Course code : **CSE2005**

Course title : **Object Oriented Programming**

Thread Synchronization

Objectives

This session will give the knowledge about

- Thread Synchronization
- Inter-thread communication

Synchronization

Threads often need to share data. Different threads shouldn’t try to access and change the same data at the same time

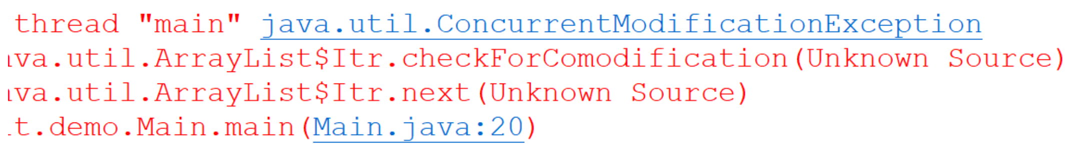
Threads must therefore be synchronized

Key to synchronization is the concept of the monitor (also called a semaphore).

For example, imagine a Java application where one thread (which let us assume as Producer) writes data to a data structure, while a second thread (consider this as Consumer) reads data from the data structure

Synchronization

```
public class Main {
        public static void main(String[] args) {
                ArrayList lt=new ArrayList<>();
                lt.add(23);                     lt.add(44);
                lt.add(12);                     lt.add(30);
                Iterator it=lt.iterator();
                while(it.hasNext()){
                                lt.remove(it.next());
                }
                System.out.println(lt);
        }
}
```



```
 thread "main" java.util.ConcurrentModificationException
va.util.ArrayList$Itr.checkForComodification(Unknown Source)
va.util.ArrayList$Itr.next(Unknown Source)
t.demo.Main.main(Main.java:20)
```

Synchronization

```
public class Main {
        public static void main(String[] args) {
                ArrayList lt=new ArrayList<>();
                lt.add(23);             lt.add(44);
                lt.add(12);             lt.add(30);
                Integer ele=new Integer(12);
                if(lt.contains(ele))
                        lt.remove(ele);
                System.out.println(lt);
        }
}
```

Synchronization

```
public class Main {
        public static void main(String[] args) {
                ArrayList lt=new ArrayList<>();
                lt.add("cse");
                lt.add("ece");
                lt.add("eee");
                lt.add("mech");
                if(lt.contains("eee"))
                        lt.remove("eee");
                System.out.println(lt);
        }
}
```

Synchronization – producer/consumer example

The producer/consumer problem describes two processes, the producer and the consumer, who share a common, fixed-size buffer used as a queue.

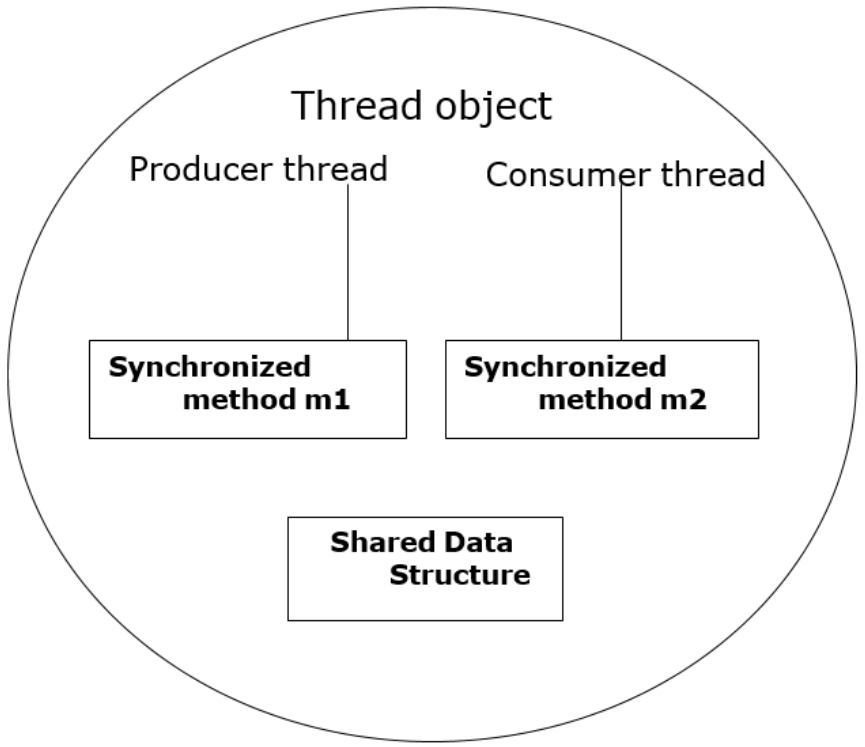
The producer's job is to generate data, put it into the buffer, and start again.

At the same time, the consumer is consuming the data (i.e., removing it from the buffer), one piece at a time.

The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

Synchronization – producer/consumer example

This example use concurrent threads that share a common resource: a data structure.



Concurrency Control Mechanism

- The current thread operating on the shared data structure, must be granted mutually exclusive access to the data
- The current thread gets an exclusive lock on the shared data structure, or a mutex
  - A monitor (control mechanism) is an object, which is used as a concurrency control mechanism. It is used as a mutually exclusive lock(mutex).

Concurrency Control Mechanism

  - When a thread enters a monitor (synchronized method), all other threads, that are waiting for the monitor of same object, must wait until that thread exits the monitor

- A mutex is a concurrency control mechanism used to ensure the integrity of a shared data structure

- Mutex is not assured, if, the methods of the object accessed by competing threads are ordinary methods

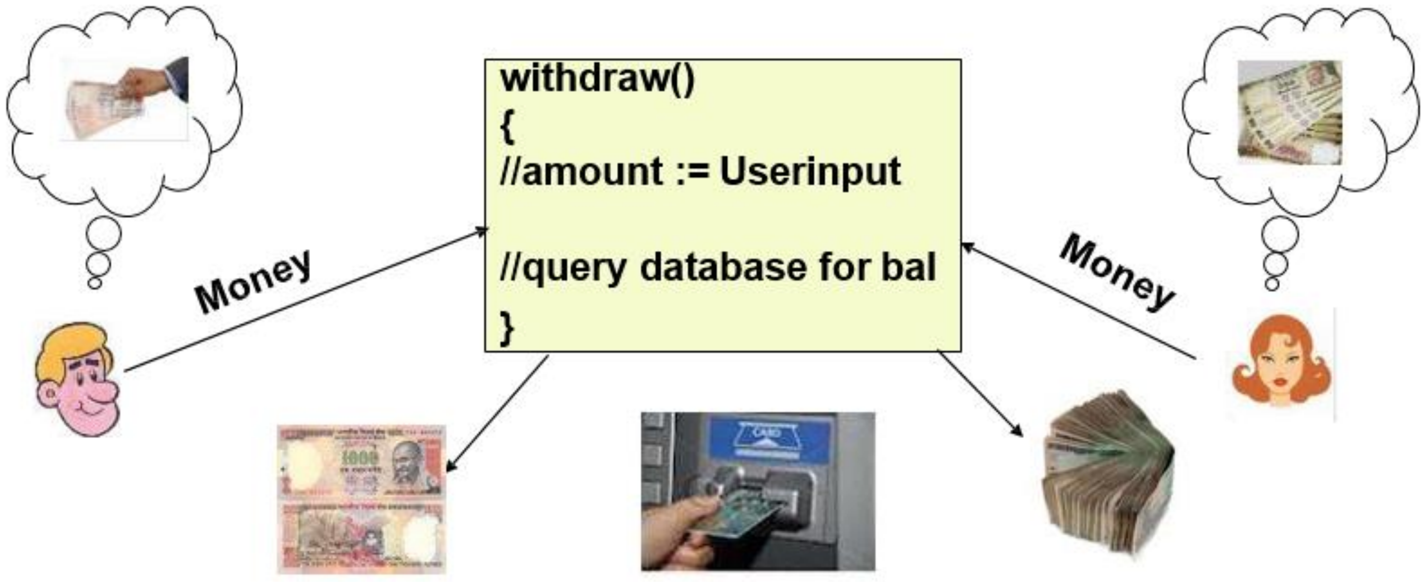
Concurrency Control Mechanism

- It might lead to a race condition when the competing threads will race each other to complete their operation
- A race condition can be prevented by defining the methods accessed by the competing threads as synchronized.
- Synchronized methods are an elegant variation on a time-tested model of inter process-synchronization: the monitor
- Once a thread is inside a synchronized method, no other thread can call any other synchronized method on the same object.

Using Synchronized Methods

- Implementation of concurrency control mechanism is very simple because every Java object has its own implicit monitor associated with it
- If a thread wants to enter an object’s monitor, it has to just call the synchronized method of that object
- While a thread is executing a synchronized method, all other threads that are trying to invoke that particular synchronized method or any other synchronized method of the same object, will have to wait

Using Synchronized Methods

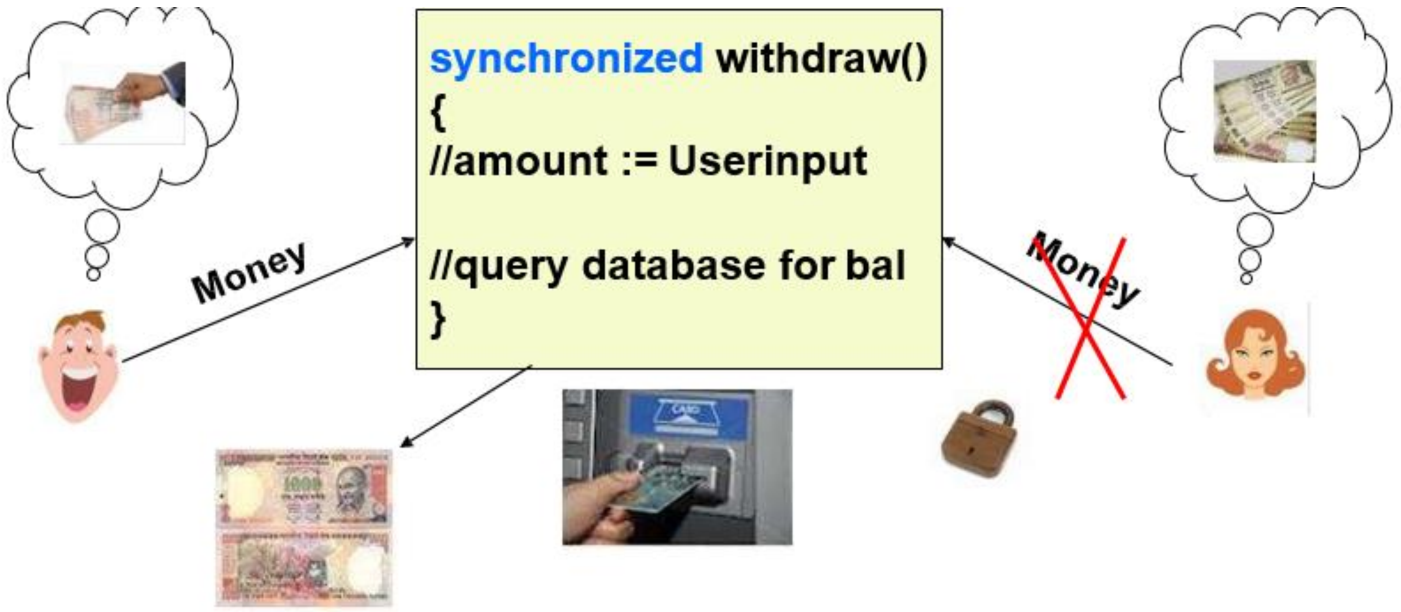


```
withdraw()
{
//amount := Userinput

//query database for bal
}
```

- John and Mary are withdrawing money from their joint account. They are withdrawing at the same instance from different ATMs. What will be the concern here?
- Each transaction occurs independently through different threads. Since both transactions are unaware of the other, this may lead to inconsistent state. How to avoid this situation?

Using Synchronized Methods



```
synchronized withdraw()
{
//amount := Userinput

//query database for bal
}
```

- The concern shared in the previous slide can be easily handled using synchronization. When John wants to withdraw cash from his account, the thread that is responsible for handling this transaction gets an exclusive lock(monitor), which ensures that Mary cannot access this method concurrently. This mechanism comes in the form of synchronized method

Synchronization

- Every object in Java has a lock
- Using synchronization enables the lock and allows only one thread to access that part of code
- Synchronization can be applied to:
  - A method : `public synchronized withdraw(){…}`
  - A block of code: `synchronized (objectReference){…}`
- Synchronized methods in subclasses use same locks as their superclasses

The Synchronized Statement

Syntax for block of code:

```
public void run()
{
        synchronized(obj)  {
                obj.withdraw(500);
        }
 }
```

Synchronized method - Example

```
class Account {
        private int balance;
        public Account(int amount) {
                balance = amount;
        }
        public synchronized void withdraw(int bal) {
                try {
                        Thread.sleep(1000);
                } catch (InterruptedException ex) {
                        System.out.println(" Interrupted .." + ex);
                }
```

Synchronized method - Example

```
                balance = balance - bal;
                System.out.println("Balance remaining:::" + balance);
        }
}
class Joint_Account implements Runnable {
        Account account;
        public Joint_Account(Account account) {
                this.account = account;
        }
        public void run() {
                account.withdraw(500);
        }
```

Synchronized method - Example

```
}
public class ThreadDemo {
        public static void main(String args[]) {
                Account a1 = new Account(10000);
                Joint_Account partner1 = new Joint_Account(a1);
                Joint_Account partner2 = new Joint_Account(a1);
                Thread t1 = new Thread(partner1);
                Thread t2 = new Thread(partner2);
                t1.start();
                t2.start();
        }
}
```

Inter-thread communication

Inter-thread communication or Co-operation is all about allowing synchronized threads to communicate with each other.

Cooperation (Inter-thread communication) is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed. It is implemented by following methods of Object class:

- wait()
- notify()
- notifyAll()

Inter-thread communication

wait() method

Causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.

public final void wait()throws InterruptedException

waits until object is notified.

public final void wait(long timeout)throws InterruptedException

waits for the specified amount of time.

Inter-thread communication

notify() method

Wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation.

notifyAll() method

Wakes up all threads that are waiting on this object's monitor.

Inter-thread communication

```
class Customer {
		int amount = 10000;
		synchronized void withdraw(int amount) {
		System.out.println("going to withdraw...(amount to withdraw) "+amount);
				if (this.amount < amount) {
		System.out.println("Less balance; waiting for deposit...(current balance) "+this.amount);
						try {
								wait();
						} catch (Exception e) {
						}
				}
```

Inter-thread communication

```
                this.amount -= amount;
        System.out.println("withdraw completed...(current balance) "+this.amount);
        }

        synchronized void deposit(int amount) {
                System.out.println("going to deposit...(deposit amount) "+amount);
                this.amount += amount;
                System.out.println("deposit completed... (current balance) "+this.amount);
                notify();
        }
}
```

Inter-thread communication

```
public class ThreadDemo {
        public static void main(String args[]) {
                final Customer c = new Customer();

                new Thread() {
                        public void run() {
                                c.withdraw(15000);
                        }
                }.start();
```

Inter-thread communication

```
                    new Thread() {
                              public void run() {
                                        c.deposit(10000);
                              }
                    }.start();
          }
}
```

Summary

We have discussed about

- Thread Synchronization
- Inter-thread communication