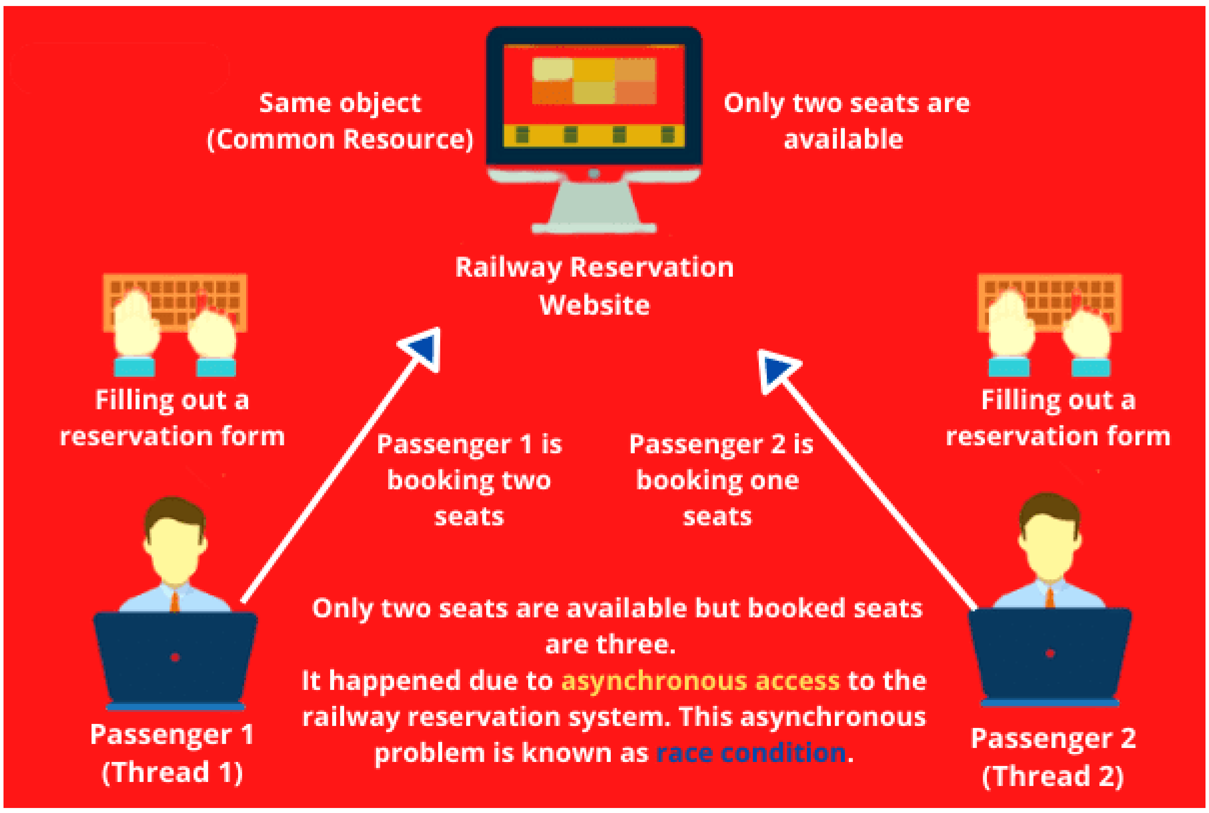
Thread Synchronization

multithreading increases the speed of execution of a program and optimizes computer resource usage.

Normally, multiple threads in a single program run asynchronously if threads do not share a common resource.

Under
some circumstances, it is possible that more than one thread access the
common resource (shared resource). That is, more than one thread access
the same object of a class.


Same object
(Common Resource)
Only two seats are available
Railway Reservation Website
Filling out a reservation form
Filling out a reservation form
Passenger 1 is booking two seats
Passenger 2 is booking one seats
Only two seats are available but booked seats are three.
It happened due to asynchronous access to the railway reservation system. This asynchronous problem is known as race condition.
Passenger 1 (Thread 1)
Passenger 2 (Thread 2)

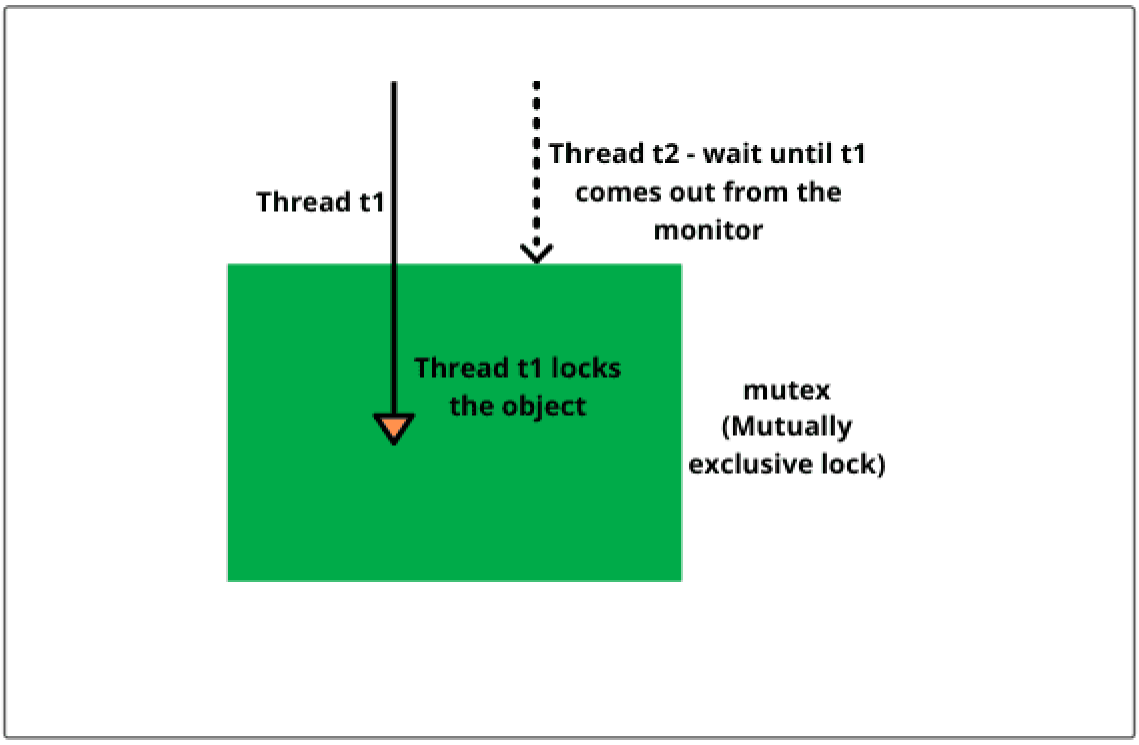
Object Lock in Java

The code in Java program can be synchronized with the help of a lock. A lock has two operations: acquire and release. The process of acquiring and releasing object lock (monitor) of an object is handled by Java runtime system (JVM).

In Java programming language, every object has a default object lock
that can be used to lock on a thread. This object lock is also known as

monitor that allows only one thread to use the shared resources (objects) at a time.

To acquire an object lock on a thread, we call a method or a block with the synchronized keyword. Before entering a synchronized method or a block, a thread acquires an object lock of the object.


Thread t1
Thread t2 - wait until t1 comes out from the monitor
Thread t1 locks the object
mutex (Mutually exclusive lock)

Rules for synchronizing shared resources in Java

In the Java program, there must mainly three rules to be followed when synchronizing share resources. The rules are as follows:

1.A thread must get an object lock associated with it before using a shared resource. If a thread has an object lock of the shared resource, Java runtime system (JVM) will not allow another thread to access the shared resource.

If a thread is trying to access the shared resource at the same time, it is blocked by JVM and has to wait until the resource is available.

2. Only methods or blocks can be synchronized. Variables or classes cannot be synchronized.

3. Only one lock is associated with each object or shared resource.

How can we achieve Synchronization in Java?

There are two
ways by which we can achieve or implement synchronization in Java. That
is, we can synchronize the object. They are:

- Synchronized Method
- Synchronized Block

When we declare a synchronized keyword in the header of a method, it is called **synchronized method**. Once a method is made synchronized, and thread calls the synchronized method, it gets locked on the method.

All other threads will have to wait for the current thread to release the lock. Using the synchronized keyword, we can synchronize the entire method.

Why use Synchronization?

The synchronization is mainly used to prevent thread interference and consistency problem.

Key Points to Remember

1. There are two types of synchronization: Process synchronization and Thread synchronization.

2. Thread synchronization in Java program is achieved through the monitor (lock) concept.

3. A monitor is an object that can block and resume threads.

4. Every object in Java programming language has an associated monitor.

5. Java programming language supports only two kinds of threads synchronization: Mutual exclusion synchronization and Conditional synchronization.

6. In mutual exclusion synchronization, only a single thread is allowed to have access to code at a time.

7. In conditional synchronization, multiple threads are allowed to work together to get results.

```
public class Table
{
  void printTable(int n) // Here, method is not synchronized.
  {
    for(int i = 1; i <= 5; i++)
    {
     System.out.println(n * i);
       try
       {
          Thread.sleep(400);
       }
      catch(InterruptedException ie)
      {
         System.out.println(ie);
       }
     }
  }
}
public class Thread1 extends Thread
{
  Table t; // Declaration of variable t of class type Table.

// Declare one parameterized constructor and pass variable t as a parameter.
     Thread1(Table t)
     {
        this.t = t;
     }
    public void run()
    {
      t.printTable(2);
    }
}
public class Thread2 extends Thread
{
  Table t;
  Thread2(Table t)
  {
    this.t = t;
  }
  public void run()
  {
    t.printTable(10);
  }
}
public class UnsynchronizedMethod {
public static void main(String[] args)
{
// Create an object of class Table.
    Table obj = new Table();
    Thread1 t1 = new Thread1(obj);
    Thread2 t2 = new Thread2(obj);
      t1.start();
      t2.start();
 }
}
```

```
public class Table
{
   synchronized void printTable(int n) // Here, method is synchronized.
   {
       for(int i = 1; i <= 5; i++)
       {
         System.out.println(n * i);
           try
           {
             Thread.sleep(400);
           }
          catch(InterruptedException ie)
          {
             System.out.println(ie);
          }
      }
  }
}
public class Thread1 extends Thread
{
  Table t; // Declaration of variable t of class type Table.

// Declare one parameterized constructor and pass variable t as a parameter.
   Thread1(Table t)
   {
      this.t = t;
   }
  public void run()
  {
     t.printTable(2);
  }
}
public class Thread2 extends Thread
{
  Table t;
  Thread2(Table t)
  {
    this.t = t;
  }
  public void run()
  {
    t.printTable(10);
  }
}
public class SynchronizedMethod {
public static void main(String[] args)
{
// Create an object of class Table.
  Table obj = new Table();
  Thread1 t1 = new Thread1(obj);
  Thread2 t2 = new Thread2(obj);
    t1.start();
    t2.start();
 }
}
```

Simultaneous execution of synchronized methods is possible for two different objects of the same class.

```
public class Table
{
  synchronized void printTable1(int x) // First synchronized method.
  {
    for(int i = 1; i <= 3; i++)
    {
      System.out.println(x * i);
      try
      {
        Thread.sleep(400);
      }
      catch(InterruptedException ie)
      {
        System.out.println(ie);
      }
   }
}
synchronized void printTable2(int y) // Second synchronized method.
{
  for(int j = 1; j <= 3; j++)
  {
    System.out.println(y * j);
    try
    {
      Thread.sleep(400);
    }
   catch(InterruptedException ie)
   {
      System.out.println(ie);
   }
  }
} }
public class Thread1 extends Thread
{
  Table t; // Declaration of variable t of class type Table.
  Thread1(Table t)
  {
     this.t = t;
  }
  public void run()
  {
     t.printTable1(2); // Calling first synchronized method.
   }
}
public class Thread2 extends Thread
{
 Table t;
 Thread2(Table t)
 {
   this.t = t;
 }
 public void run()
 {
   t.printTable2(10); // Calling second synchronized method.
 }
}
```

```
public class SynchronizedMethod {
public static void main(String[] args)
{
// Create two objects of class Table.
   Table obj = new Table();
   Table obj2 = new Table();

  Thread1 t1 = new Thread1(obj);
  Thread2 t2 = new Thread2(obj2);
    t1.start();
    t2.start();
 }
}
```

Can we synchronize static method in Java?

```
public class Table
{
synchronized static void printTable(int x) // Here, static method is synchronized.
{
for(int i = 1; i <= 3; i++)
{
 System.out.println(x * i);
 try
 {
   Thread.sleep(400);
 }
catch(InterruptedException ie)
{
   System.out.println(ie);
 }  }
}
}
public class Thread1 extends Thread
{
public void run()
{
   Table.printTable(2); // Calling synchronized static method using class name with pa
ssing argument 2.
 }
}
public class Thread2 extends Thread
{
public void run()
{
  Table.printTable(10); // Calling synchronized static method using class name with pa
ssing argument 10.
 }
}
public class SynchronizedMethod
{
public static void main(String[] args)
{
Thread1 t1 = new Thread1();
Thread2 t2 = new Thread2();
```

```
 t1.start();
 t2.start();
 }
}
```