Main Thread
          Iterating List &
          current object is :" +
                    s1);
    Thread.sleep(3000);
}

S.O.P(cl);
}
}
```

O/P :-

Main Thread Iterating List &
current object is : A

Child Thread updating list

Main Thread Iterating List &
current object is : B

[A, B, C]

[Here, we won't get CME bcz of COWAL]

⇒ Ex 2 :-

```
import java.util.concurrent.*;
import java.util.Iterator;

class Test
{
    p s v main (Str)
    {
        COWAL cl = new COWAL();
        cl.add("A");
        cl.add("B");
        cl.add("C");
        cl.add("D");

        S.O.P(cl); // [A, B, C, D]

        Iterator itr = cl.iterator();
        while ( itr.hasNext())
        {
            String s = (String) itr.next();
            if (s.equals("D"))
            {
                itr.remove();
            }
        }
        S.O.P(cl); // R.E. java.lang.
    }                  Unsupported
}                      Operation
                       Exception.
```

[If we replace COWAL with AL we won't get any R.E. & O/P = [A, B, C]]

EX-3 :-

```
import java.util.concurrent.*;
import java.util.Iterator;

class Test
{
    p s v main(Str[])
    {
        CowAL cl = new CowAL();

        cl.add("A");
        cl.add("B");
        cl.add("C");          ——— (1)

        Iterator itr = cl.iterator();
                              ——— (2)
        cl.add("D");
                              ——— (3)

        while(itr.hasNext())
        {
            String s = (String)
                       itr.next();

            S.o.p(s);
        }
    }
}
```

O/P :- [A, B, C]

→ After (1)

[A|B|C]

After (2)

itr → [A|B|C]

After (3)

itr → [A|B|C] [A|B|C|D]

→ So, After getting Iterator on old list if we try to update it then separate cloned copy gets generated which won't get Iterator. So O/P will be [A, B, C]

→ If we Replace CowAL with AL then R.E. CME, bcz in AL we are not allowed to update AL after getting iterator on it.

→	ArrayList	CopyOnWrite ArrayList
(1)	It is not Thread-safe	It is not Thread safe because every update operation will be performed on separate cloned copy.

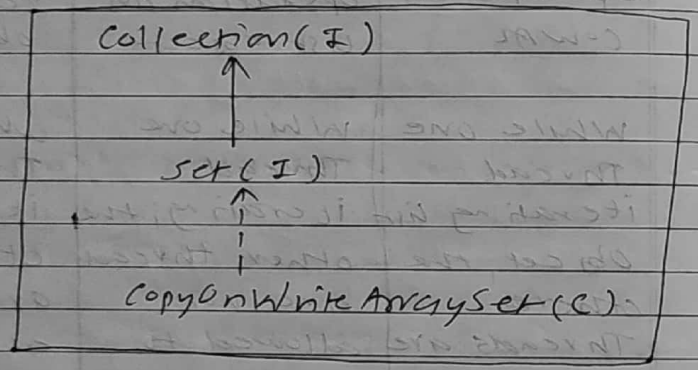
(2)	while one Thread iterating list Object, the other Thread are not allowed to modify list otherwise we will get CME.	while one Thread iterating list Object the other threads are allowed to modify list in safe manner & we won't get CME.
(3)	Iterator is Fail-Fast	Iterator is Fail-Safe.
(4)	Iterator of AL can perform remove Operation	Iterator of CowAL can't perform Remove operation Otherwise we will get RE: UOE
(5)	Introduced in 1.2 version	Introduced in 1.5 version

→

CopyOnWriteArrayList	synchronizedList()	Vector
(1) we will get Thread Safety because Every update Operation will be performed on separate cloned copy	we will get Thread Safety bcz at a time only one Thread is allowed to access the list	we will get Thread Safety bcz at a time only one Thread is allowed to access Vector.
(2) At a time multiple Threads are allowed to access/ operate on CowAL	At a time only one Thread is allowed to perform any operation on list object	At a time only one Thread is allowed to operate on vector object.
(3) While one Thread iterating list Object, the other Threads are allowed to modify & we won't get any CME	While one Thread iterating, the other threads are not allowed to modify list otherwise we will get CME.	while one Thread iterating, the other threads are not allowed to modify vector object otherwise we will get CME
(4) Iterator is Fail-Safe & won't raise CME.	Iterator is Fail-Fast & will raise CME.	Iterator is Fail-Fast & will raise CME.
(5) Iterator can't perform remove operation else we will get	Iterator can perform remove operation	Iterator can perform remove operation.

⑥	Introduced in 1.6 v	Introduced in 1.2 v	Introduced in 1.0 v

✱ CopyOnWriteArraySet(C)



⇝ It is Thread-safe version of set.

⇝ Internally implemented by CopyOnWriteArrayList

⇝ Insertion order is preserved.

⇝ Duplicate objects are not allowed.

⇝ Multiple Threads can able to perform Read operation simultaneously but for every update operation a separate cloned copy will be created.

⇝ As for every update operation a separate cloned copy will be created which is costly, hence if multiple update operation are required then it is not recommended to use CopyOnWriteArraySet.

⇝ While one Thread iterating set the other Threads are allowed to modify set & we won't get CME.

⇝ Iterator of CopyOnWriteArraySet can perform only Read operation & won't perform UOE.

⇝ Constructors :-

① `CopyOnWriteArraySet cs = new CopyOnWriteArraySet();`

② `CopyOnWriteArraySet cs = new CopyOnWriteArraySet(Collection c);`

Methods :-

Whatever methods present in collection & set interface are the only methods applicable for CopyOnWriteArrayset & there are no special methods bcz in set duplicates are not allowed already.

Ex.

```
import java.util.concurrent.*;

class Test
{
    p s v main(Sth)
    {
        COWAS cs = new COWAS();

        cs.add("A");
        cs.add("B");
        cs.add("C");
        cs.add("A");
        cs.add(null);
        cs.add(10);
        cs.add("D");
    }
}
```

O/P :- [A, B, A, null, 10, B]

	CopyOnWriteArrayset	synchronized set()
(1)	It is Thread-safe bcz every update operation will be performed on separate cloned copy	It is Thread safe because at a time only one thread can perform operation.
(2)	While one Thread iterating set, the other Threads are allowed to modify & we won't get CME	While one Thread the other Threads are not allowed to modify set otherwise we will get CME.
(3)	Iterator is Fail-safe.	Iterator is Fail-Fast.
(4)	Iterator can perform only Read operation & can't perform Remove operation else we will get R.E. UOE	Iterator can perform Both Read & Remove operation.
(5)	Introduced in 1.5 V	Introduced in 1.2 V.