Synchronization in java

Inconsistency-Example1

- Consider a method

```
update()
{
n = n+1;
val = f(n);
}
```

Shared Stack Data Structure-Example 2

```
void Stack::Push(Item *item) {
    item->next = top;
    top = item;
}
```

- Suppose two threads, red and blue, share this code and a Stack `s`
- The two threads both operate on `s`
  - each calls `s->Push(…)`
- Execution is interleaved by context switches

Stack Example

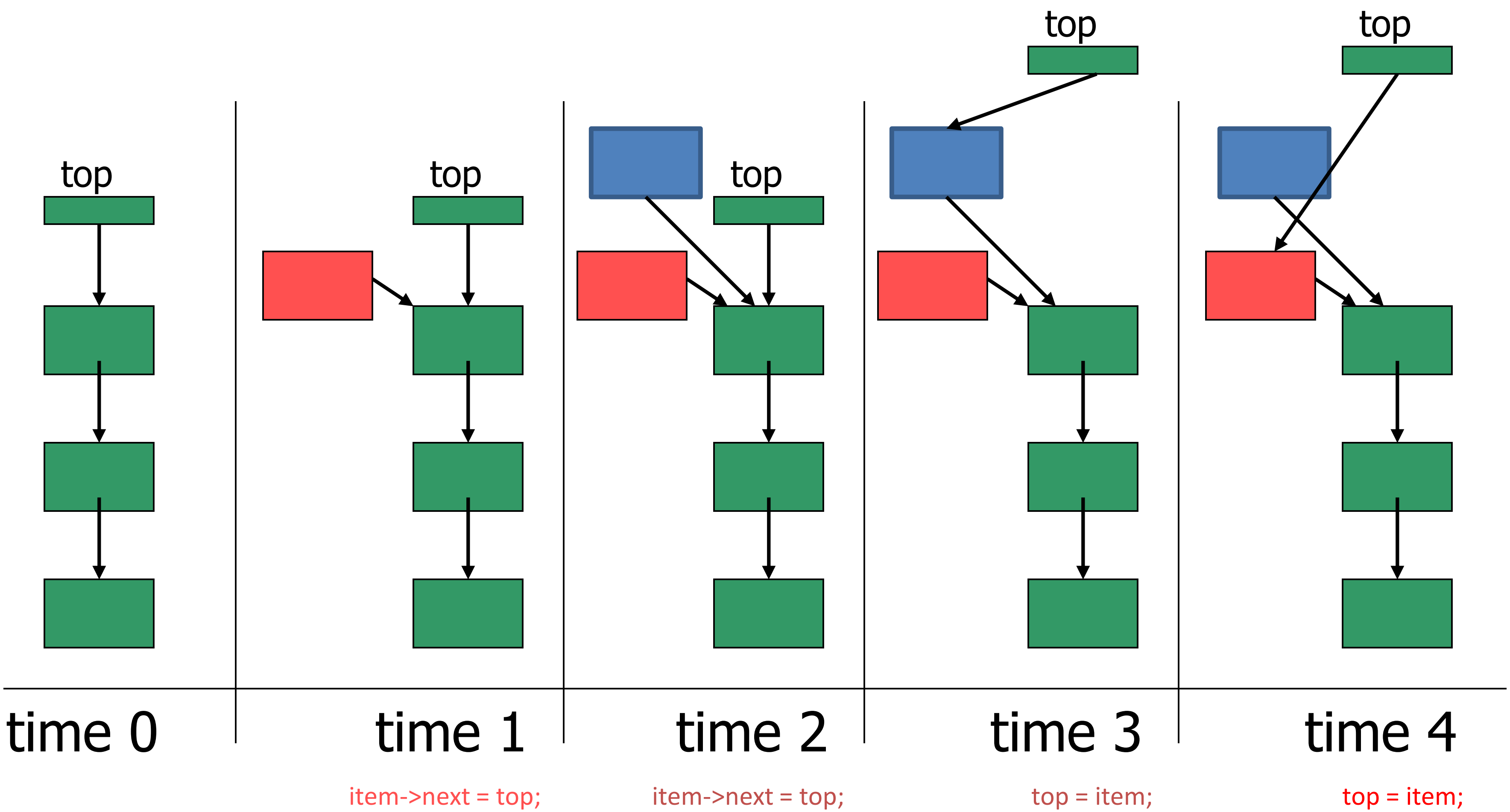
- Now suppose that a context switch occurs at an "inconvenient" time, so that the actual execution order is

context switch from red to blue

1	`item->next = top;`	
2		`item->next = top;`
3		`top = item;`
4	`top = item;`	

context switch from blue to red

Disaster Strikes



Shared Stack Solution

- How do we fix this using locks?
  - See also how to make a lock

```
void Stack::Push(Item *item) {
    lock->Acquire();
    item->next = top;
    top = item;
    lock->Release();
}
```

Correct Execution

- Only one thread can hold the lock

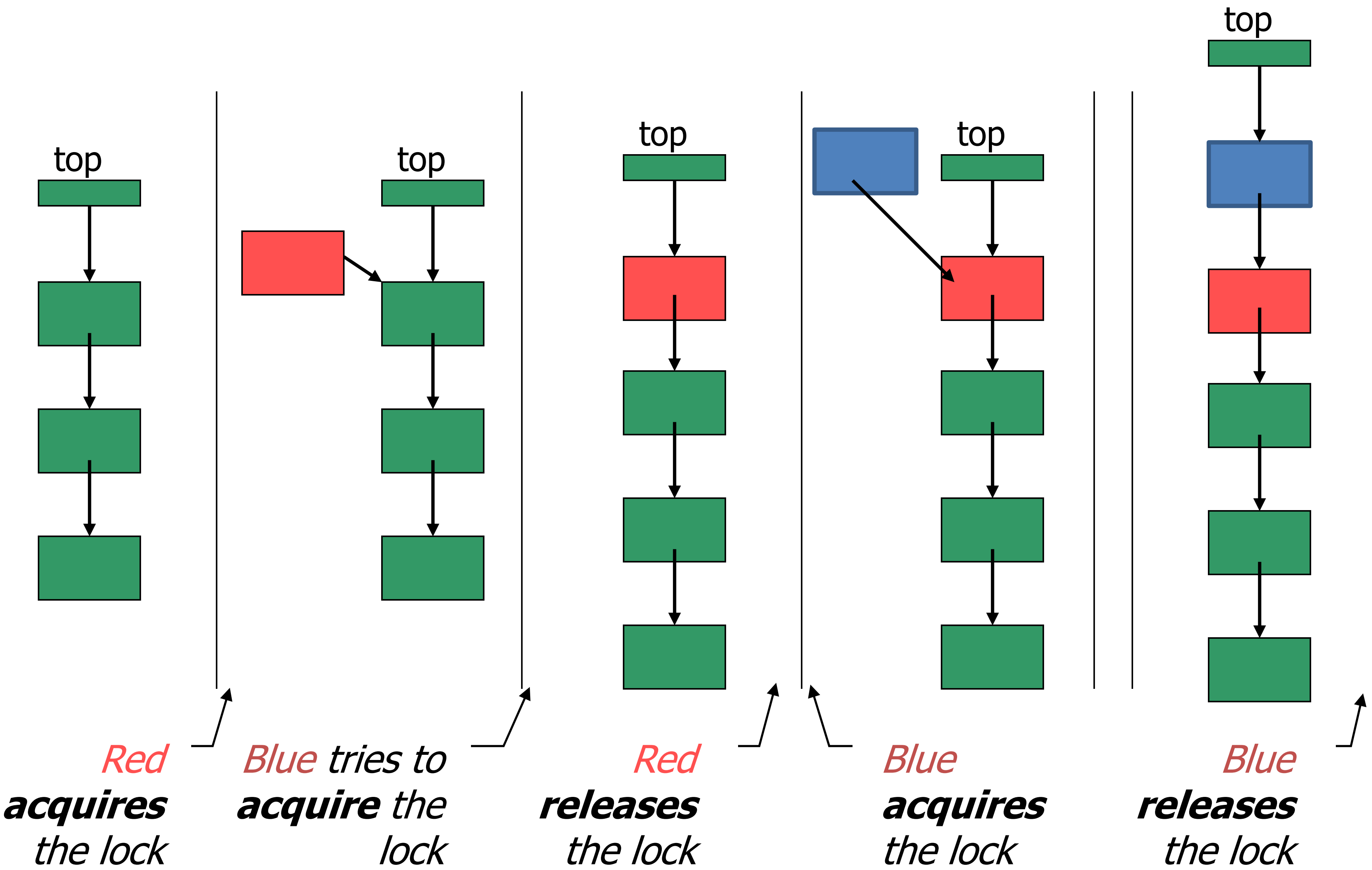
```
lock->Acquire();
item->next = top;

top = item;
lock->Release();
```

```
lock->Acquire();
```

wait for lock acquisition

```
item->next = top;
top = item;
lock->Release();
```

Correct Execution



Synchronization

- When **two or more threads** need access to a **shared resource**
- they need some way to ensure that the resource will be used **by only one thread at a time.**
- The process by which this is achieved is called ***synchronization***.

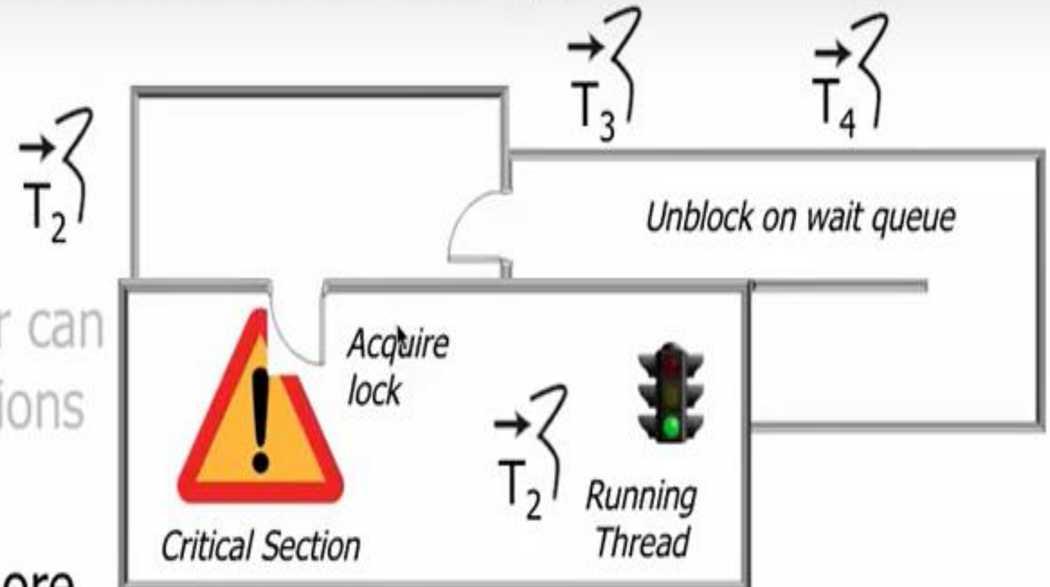
Implementation in java

- In the Java virtual machine, every object and class is logically associated with a monitor.
- To implement the mutual exclusion capability of monitors, a lock (sometimes called a mutex) is associated with each object and class.
- This is called a semaphore in operating systems books, mutex is a binary semaphore.
- If one thread owns a lock on some data, then no others can obtain that lock until the thread that owns the lock releases it.
- It would be not convenient if we need to write a semaphore all the time when we do multi-threading programming.
- Luckily, we don't need to since JVM does that for us automatically.
- To claim a monitor region which means data not accessible by more than one thread, Java provide synchronized statements and synchronized methods.
- Once the code is embedded with synchronized keyword, it is a monitor region.
- The locks are implemented in the background automatically by JVM.

Monitor functionalities

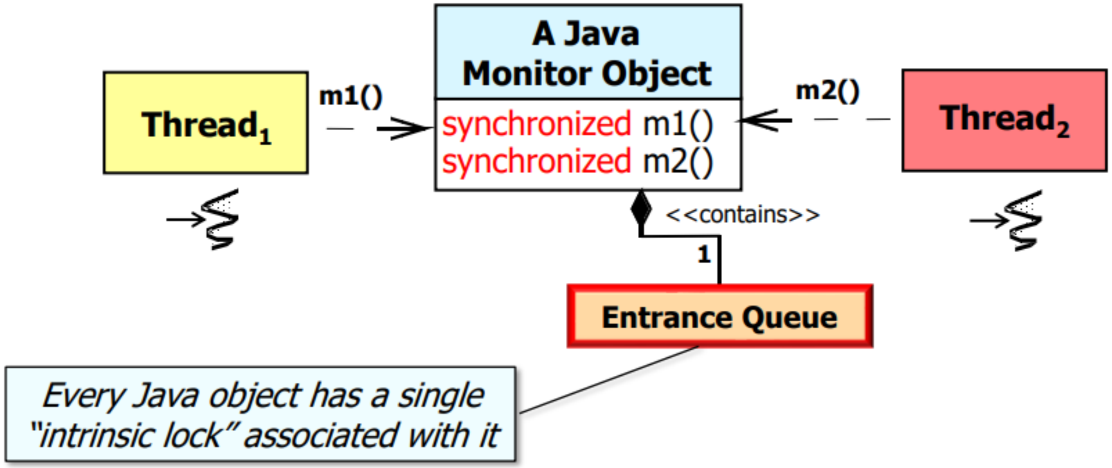
A monitor provides three capabilities to concurrent programs

1. Only one thread at a time has mutually exclusive access to a critical section
2. Threads running in a monitor can block awaiting certain conditions to become true
3. A thread can notify one or more threads that conditions they're waiting on have been met



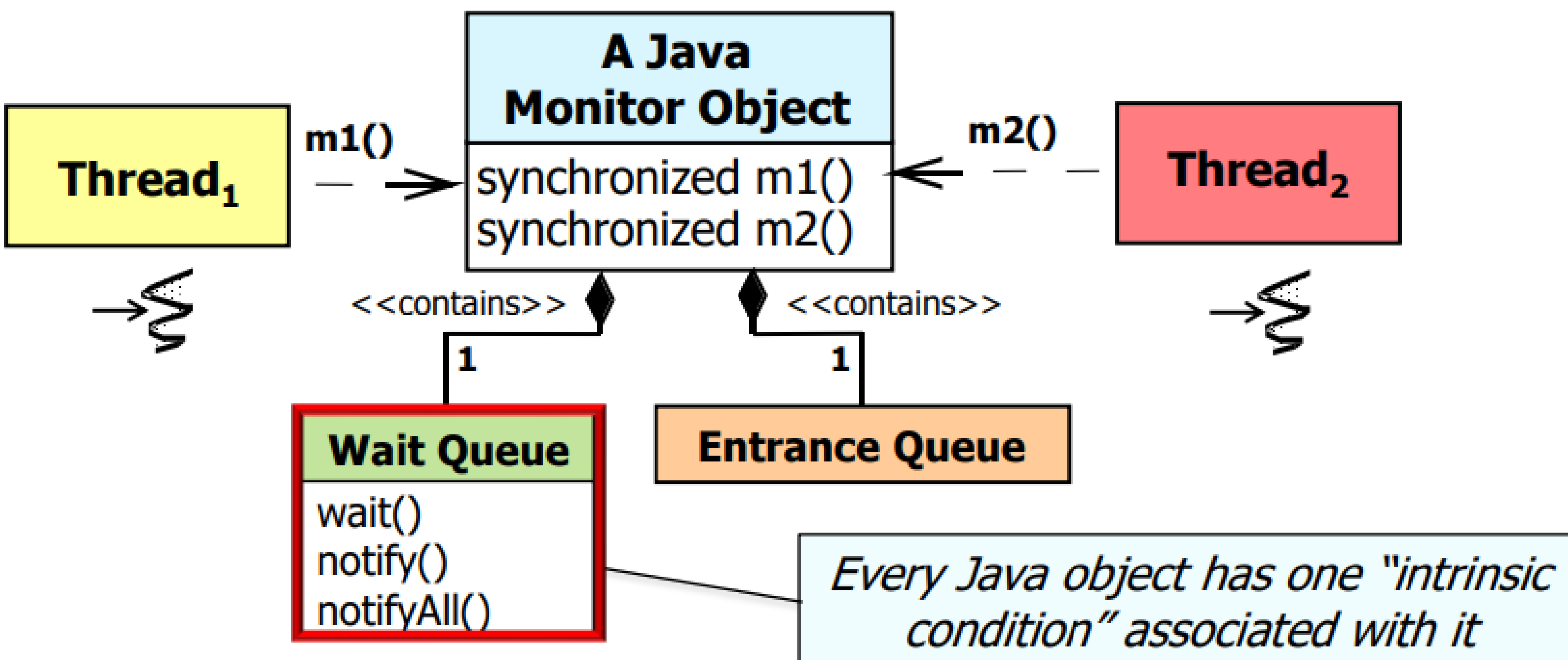
Monitor object

- All objects in Java can be used as built-in monitor objects, which support two types of thread synchronization
  - **Mutual exclusion** – allows concurrent access & updates to shared data without race conditions



Monitor object

- **Coordination** – Ensures computations run properly, e.g., in the right order, at the right time, under the right conditions, etc.

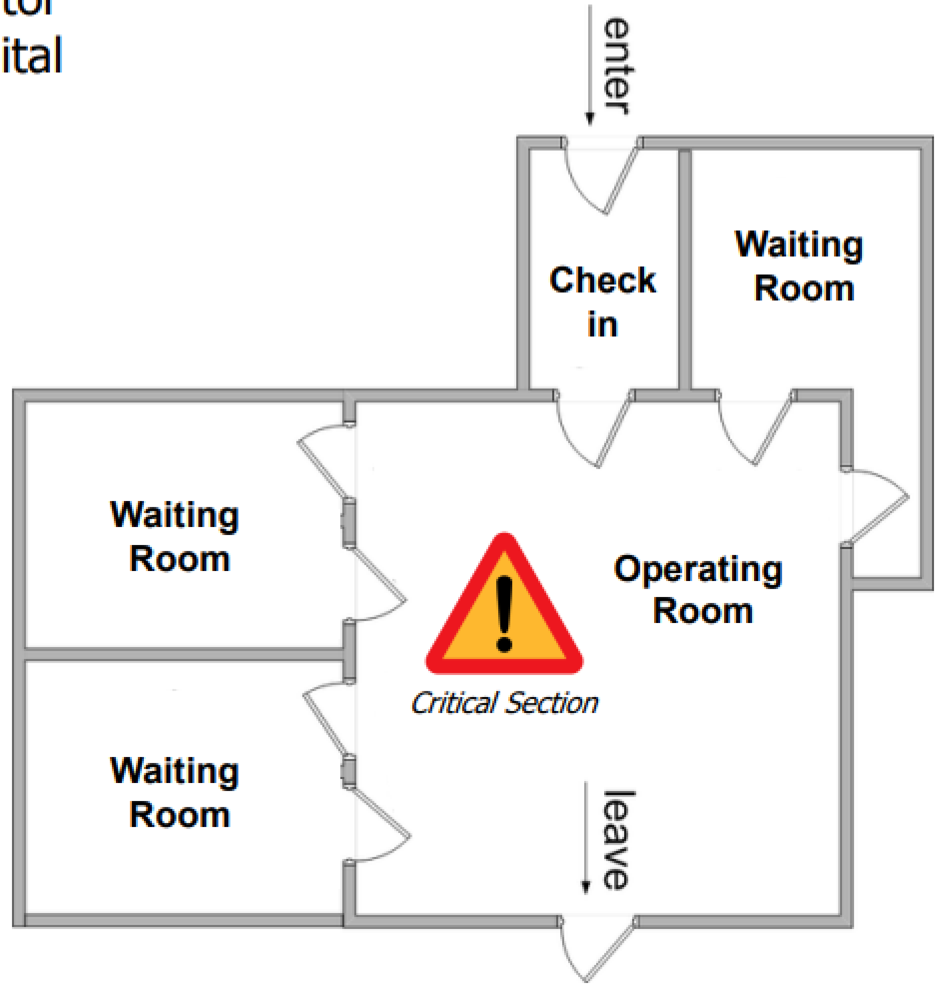


mutual exclusion and *cooperation*.

- Java's monitor supports two kinds of thread synchronization: *mutual exclusion* and *cooperation*.
- **Mutual exclusion**, which is supported in the Java virtual machine via **object locks**, enables multiple threads to independently work on shared data without interfering with each other.
- **Cooperation,** which is supported in the Java virtual machine via the wait() and notify() methods of class Object, enables threads to work together towards a common goal.

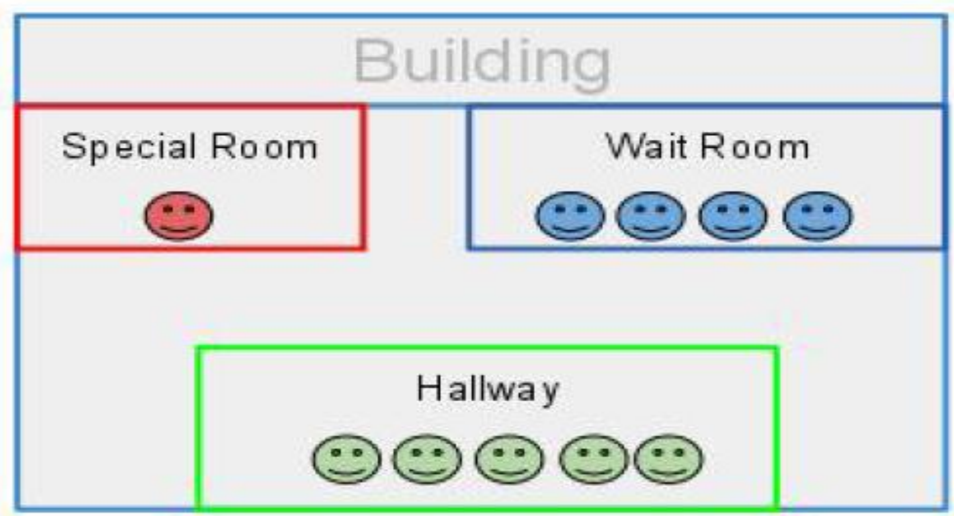
Human Know Use of Monitors

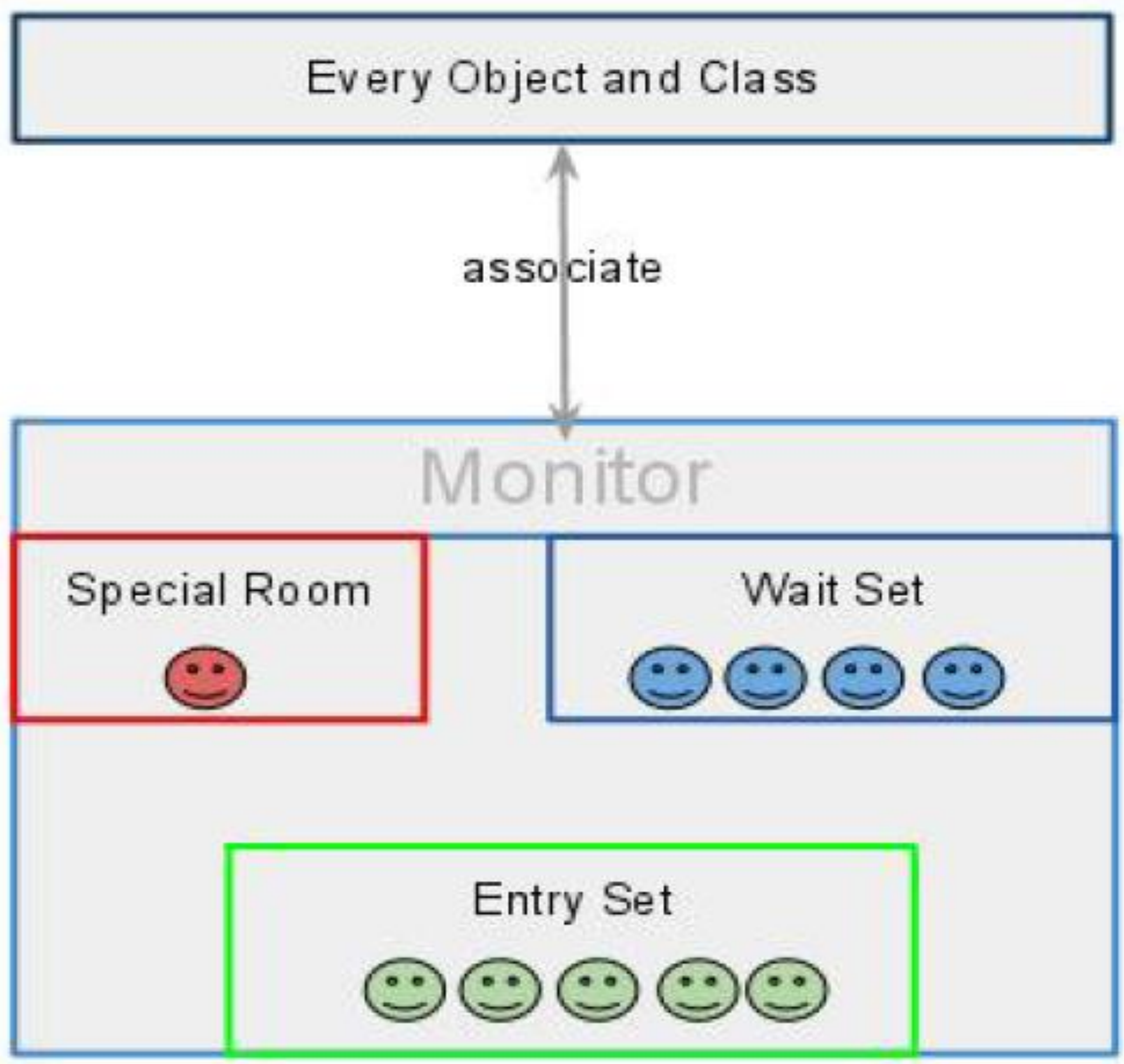
- A human known use of a monitor is an operating room in a hospital



Monitor

- A monitor can be considered as a building which contains a special room.
- The special room can be occupied by only one customer(thread) at a time. The room usually contains some data and code.




Every Object and Class
associate
Monitor
Special Room
Wait Set
Entry Set

MONITOR

- Any object (except built in object-int,float) can be a monitor object(object class-wait,notify)
- Key to synchronization is the concept of the monitor (also called a *semaphore)*
- Internally synchronized is implemented by means of lock (JVM does this automatically).
- Only one thread can *access the lock* at a given time.
- When a thread acquires a lock, it is said to have *entered the* monitor.
- *Other threads will be in wait state to execute the synchronized method*

Synchronized vs Non synchronized

- Synchronized –One thread can only access-Object state change –add,update,delete and modify
- Ex-ticket booking
- Non synchronized methods-multiple threads can access-no change in state of the object-read operations.
- Ex-check availability of ticket

1)Synchronization-Mutual exclusion

When multiple users act on the same java object data inconsistency problem

a)Synchronized method-synchronization occurs at method level

b)Synchronized block-more finer level

Synchronized code

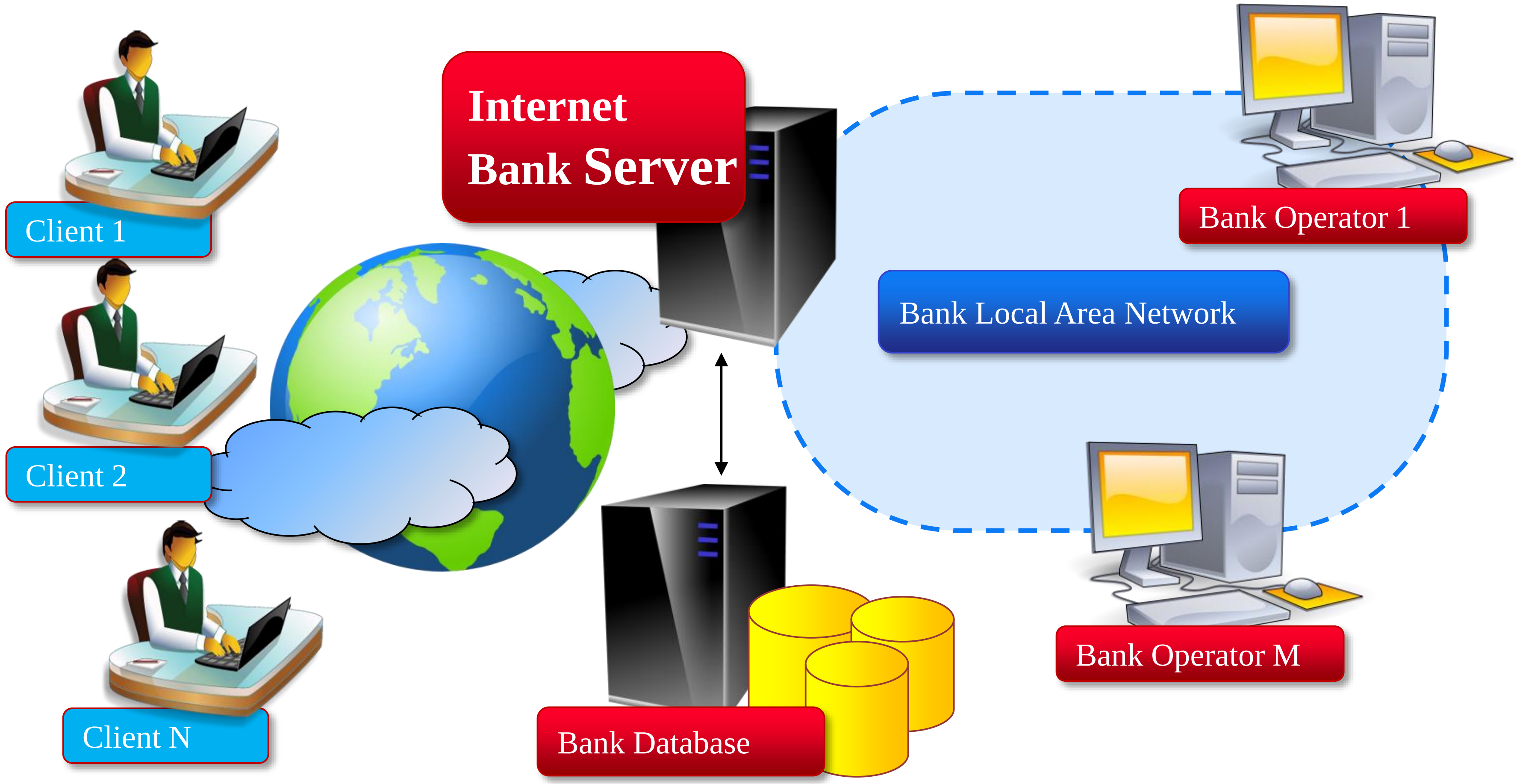
- The Java platform associates a **lock with every object that has `synchronized` code**
- A **method or a code block {...} can be synchronized**
- The **lock is acquired before the block is entered and released when the block is exited**

- A Java method may be synchronized, which guarantees that at most one thread can execute the method at a time.
- Other threads wishing access, are forced to wait until the currently executing thread completes.

EX1-Accessing Shared Resources

- Applications access to shared resources need to be coordinated.
  - Printer (two person jobs cannot be printed at the same time)
  - Simultaneous operations on your bank account.
  - Can the following operations be done at the same time on the same account?
    - Deposit()
    - Withdraw()
    - Enquire()

Online Bank: Serving Many Customers and Operations



Shared Resources

- If one thread tries to read the data and other thread tries to update the same data, it leads to inconsistent state.
- This can be prevented by synchronising access to the data.
- Use “synchronized” method:
  - public synchronized void update()
  - {
    - …
  - }

Monitor (shared object access): serializes operation on shared objects

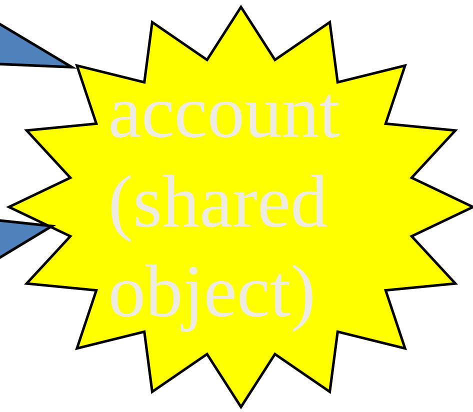
```
class Account {   // the 'monitor'
  int balance;

     // if 'synchronized' is removed, the outcome is unpredictable
     public synchronized void deposit( ) {
        // METHOD BODY : balance += deposit_amount;
     }

      public synchronized void withdraw( ) {
         // METHOD BODY: balance -= deposit_amount;
      }
      public synchronized void enquire( ) {
         // METHOD BODY: display balance.
      }
}
```

the driver: 3 Threads sharing the same object

```
class InternetBankingSystem {
        public static void main(String [] args  ) {
            Account accountObject = new Account ();
            Thread t1 = new Thread(new MyThread(accountObject));
            Thread t2 = new Thread(new YourThread(accountObject));
            Thread t3 = new Thread(new HerThread(accountObject));
            t1.start();
            t2.start();
            t3.start();
          // DO some other operation
        } // end main()
}
```

Shared account object between 3 threads

```
class MyThread implements Runnable  {
 Account account;
       public MyThread (Account s) {  account = s;}
       public void run() { account.deposit(); }
} // end class MyThread

class YourThread implements Runnable  {
 Account account;
       public YourThread (Account s) { account = s;}
       public void run() { account.withdraw(); }
} // end class YourThread

class HerThread implements Runnable  {
 Account account;
       public HerThread (Account s) { account = s; }
       public void run() {account.enquire(); }
} // end class HerThread
```



EX-2 Synchronization in Java

```
class Table{
synchronized void printTable(int n){
   for(int i=1;i<=5;i++){
     System.out.println(n*i);
     try{
      Thread.sleep(400);
     }catch(Exception e){System.out.println(e);}
   }

 }
}
```

Thread1-5 table

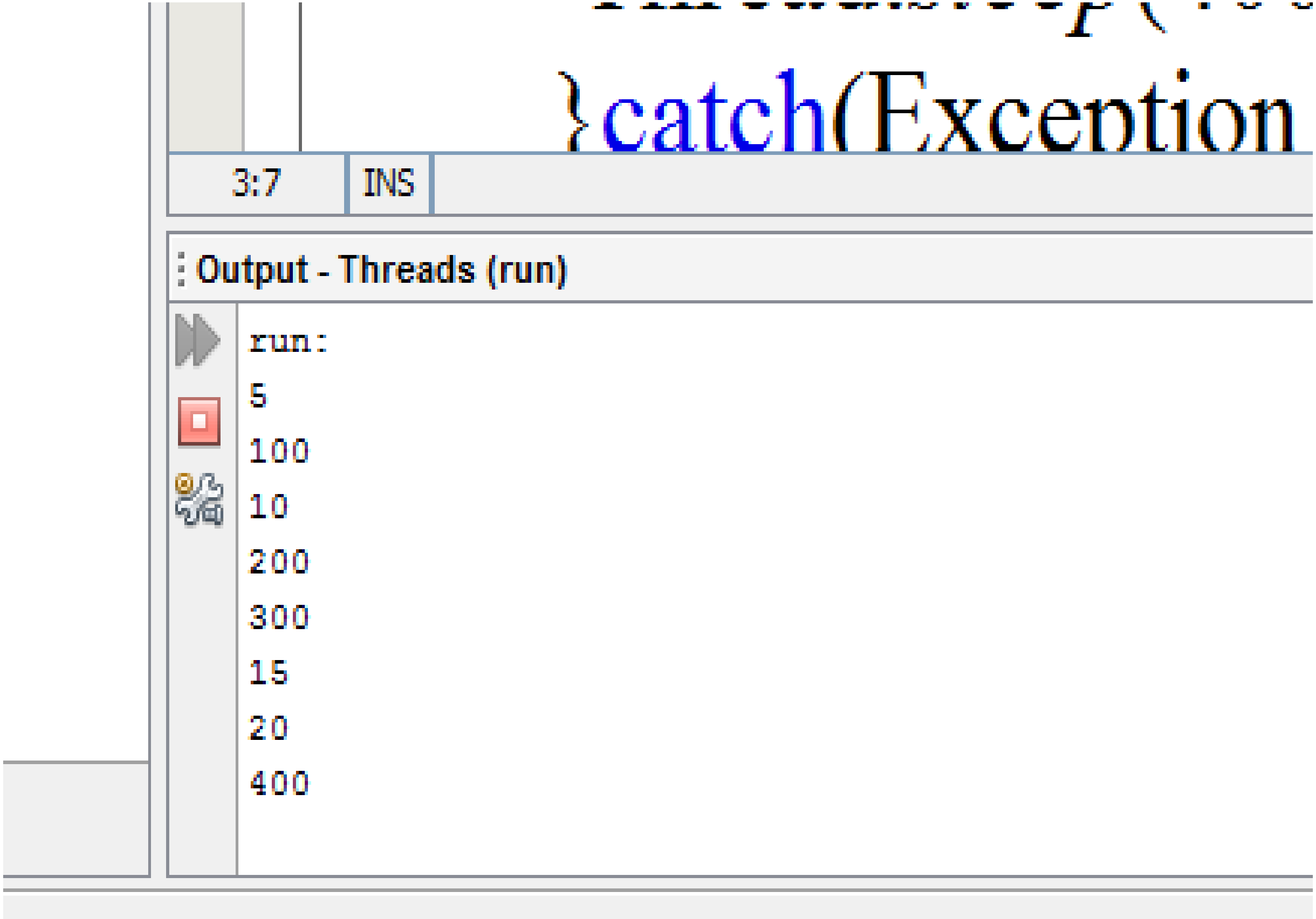
```
class MyThread1 extends Thread{
Table t;
MyThread1(Table t){
this.t=t;
}
public void run(){
t.printTable(5);
}
}
```

Thread2-100table

```
class MyThread2 extends Thread{
Table t;
MyThread2(Table t){
this.t=t;
}
public void run(){
t.printTable(100);
}
}
```

Main

```
class Sync{
public static void main(String args[]){
Table obj = new Table();//only one object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
}
}
```


}catch(Exception
3:7
INS
Output - Threads (run)
run:
5
100
10
200
300
15
20
400

Class level lock

- Every class in Java has a unique lock which is nothing but **class level lock**.
- If a thread wants to execute a static synchronized method, then the thread requires a class level lock.
- Class level lock prevents **multiple threads to enter a synchronized block in any of all available instances of the class on runtime**.
- If a thread wants to execute a static synchronized method, then the thread requires a class level lock.
- Once a thread got the class level lock, then it is allowed to execute any static synchronized method of that class.
- Once method execution completes automatically thread releases the lock.
- Thread can acquire the lock at a class level by two methods namely
  - Using the synchronized static method.
  - Using synchronized block.

Example

```
public class ClassLevelLockingExample extends Thread {
	public static void main(String[] args) {
		Thread t1 = new ClassLevelLockingExample();
		Thread t2 = new ClassLevelLockingExample();
		t1.start();
		t2.start();
	}
	public void run() {
		ClassLevelLockingExample.classLevelLockMethod();
	}
	private static synchronized void classLevelLockMethod() {
		try {
			System.out.println("Entered into the Class Level Lock Method");
			Thread.sleep(10000);
			System.out.println("After this Statement Only Any Other thread can enter 
	this method");
		} catch (InterruptedException e) {

			e.printStackTrace();
		}
	}
}
```

Synchronized block

- The Java synchronized statement is a form of monitor
- general form of the **synchronized statement:**

1.synchronized(*object) {//2.synchronized(this)*
//3.synchronized(classname.class)
//get lock(object)
// statements to be synchronized
//free lock(object)
}

object is a reference to the object being synchronized.

Synchronized statements are also useful for improving concurrency with **fine-grained synchronization.**

Synchronized block

- synchronized statement to execute only when it has the lock for obj.
- Thus two different threads can never simultaneously execute the body of a synchronized statement because two threads can't simultaneously hold obj's lock.
- Synchronized blocks don't offer any mechanism of a waiting queue and after the exit of one thread, any thread can take the lock. This could lead to starvation of resources for some other thread for a very long period of time.
- There's no fairness in the thread access –not flexible

Example-synchronized block

```
class Table{

 void printTable(int n){
      //extra code
   synchronized(this){//synchronized block
     for(int i=1;i<=5;i++){
      System.out.println(n*i);
      try{
       Thread.sleep(400);
      }catch(Exception e){System.out.println(e);}
     }
   }
//extra code
 }//end of the method
}
```

Extrinsic locks

- synchronize any object we get an **Intrinsic Lock** on that object.
- *java.util.concurrent.locks* package provides us with lock objects.
- some explicit locking classes which can be used to replace the the intrinsic locks.
- ***void lock()*** – Acquire the lock if it's available. If the lock isn't available, a thread gets blocked until the lock is released

Extrinsic locks

- **void *unlock()*** unlocks the *Lock* instance.
- *ReadWriteLock* declares methods to acquire read or write locks:
- ***Lock readLock()*** returns the lock that's used for reading.
- ***Lock writeLock()*** returns the lock that's used for writing.

Example

```
class Counter{

  private Lock lock;
  private int count = 0;
 public Counter(Lock myLock)
 { this.lock = myLock; }

  public int inc(){
    lock.lock();
    int newCount = ++count;
    lock.unlock();
    return newCount;
  }
}
```

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
public class LockDemo
{
public static void main(String args[])
{
final Counter myCounter = new Counter(new ReentrantLock());
Runnable r = new Runnable()
{
@Override public void run()
{
System.out.printf("Count at thread %s is %d %n", Thread.currentThread().getName(),
myCounter.inc());
}
};
Thread t1 = new Thread(r, "T1");
Thread t2 = new Thread(r, "T2");
Thread t3 = new Thread(r, "T3");
//starting all threads
t1.start(); t2.start(); t3.start();
} }
```

Output:

- Count at thread T2 is 2
- Count at thread T3 is 3
- Count at thread T1 is 1

Count at thread T1 is **1**
Count at thread T2 is **2**
Count at thread T3 is **3**

Starvation and Fairness

Java's synchronized blocks makes no guarantees about the sequence in which threads trying to enter them are granted access.
Therefore, if many threads are constantly competing for access to the same synchronized block, there is a risk that one or more of the threads are never granted access - that access is always granted to other threads. This is called **starvation**.
To avoid this a Lock should be fairness.
Threads can experience deadlocks when locks aren’t unlocked after use.