Multithreading

Thread Synchronization

Agenda

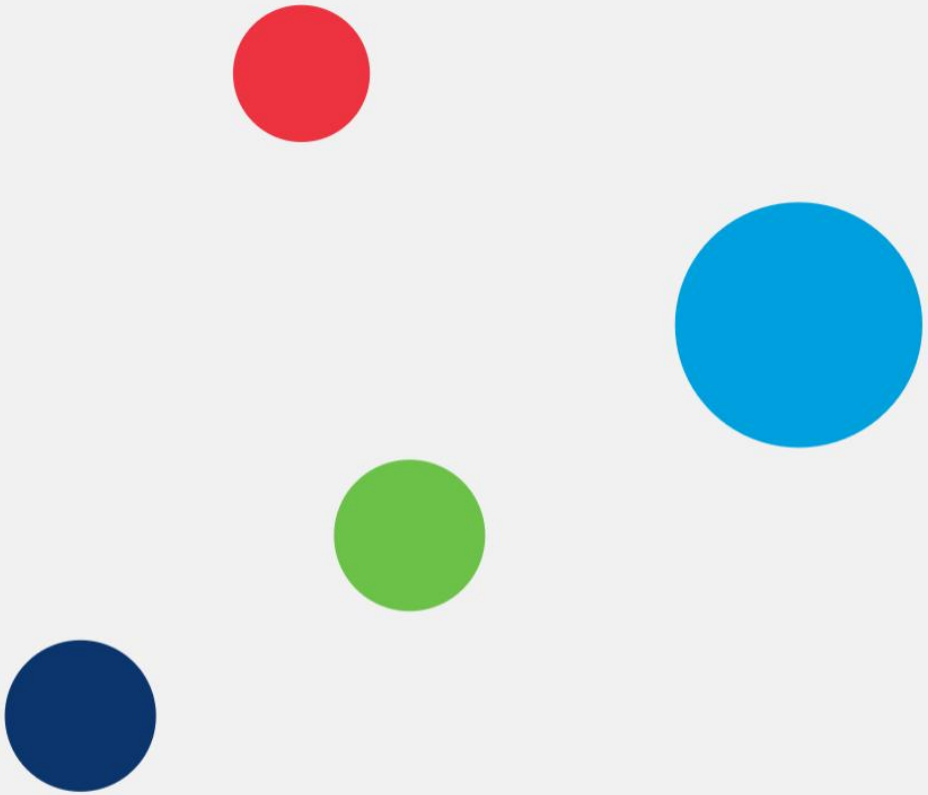
1 **Thread Synchronization**

Objectives

At the end of this module, you will be able to:

- Thread Synchronization

Thread Synchronization

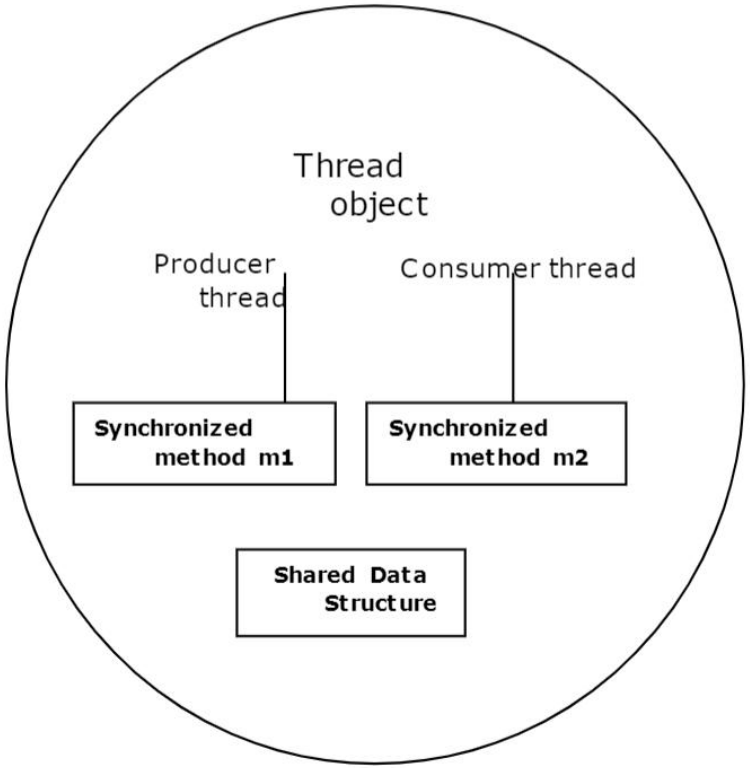


Synchronization

- It is normal for threads to be sharing objects and data
- Different threads shouldn't try to access and change the same data at the same time
- Threads must therefore be synchronized
- 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 (Contd.).

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



Synchronization (Contd.).

- Suppose there is a producer thread in the threaded object that is writing into a data structure within the thread object using a method say M1.
- While the producer thread is in method M1 and in the process writing into the data structure, care must be taken to ensure that while data is in the process of being written to the data structure, no other thread, say a consumer thread must be allowed to read the data through some other method (say M2) of the thread object at the same time while the writing of data is on.
- The consumer thread should wait till the producer thread has finished writing into the data structure, i.e., till the producer thread returns from method M1.
- The moment the producer thread returns from method M1, the consumer thread should be allowed to access the data structure through method M2.

Synchronization (Contd.).

- The argument is moving towards a mechanism by which you are trying to ensure that no two threads end up accessing a shared data structure at the same time that might lead to corrupting the data in the data structure, and might also lead to unpredictable results. This mechanism will in fact serialize access to shared data by multiple threads, i.e., each thread lines up behind a running thread till the running thread has finished with its operation.

Synchronization (Contd.).

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

Synchronization (Contd.).

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

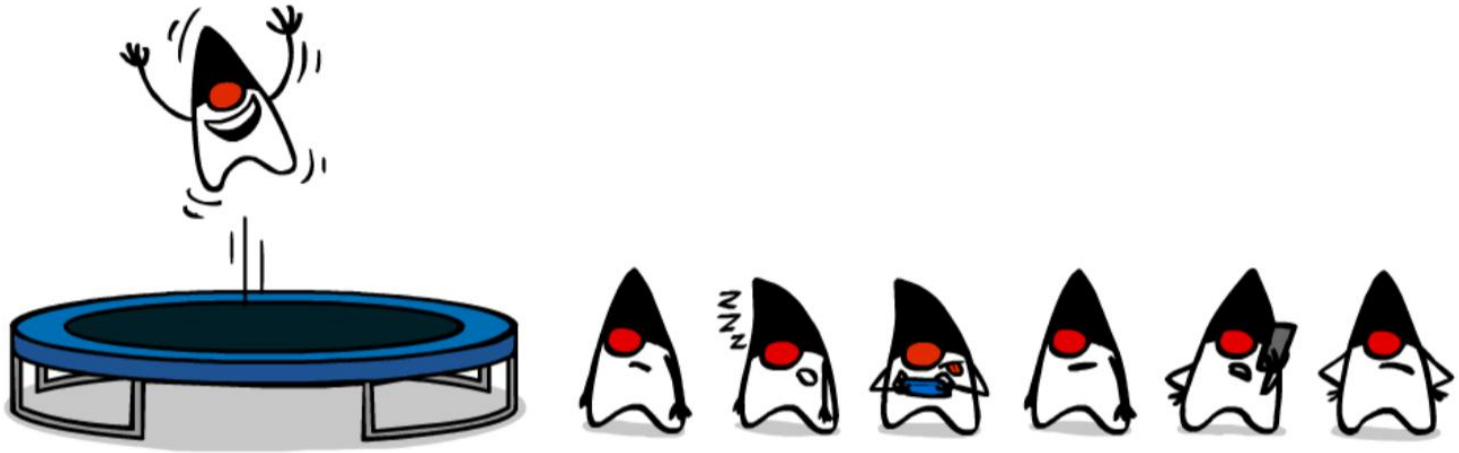
Synchronization (Contd.).

- Synchronized methods are an elegant variation on a time-tested model of interprocess-synchronization: the ***monitor***

- The *monitor* is a thread control mechanism
- A monitor is an object, which is used as a concurrency control mechanism. It is used as a mutually exclusive lock(mutex).
- 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

- Once a thread is inside a synchronized method, no other thread can call any other synchronized method on the same object.

Example



Here when one Duke is using the trampoline, the other Dukes are waiting for their turn.
As we discussed in the last slide, when the first Duke enters a monitor, the remaining Duke's are waiting for the first one to complete.

***Duke is Java's Mascot**

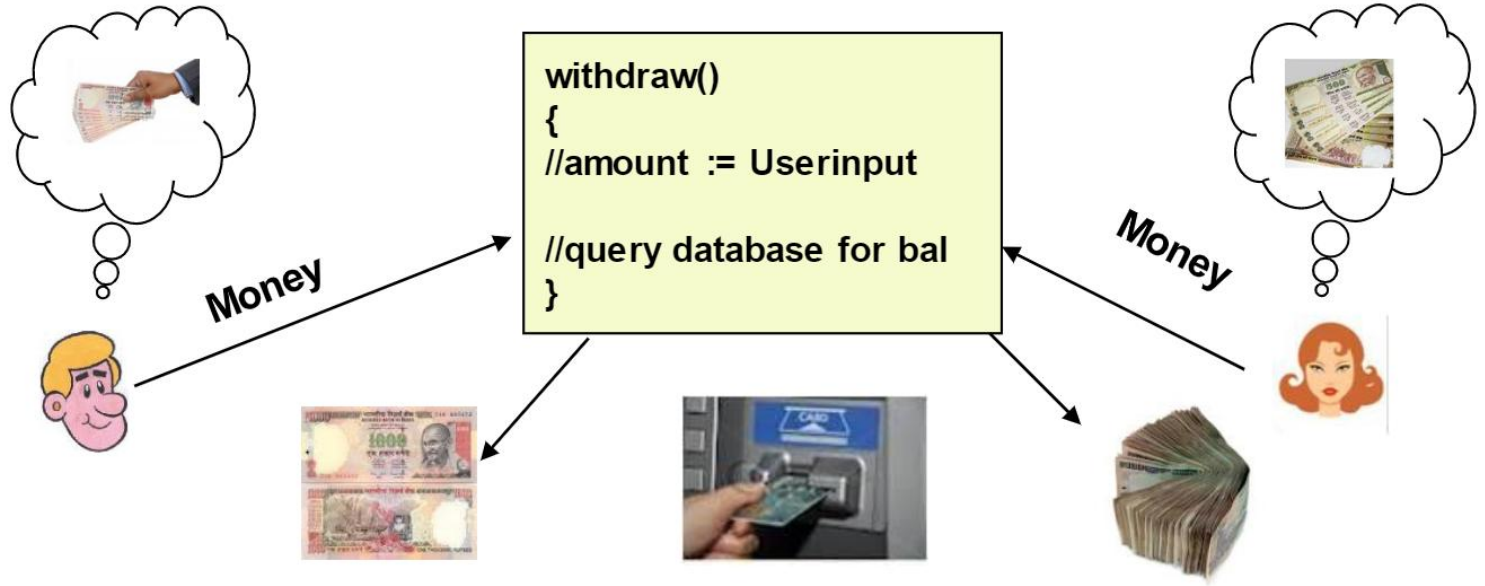
Synchronization (Contd.).

- Threads often need to share data.
- There is a need for a mechanism to ensure that the shared data will be used by only one thread at a time
- This mechanism is called synchronization.
- Key to synchronization is the concept of the monitor (also called a semaphore).

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 (Contd.).

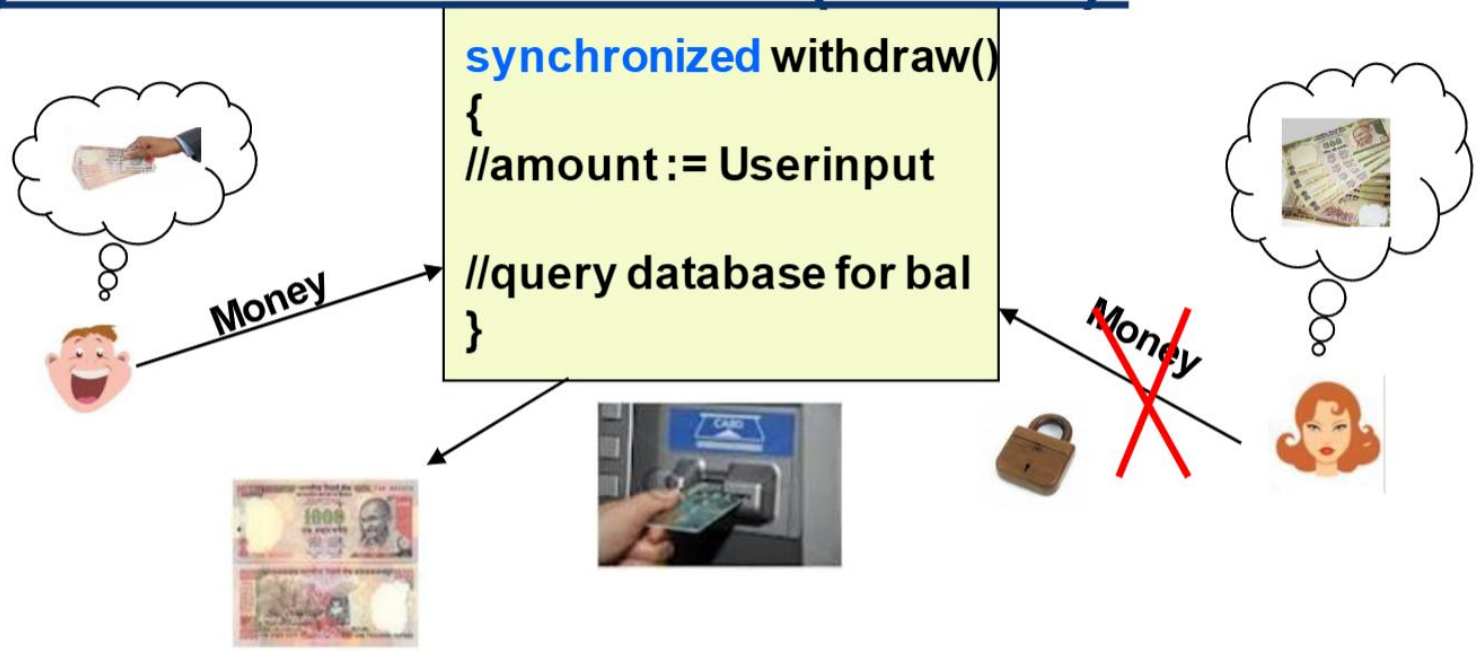


John and Mary are withdrawing money from their joint account. They are withdrawing at the same **instance** from different ATMs.

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 (Contd.).



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(){…}
    ```

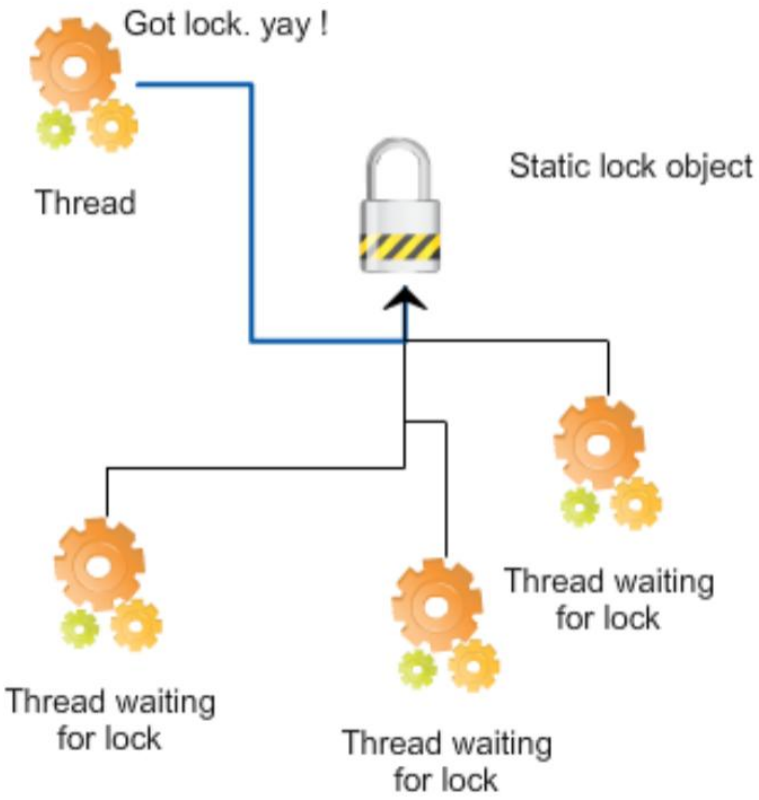
  - A block of code

    ```
    synchronized (objectReference){…}
    ```

- Synchronized methods in subclasses use same locks as their superclasses

Synchronization

- When one thread acquires the lock on the shared resource, the other threads need to wait for the lock to be released.



The Synchronized Statement

Syntax for block of code:

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


Implementing Synchronization-Example

```
class Account {
                int balance;
                public Account(){
                                balance=5000;
                }
            public synchronized void withdraw(int bal){
                                try{
                                                Thread.sleep(1000);
                                }
        catch(Exception ex){
            System.out.println("EXCEPTION OCCURED.."+ex);
        }
                balance= balance-bal;
                System.out.println("Balance remaining:::" + balance);
            }     }
```

Implementing Synchronization-Example

```
class C implements Runnable{
                    Account obj;
          public C(Account a){
          obj=a;
                    }
     public void run(){
                         obj.withdraw(500);
                    }                          }
class SynchEx{
               public static void main(String args[]){
                              Account a1=new Account();
                              C c1=new C(a1);
                              Thread t1=new Thread(c1);
                              Thread t2=new Thread(c1);
                              t1.start();
                              t2.start();
            }}
```

Assignment

Summary

- Thread Synchronization
- Synchronization methods

Thank You