



ADVANCED THREADS & CONCURRENCY

MEHRSHAD SAADATINIA

Thread Important Methods (review)

```
Thread t = Thread.currentThread();
```

- Returns the currently executing thread

```
Thread.sleep(milliseconds);
```

- Stops the currently executing thread for specified number of milliseconds

But There Are More...

- Something a specific thread needs to end before the program can continue its execution
- We do this using `join()` method

```
Thread virusScan = new VirusScanThread();  
virusScan.start();  
prepareEmail();  
virusScan.join(); //finish virus scan then continue  
sendEmail();
```

- How to set priority of execution for a thread?

```
MyThread th = new MyThread();  
th.setPriority(Thread.MAX_PRIORITY);  
th.start();
```

$1 < \text{MAX_PRIORITY} < 10$

- **Daemon Threads:**

- Running in the background providing service to main threads and has **NO** significant role independently
- If in a program only daemon threads are alive , JVM terminates the program

```
MyThread th = new MyThread();
```

```
th.setDaemon(true);
```

```
th.start();
```

What is the problem with using multiple threads?

Answer: critical sections

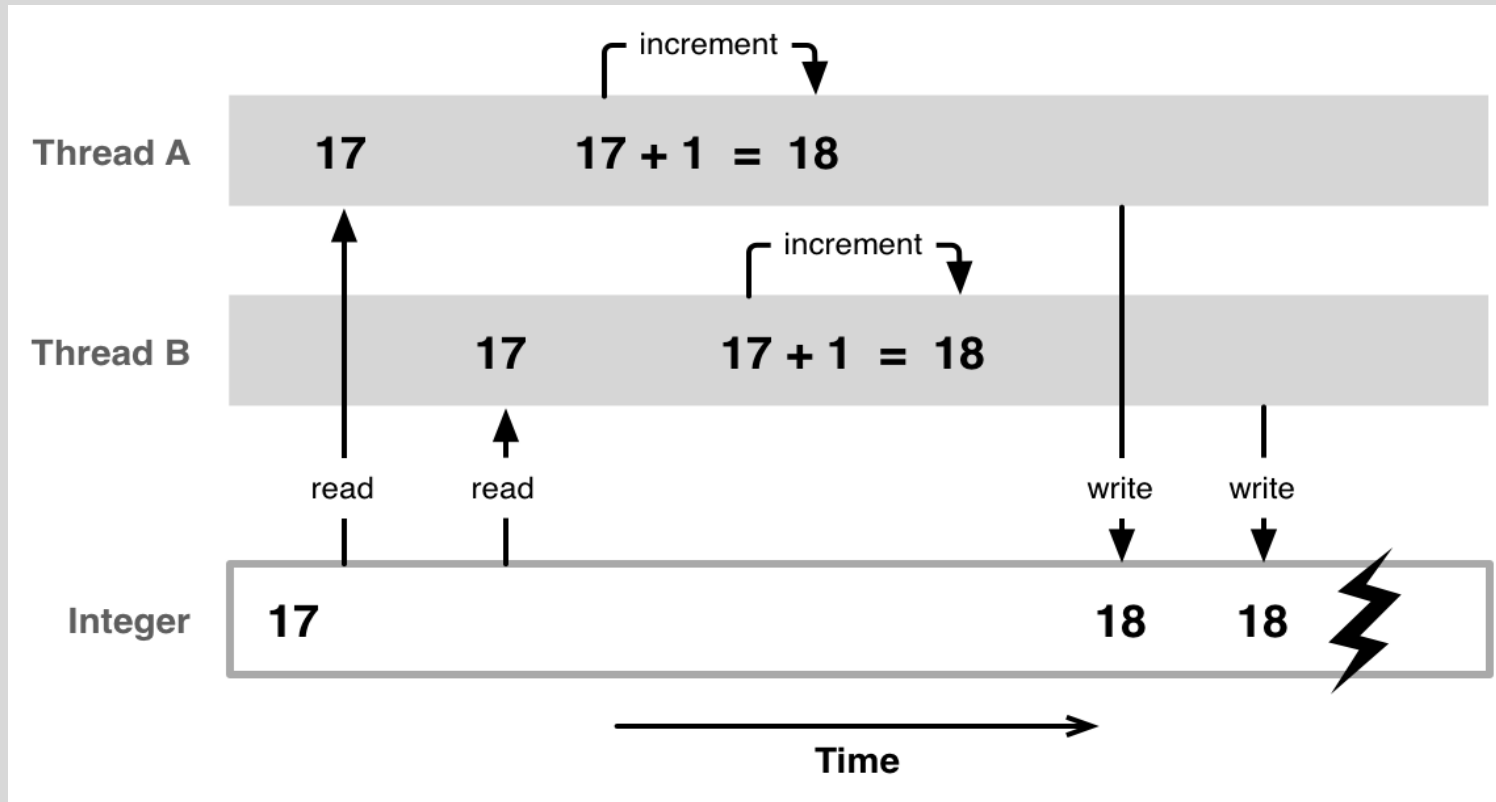
- **Shared resources:**

an entity (object , variable, ...) used in multiple threads in a shared manner

- **Race condition:**

multiple threads trying to access a **shared resource** and at least one of them trying to change it

Race condition:



`X = 17;`
`Thread1: X++;`
`Thread2: X++;`

Synchronized

```
◦ public class BankAccount {  
    private float balance;  
    public synchronized void deposit(float amount)  
    { balance += amount; }  
    public synchronized void withdraw(float amount)  
    { balance -= amount; }  
}
```

Deposit and withdraw methods are **critical sections** here

Two threads can't simultaneously call synchronized methods (enter synchronized block) on the same object

Inter-Thread Communication

- Two threads can communicate using `notify()` and `wait()` methods:

Sometimes a thread needs to wait until other thread notify it to continue execution

There can be `multiple threads` on in waiting state on each object, `notifyAll()` will notify all of them to continue running

notify() & wait()

```
synchronized( lockObject )
{
    while( ! condition )
    {
        lockObject.wait();
    }

    //take the action here;
}
```

```
synchronized(lockObject)
{
    //establish_the_condition;

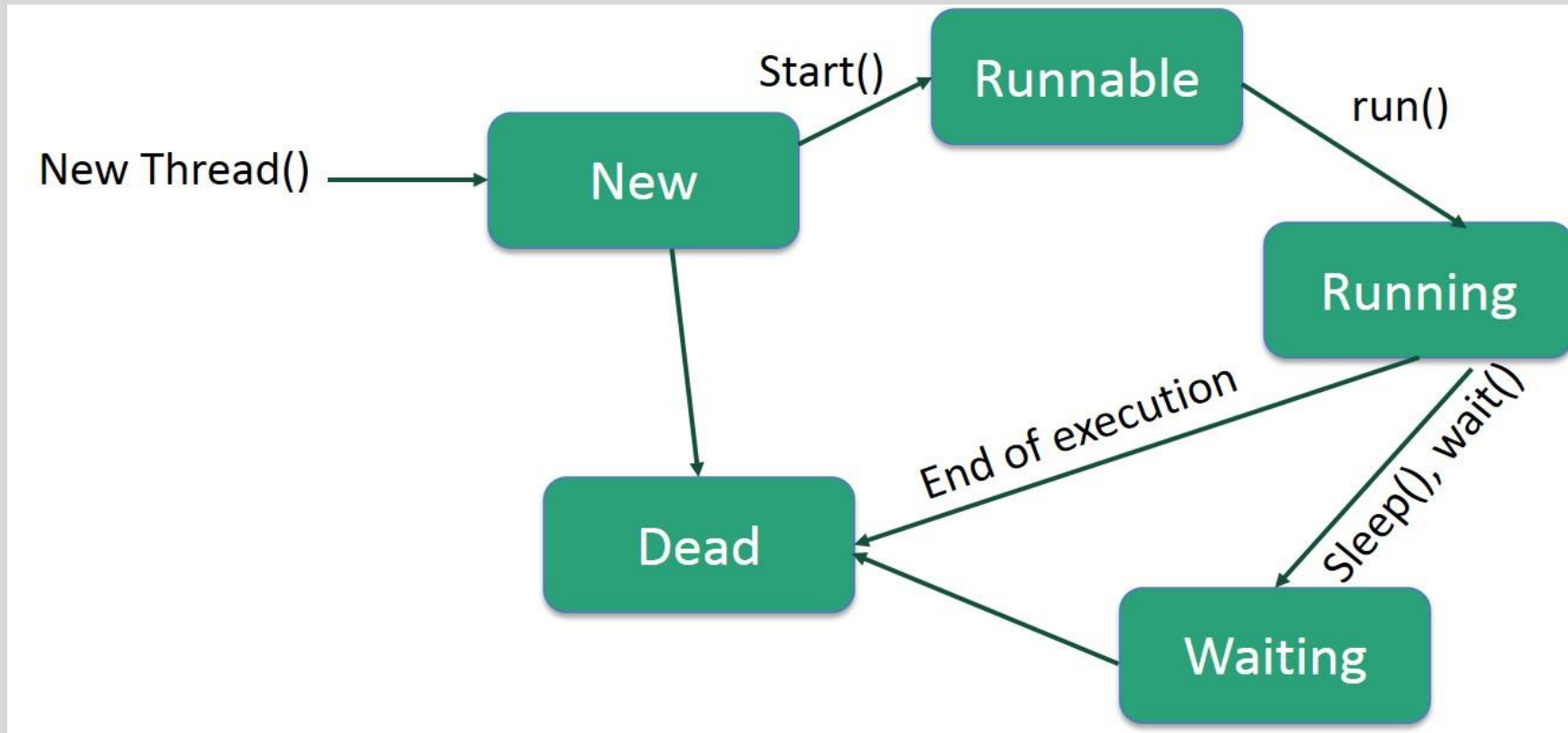
    lockObject.notify();

    //any additional code if needed
}
```

Interrupt()

- A thread might be in a waiting status, if we call interrupt on it , thread exits waiting state and **InterruptedException** will be thrown
- If the thread is not in the sleeping or waiting state then calling the **interrupt()** method performs a normal behavior and doesn't interrupt the thread but sets the interrupt flag to true.

Thread LifeCycle



producer – consumer problem

- 1- **producer** : one or more threads are producing data
- 2- **consumer** : one or more threads are consuming data
- 3- **data source**: data is written in or read from a **shared data source**

Thread Safety

- When using thread safe Objects we don't to use synchronized because they are **thread safe**

Thread safe	Non-thread safe
Vector	ArrayList
StringBuffer	StringBuilder
ConcurrentHashMap	HashMap

And all **immutable** Objects such as String , Integer, ...

Synchronizers

- Used in order to synchronize multiple thread ([java.util.concurrent](#))

1. Semaphore

2. CountdownLatch

3. Exchanger

4. CyclicBarrier

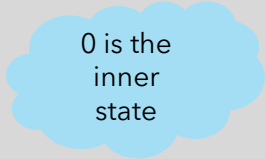
Semaphore

- Methods : `acquire()` , `release()`

Using a number as inner State it determines how many threads can use a shared resource simultaneously

In producer /consumer problem:

```
Semaphore semaphore = new Semaphore(0);
```



0 is the
inner
state

```
semaphore.acquire();  
synchronized (list)  
{ obj = list.remove(0); }
```

```
synchronized(list)  
{ list.add(obj); }  
semaphore.release();
```

Lock interface

- Gives us more control over how we want to execute critical sections

Lock : controlled by programmer explicitly

Synchronized: automatically locks critical section

```
Lock l = new ReentrantLock(); // ReentrantLock implements Lock
l.lock();
try { ... // critical section }
finally { l.unlock(); }
```

ReadWriteLock Interface

- Imagine the following situation in a program:

Some threads **only read** shared resources

Some thread **mutate** (change) shared resources

Using Synchronized block we limit access for **read only threads** as well as **mutator threads**.

- We don't want that , so we use readWriteLock:
- Provides two lock : **readLock()** and **writeLock()**
- **class** ReentrantReadWriteLock **implements** ReadWriteLock

ReentrantReadWriteLock

```
List<Double> list= new LinkedList<>();  
ReadWriteLock lock = new ReentrantReadWriteLock();
```

```
class Reader extends Thread{  
    public void run() {  
        lock.readLock().lock();  
        System.out.println(list.get(0));  
        lock.readLock().unlock();  
    }  
}
```

```
class Writer extends Thread{  
    public void run() {  
        lock.writeLock().lock();  
        list.add(0, Math.random());  
        lock.writeLock().unlock();  
    }  
}
```

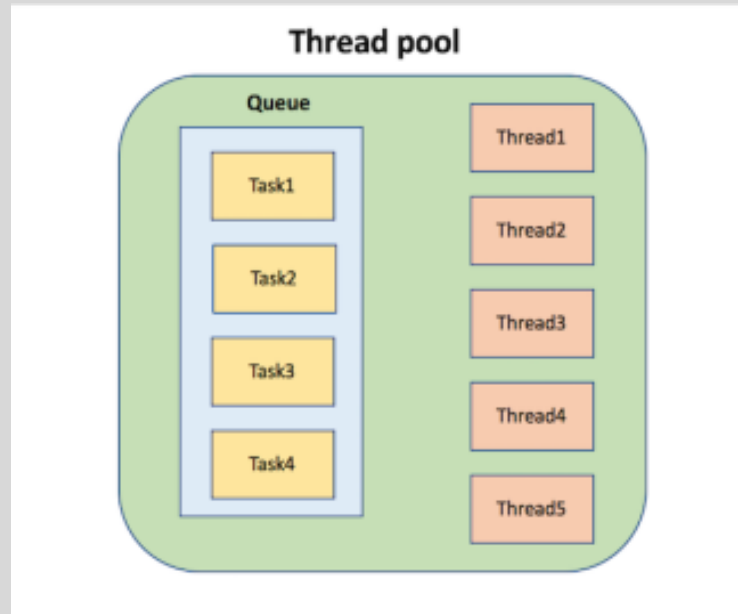
Executors

- Contemporary way of creating threads
program determines that a part needs a new thread (task)

In large scale enterprise application thread management is not done by programmer

Executor framework is used in this case

Thread pool



Limits the number of thread created for specific umber of tasks and manages creation and termination of threads

Executor Methods

- **newSingleThreadExecutor** :

creates a thread pool with only one thread , tasks are executed **sequentially**

- **newFixedThreadPool** :

determines a number of threads to execute tasks, if tasks are more than threads, some task will wait until a previous task is done

- **newCachedThreadPool** :

a new thread is created for each task and if a task finishes , it's thread is kept alive for other tasks in the future

```
Executor e = Executors.newFixedThreadPool(2);
Runnable runnable = new Runnable(){
    public void run() {
        for (int i = 0; i < 4; i++)
            System.out.println(Thread.currentThread().getId()+":"+i);
    }
};
for (int i = 0; i < 3; i++)
    e.execute(runnable);
```

```
9:0
10:0
10:1
10:2
9:1
10:3
9:2
10:0
9:3
10:1
10:2
10:3
```



```
Executor e = Executors.newSingleThreadExecutor();
Runnable runnable = new Runnable(){
    public void run() {
        for (int i = 0; i < 4; i++)
            System.out.println(Thread.currentThread().getId()+":"+i);
    }
};
for (int i = 0; i < 3; i++)
    e.execute(runnable);
```

```
9:0
9:1
9:2
9:3
9:0
9:1
9:2
9:3
9:0
9:1
9:2
9:3
```