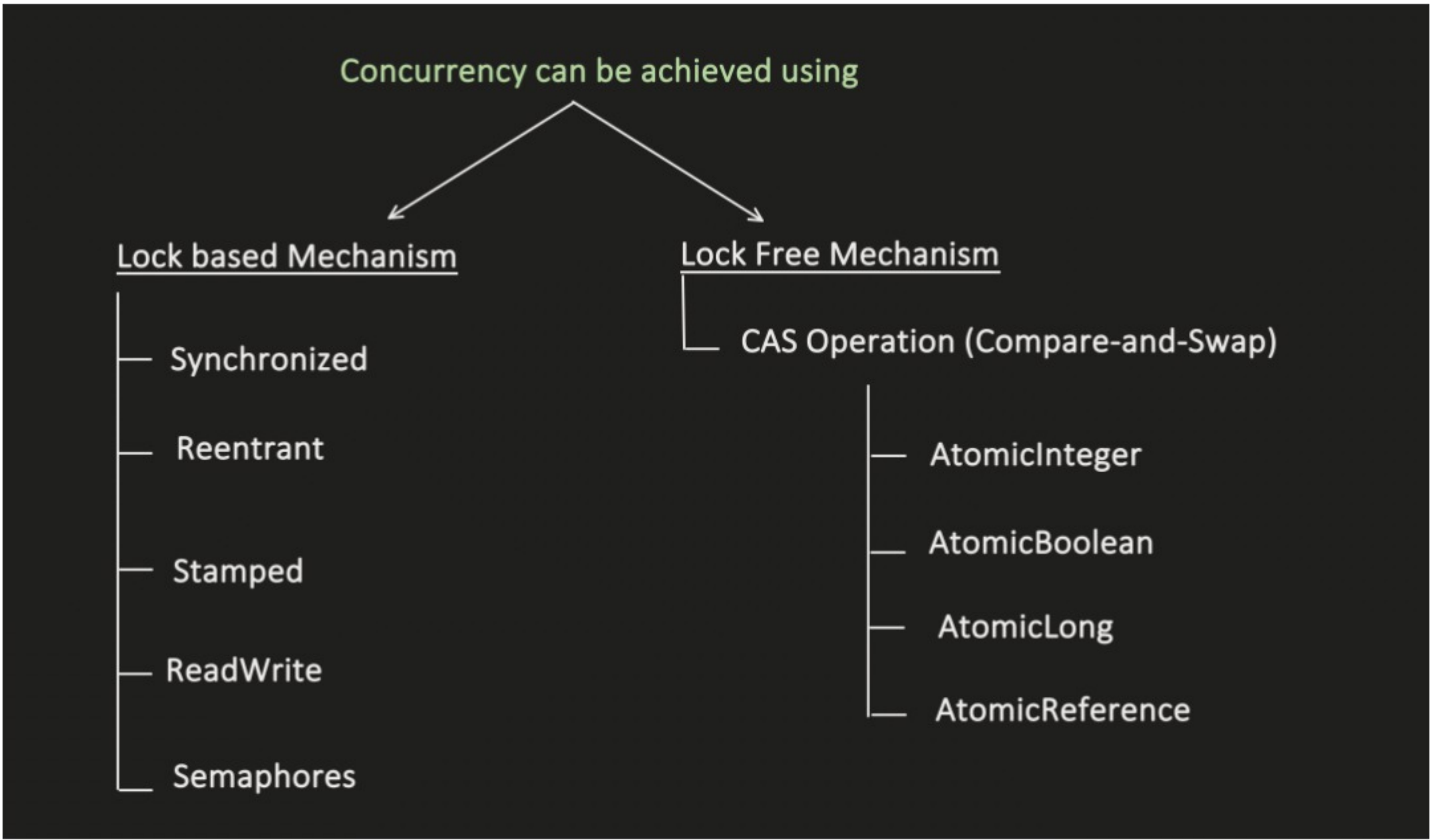


Lock Free Concurrency (CAS) : Multithreading Part5

"Concept && Coding" YT Video Notes



Report Abuse

Lock Free Mechanism

It uses CAS (compare and Swap) technique:

- It's a Low level operation.
- Its Atomic.
- And all modern Processor supports it.

It involves 3 main parameters:

- Memory location: location where variable is stored.
- Expected Value: value which should be present at the memory.
 - ABA problem is solved using version or timestamp.
- New Value: value to be written to memory, if the current value matches the expected value.

Atomic Variables:

What ATOMIC means:

- It means Single or "all or nothing"

```
public class Main {  
  
    public static void main(String[] args) {  
  
        SharedResource resource = new SharedResource();  
        for(int i=0; i<400; i++) {  
            resource.increment();  
        }  
        System.out.println(resource.get());  
    }  
}
```

```
public class SharedResource {  
  
    2 usages  
    int counter;  
  
    no usages  
    public void increment() {  
        counter++;  
    }  
  
    no usages  
    public int get() {  
        return counter;  
    }  
}
```

Output:

400

Process finished with exit code 0

```
public class Main {  
  
    public static void main(String[] args) {  
  
        SharedResource resource = new SharedResource();  
  
        Thread t1 = new Thread() -> {  
  
            for(int i=0; i< 200; i++) {  
                resource.increment();  
            }  
        };  
  
        Thread t2 = new Thread() -> {  
  
            for(int i=0; i< 200; i++) {  
                resource.increment();  
            }  
        };  
  
        t1.start();  
        t2.start();  
  
        try {  
            t1.join();  
            t2.join();  
        } catch (Exception e) {  
        }  
  
        System.out.println(resource.get());  
    }  
}
```

```
public class SharedResource {  
  
    2 usages  
    int counter;  
  
    no usages  
    public void increment() {  
        counter++;  
    }  
  
    no usages  
    public int get() {  
        return counter;  
    }  
}
```

Output:

371

Process finished with exit code 0

2 solutions:

1. Using lock like synchronized
2. Using lock free operation like AtomicInteger

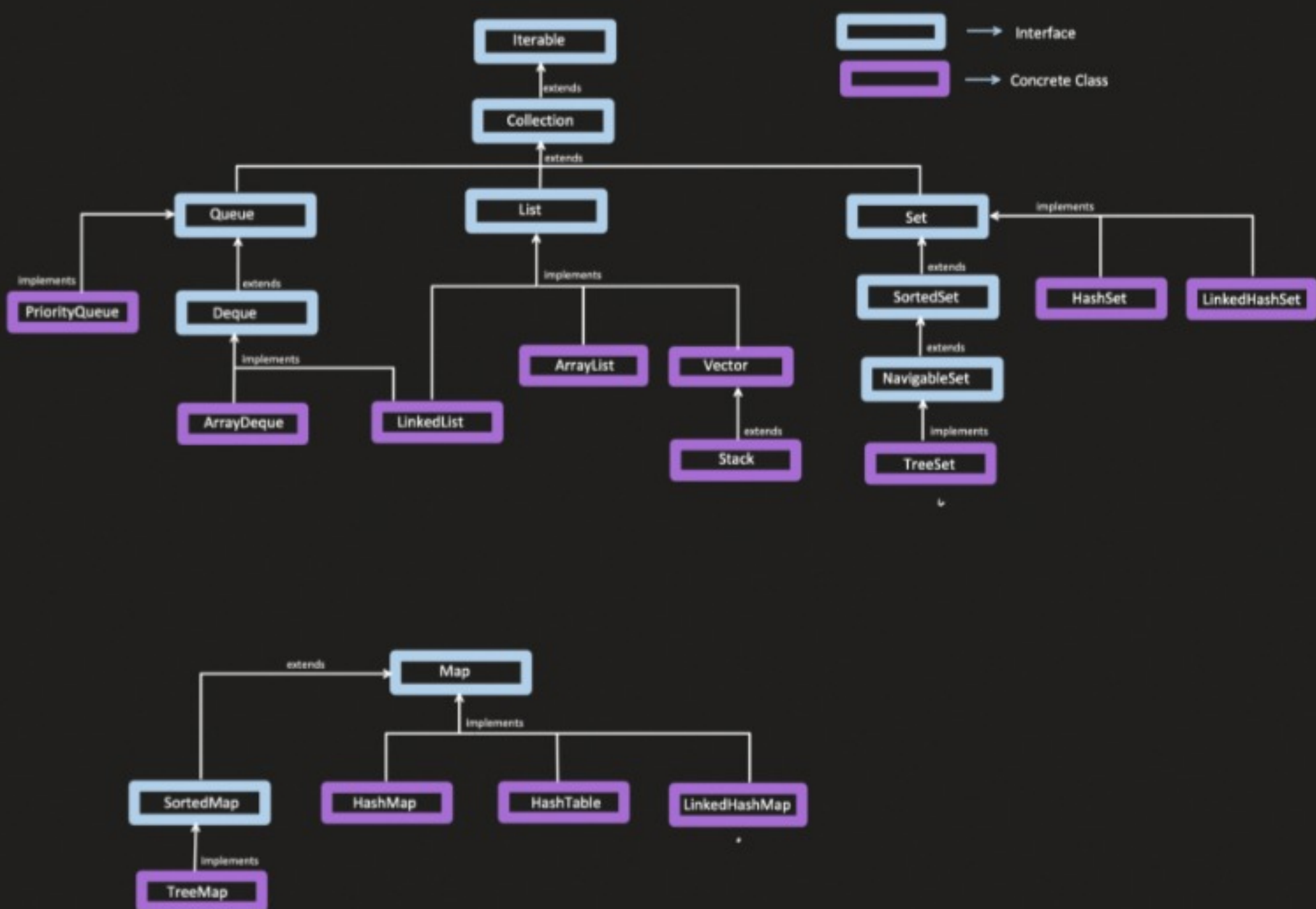
1. Using lock like synchronized

```
public class SharedResource {  
  
    2 usages  
    int counter;  
  
    2 usages  
    public synchronized void increment() {  
        counter++;  
    }  
  
    1 usage  
    public int get() {  
        return counter;  
    }  
}
```

2. Using lock free operation like AtomicInteger

```
public class SharedResource {  
  
    2 usages  
    AtomicInteger counter = new AtomicInteger( initialValue: 0);  
  
    2 usages  
    public void increment() {  
        counter.incrementAndGet();  
    }  
  
    1 usage  
    public int get() {  
        return counter.get();  
    }  
}
```

Concurrent Collection



Collection	Concurrent Collection	Lock
PriorityQueue	PriorityBlockingQueue	ReentrantLock
LinkedList	ConcurrentLinkedDeque	Compare-and-swap operation
ArrayDeque	ConcurrentLinkedDeque	Compare-and-swap operation
ArrayList	CopyOnWriteArrayList	ReentrantLock
HashSet	newKeySet method inside ConcurrentHashMap	Synchronized
TreeSet	Collections.synchronizedSortedSet	Synchronized
LinkedHashSet	Collections.synchronizedSet	Synchronized
Queue Interface	ConcurrentLinkedQueue	Compare-and-swap operation