Java Concurrency

What are Threads

- Threads are an independent path of execution in a single process.

- Threads can be started by other threads to trigger another parallel execution path (fork)

- Java threads are mapped to an operating system thread

Simple Thread

```java
public class MyThread extends Thread{
    @Override
    public void run() {
        for(int i=0; i<1000; i++){
            System.out.println("Value of i "+i);
        }
    }

    public static void main(String[] args) {
        MyThread th = new MyThread();
        th.start();
        for(int i=0; i<1000; i++){
            System.out.println("Value of i ********** "+i);
        }
        System.out.println("Started thread");
    }

}
```

Another (Preferred) Path

```java
public class MyRunnable implements Runnable{
    @Override
    public void run() {
        for(int i=0; i<1000; i++){
            System.out.println("Value of i "+i);
        }
    }

    public static void main(String[] args) {
        MyRunnable r = new MyRunnable();
        Thread t = new Thread(r);
        t.start();
        for(int i=0; i<1000; i++){
            System.out.println("Value of i ********** "+i);
        }
        System.out.println("Started thread");
    }

}
```

Divide Work

- Compute sin() of all numbers from 1 to 10000 and add it all up in two separate threads.. compare the time it takes to do the same in a single thread.
- Explore the concept of Join

Daemon Threads

- Threads that are meant to run background processess that support the main process
- Main process threads (user threads) keep the jvm alive. Whereas daemon threads do not
- Try an example to demonstrate this concept

Sleeping & Interrupting

- Threads can be put to sleep with Thread.sleep(duration)
- A sleeping thread can be interrupted by another thread by calling interrupt() on the thread reference
- Interrupting can be used on a few other blocked states such as join, wait, etc

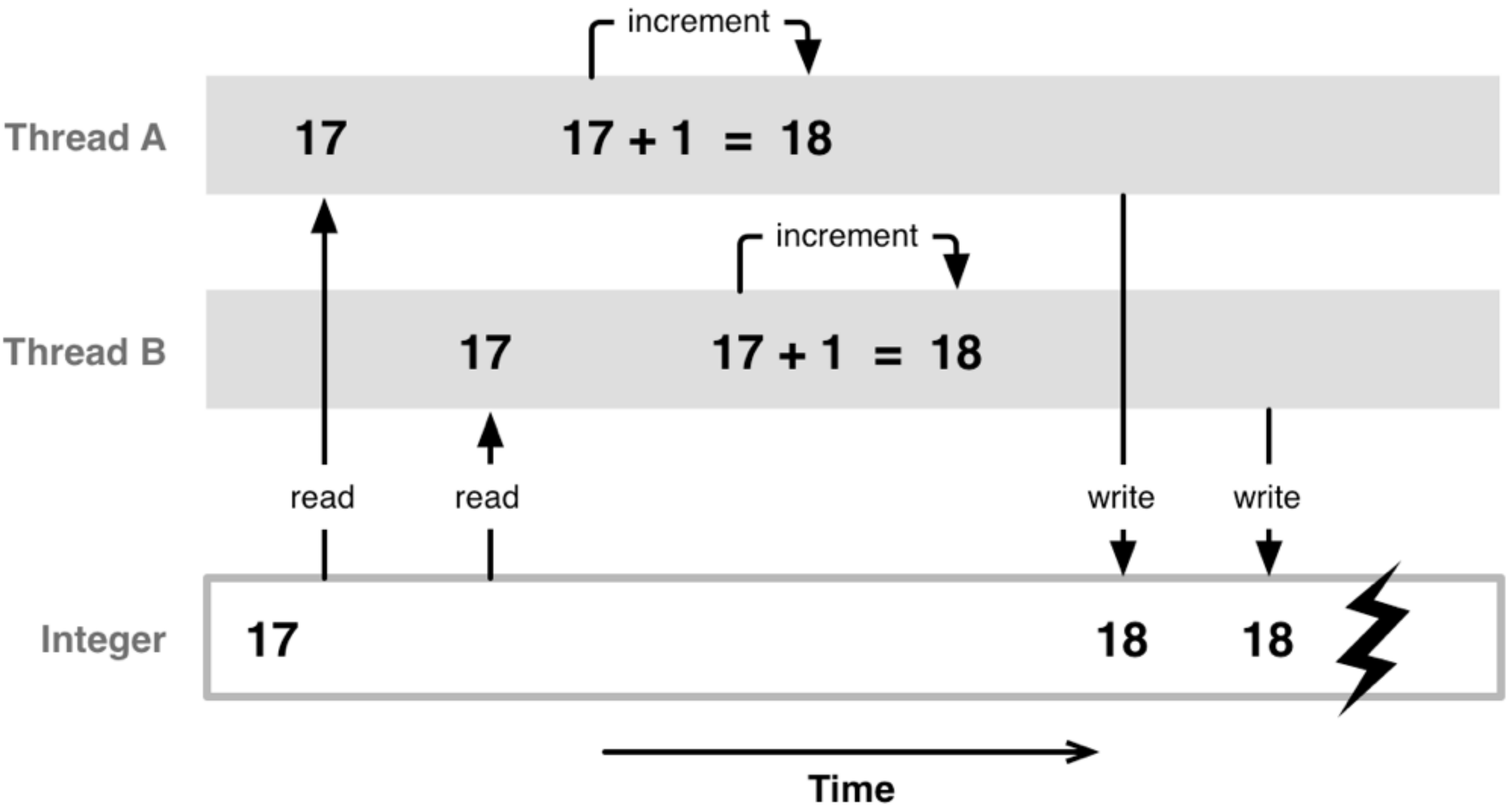
Thread Racing

```java
public class ThreadRacing extends Thread{

    public static int k=0;

    public void run() {
        for(int i=0; i< 1000; i++){
            k = i;
            System.out.println("Value of i = "+k);
            if(k!=i){
                System.out.println("This cant print!!!!!!!!!!!!");
            }
        }
    }

    public static void main(String[] args) {
        ThreadRacing t = new ThreadRacing();
        t.start();
        for(int j=0; j< 1000; j++){
            k = j;
            System.out.println("######### Value of j = "+j);
            if(k!=j){
                System.out.println("This cant print!!!!!!!!!!!!");
            }
        }
    }
}
```

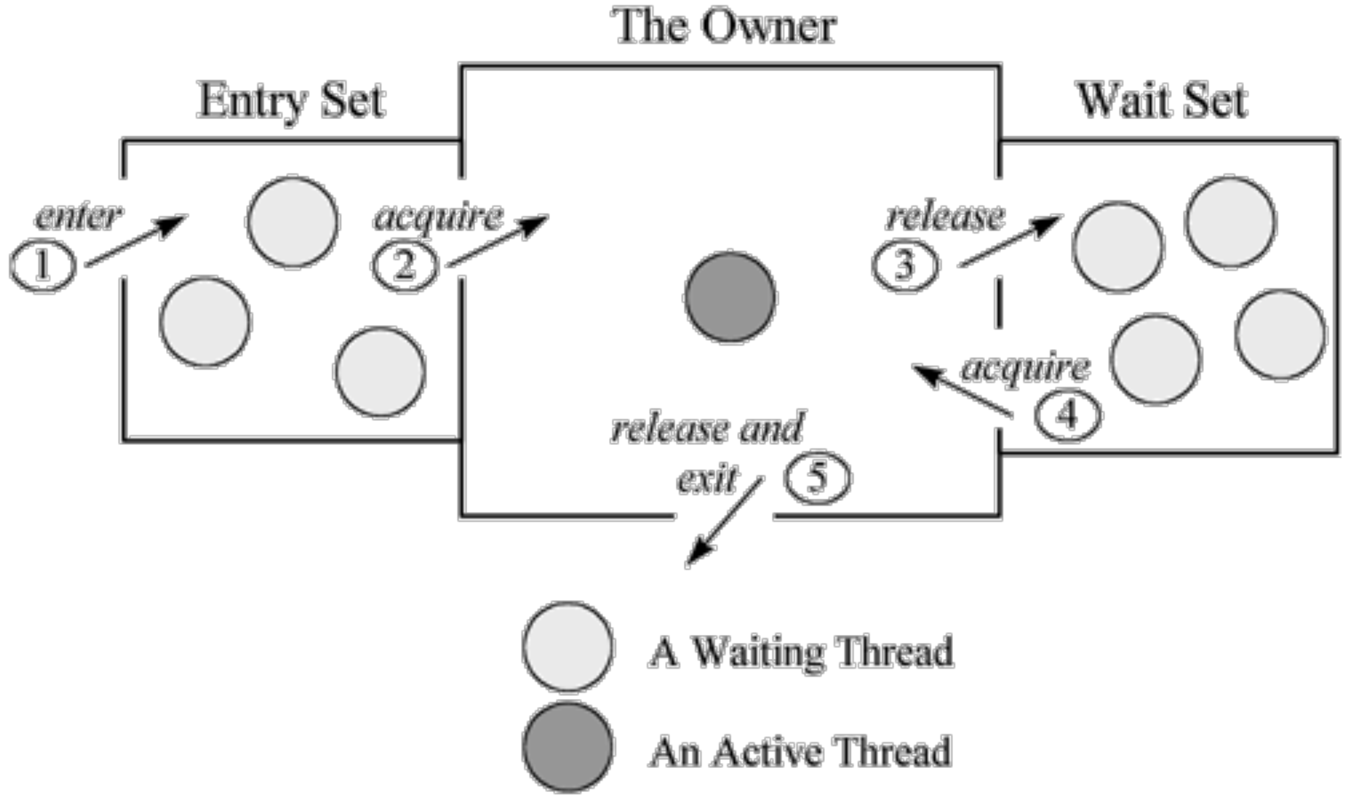
Increments

```
public class ThreadRacingIncrements extends Thread{

    public static int k=0;

    public void run() {
        for(int i=0; i< 10000; i++){
            k++;
        }
    }

    public static void main(String[] args) throws Exception{
        ThreadRacingIncrements t = new ThreadRacingIncrements();
        t.start();
        for(int j=0; j< 10000; j++){
            k++;
        }
        t.join();
        System.out.println("Final K = "+k);
    }
}
```

Thread Racing on Increments



Thread Sync


The Owner
Entry Set
Wait Set
enter
1
acquire
2
release
3
acquire
4
release and exit
5
A Waiting Thread
An Active Thread

synchronized

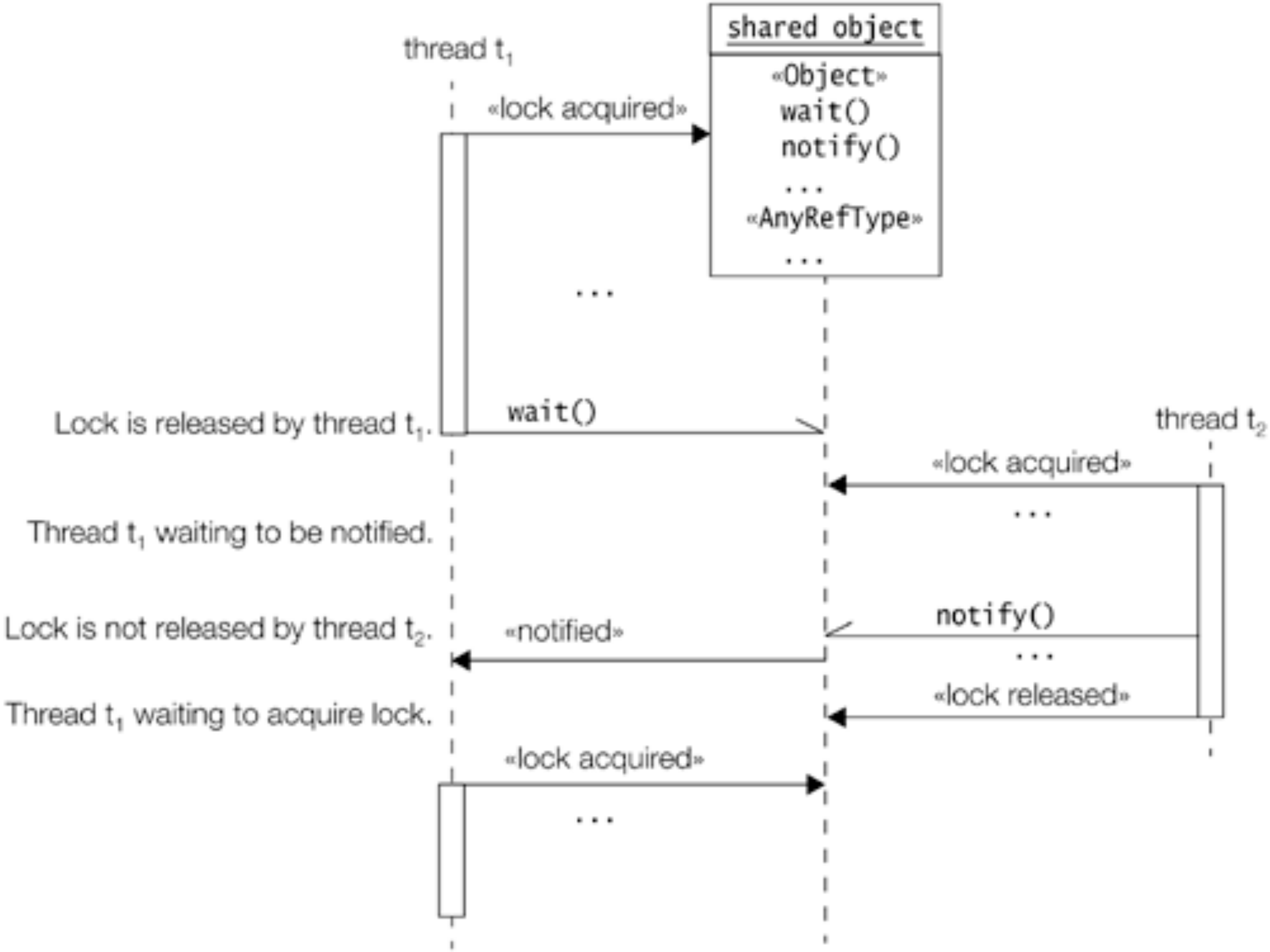
- synchronized is a keyword applicable to methods
    - Instance methods lock the current instance (this)
    - static methods lock the class object
- Also can create blocks like this. Code inside this block is considered to have a lock on obj and any other thread that needs a lock on obj cannot access that execute this this block is over:

```
synchronized (obj) {

}
```

Typical Producer Consumer

```
public static List<String> dataList = new ArrayList<String>();

public void run() {
    InputStreamReader r=new InputStreamReader(System.in);
    BufferedReader br=new BufferedReader(r);
    try{
        while(true){
            System.out.print("Enter text: ");
            String text = br.readLine();
            dataList.add(text);
        }
    }catch(Exception e){}
}

public static void main(String[] args) throws Exception{
    ProducerConsumer pc = new ProducerConsumer();
    pc.start();
    while(true){
        Thread.sleep(100);
        if(dataList.size()>0){
            String str = dataList.get(0);
            dataList.clear();
            System.out.println("Consumer got data: "+str);
        }
    }
}
```

Wait-Notify


shared object
«Object»
wait()
notify()
...
«AnyRefType»
...
thread t1
«lock acquired»
...
wait()
Lock is released by thread t1.
thread t2
«lock acquired»
...
Thread t1 waiting to be notified.
notify()
Lock is not released by thread t2.
«notified»
...
«lock released»
Thread t1 waiting to acquire lock.
«lock acquired»
...

Wait-Notify Prod-Consumer

```
   while(true){
      synchronized (dataList) {
         System.out.print("Enter text: ");
         String text = br.readLine();
         dataList.add(text);
         dataList.notify();
      }
      System.out.println("Notified...");
   }

while(true){
   synchronized (dataList) {
      System.out.println("Waiting...");
      dataList.wait();
      System.out.println("Processing...");
      if(dataList.size()>0){
         String str = dataList.get(0);
         dataList.clear();
         System.out.println("Consumer got data: "+str);
      }
   }
}
```

Wait-Notify Guarantees

- When a thread is notified it is not guaranteed to wake up AS SOON AS the lock is released
    - Any other thread could get the lock
- notifyAll() can be used to wake up more than one thread but no guarantee about the order in which it wakes threads up
- Some threads can get starved for a long time

Atomic & Concurrent Datatypes

- atomic variables for int, boolean, long etc

```
x.addAndGet(2);
x.compareAndSet(2, 3);
x.incrementAndGet();
```

- atomic arrays

```
arr.addAndGet(2, 4);
arr.compareAndSet(2, 3, 4);
```

Concurrent Collections

- ConcurrentHashMap, ConcurrentLinkedQueue, etc
  - Allows iteration and modification at the same time
  - Segments data heaps based on concurrency levels
  - putIfAbsent(K key, V value), remove(Object key,Object value), replace(K key,V oldValue,V newValue), replace(K key,V value)

CopyOnWriteArrayList

- mutative operations (add, set, and so on) are implemented by making a fresh copy of the underlying array
- The iterator will not reflect additions, removals, or changes to the list since the iterator was created.
- No ConcurrentModificationException

BlockingQueue & TransferQueue

- Waits for the queue to become non-empty when retrieving an element

- Waits for space to become available in the queue when storing an element.

- add(e),offer(e),put(e),offer(e, time, unit)

    - ArrayBlockingQueue - has max capacity, supports fairness

- TransferQueue is a blocking queue but has ability to wait for receipt of element (transfer(), hasWaitingConsumer())

Thread Pools

- Thread Pools limit the max number of threads possible

- Conserve CPU resources and eliminates a thread creation overview in high concurrency apps

- Many built in implementation. Lets see an example implementation

Services

- Executor
  - An object that executes submitted Runnable tasks.
- ExecutorService
  - An Executor that provides shutDown(), shutDownNow() methods that can produce a Future for tracking progress of one or more asynchronous tasks.
- CompletionService
  - Producers submit tasks for execution. Consumers take() completed tasks and process their results in the order they complete