SYNCHRONIZATION

• Synchronization in java is the capability *to control the access of multiple threads to any shared resource.*

• Java Synchronization is better option where we want to allow only one thread to access the shared resource.

• 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. Java provides unique, language level support for it.

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

• A monitor is an object that is used as a mutually exclusive lock, or mutex. Only one thread can own a monitor at a given time. When a thread acquires a lock, it is said to have entered the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor.

• These other threads are said to be waiting for the monitor. A thread that owns a monitor can reenter the same monitor if it so desires.

• Approaches:

○○ Using synchronized Method

○○ Using synchronized Statement

Using Synchronized Methods

- Synchronization is easy in Java, because all objects have their own implicit monitor associated with them.
- To enter an object's monitor, just call a method that has been modified with the synchronized keyword.
- While a thread is inside a synchronized method, all other threads that try to call it (or any other synchronized method) on the same instance have to wait.
- To exit the monitor and relinquish control of the object to the next waiting thread, the owner of the monitor simply returns from the synchronized method.

• To understand the need for synchronization, we will consider a simple example that does not use it—but should.

• The following program has three simple classes.

• The first one, Callme, has a single method named call( ). The call( ) method takes a String parameter called msg. This method tries to print the msg string inside of square brackets. After call( ) prints the opening bracket and the msg string, it calls Thread. sleep(1000), which pauses the current thread for one second.

• The constructor of the next class, Caller, takes a reference to an instance of the Callme class and a String, which are stored in target and msg, respectively. The constructor also creates a new thread that will call this object's run( ) method. The thread is started immediately. The run( ) method of Caller calls the call( ) method on the target instance of Callme, passing in the msg string.

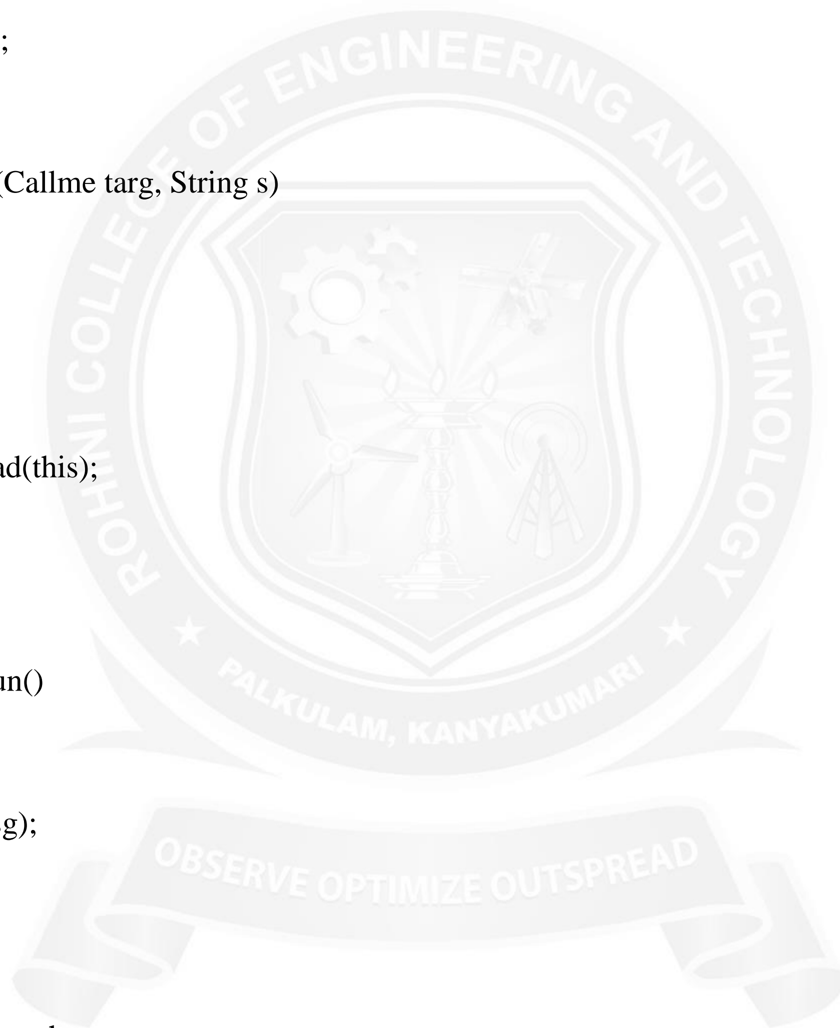
• Finally, the Synch class starts by creating a single instance of Callme, and three instances of Caller, each with a unique message string.

• The same instance of Callme is passed to each Caller.

// This program is not synchronized.

```
class Callme
{
void call(String msg)
{
System.out.print("[" + msg);
try
{
Thread.sleep(1000);
}
catch(InterruptedException e)
{
System.out.println("Interrupted");
}
```

```
System.out.println(“]”);
}
}
class Caller implements Runnable
{
String msg;
Callme target;
Thread t;
public Caller(Callme targ, String s)
{
target = targ;
msg = s;
t = new Thread(this);
t.start();
}
public void run()
{
target.call(msg);
}
}
public class Synch
{
public static void main(String args[])
{
Callme target = new Callme();
Caller ob1 = new Caller(target, “Hello”);
```



```
Caller ob2 = new Caller(target, “Synchronized”);
Caller ob3 = new Caller(target, “World”);
// wait for threads to end
try
{
ob1.t.join();
ob2.t.join();
ob3.t.join();
}
catch(InterruptedException e)
{
System.out.println(“Interrupted”);
}
}
}
```

***Sample Output*:**

```
Hello[Synchronized[World]
]
]
```

As we can see, by calling sleep( ), the call( ) method allows execution to switch to another thread. This results in the mixed-up output of the three message strings.

In this program, nothing exists to stop all three threads from calling the same method, on the same object, at the same time. This is known as a race condition, because the three threads are racing each other to complete the method.

This example used sleep( ) to make the effects repeatable and obvious. In most situations, a race condition is more subtle and less predictable, because we can’t be sure when

the context switch will occur. This can cause a program to run right one time and wrong the next. To fix the preceding program, we must serialize access to call(). That is, we must restrict its access to only one thread at a time. To do this, we simply need to precede call()'s definition with the keyword synchronized, as shown here:

This prevents other threads from entering call( ) while another thread is using it.

```
class Callme
{
synchronized void call(String msg)
{
...
```

Following is the sample java program after synchronized has been added to call( ):

```
class Callme
{
synchronized void call(String msg)
{
System.out.print("[" + msg);
try
{
Thread.sleep(1000);
}
catch(InterruptedException e)
{
System.out.println("Interrupted");
}
System.out.println("]");
}
```

```
}
class Caller implements Runnable
{
String msg;
Callme target;
Thread t;
public Caller(Callme targ, String s)
{
target = targ;
msg = s;
t = new Thread(this);
t.start();
}
public void run()
{
target.call(msg);
}
}
public class Synch
{
public static void main(String args[])
{
Callme target = new Callme();
Caller ob1 = new Caller(target, “Hello”);
Caller ob2 = new Caller(target, “Synchronized”);
Caller ob3 = new Caller(target, “World”);
```

```
// wait for threads to end
try
{
ob1.t.join();
ob2.t.join();
ob3.t.join();
}
catch(InterruptedException e)
{
System.out.println("Interrupted");
}
}
}
```

***Output*:**

```
[Hello]
[Synchronized]
[World]
```

Using synchronized Statement

While creating synchronized methods within classes that we create is an easy and effective
means of achieving synchronization, it will not work in all cases. We have to put calls to
the methods defined by the class inside a synchronized block.

Syntax:

```
synchronized(object)
{
```

```
// statements to be synchronized
}
```

Here, object is a reference to the object being synchronized. A synchronized block ensures that a call to a method that is a member of object occurs only after the current thread has successfully entered object's monitor.

Here is an alternative version of the preceding example, using a synchronized block within the run( ) method:

// This program uses a synchronized block.

```
class Callme
{
void call(String msg)
{
System.out.print("[" + msg);
try
{
Thread.sleep(1000);
}
catch (InterruptedException e)
{
System.out.println("Interrupted");
}
System.out.println("]");
}
}
class Caller implements Runnable
{
String msg;
```

```
Callme target;
Thread t;
public Caller(Callme targ, String s)
{
target = targ;
msg = s;
t = new Thread(this);
t.start();
}
// synchronize calls to call()
public void run()
{
synchronized(target)
{
// synchronized block
target.call(msg);
}
}
}
public class Synch1
{
public static void main(String args[])
{
Callme target = new Callme();
Caller ob1 = new Caller(target, “Hello”);
Caller ob2 = new Caller(target, “Synchronized”);
```

```
Caller ob3 = new Caller(target, "World");
// wait for threads to end
try
{
ob1.t.join();
ob2.t.join();
ob3.t.join();
}
catch(InterruptedException e)
{
System.out.println("Interrupted");
}
}
}
```

Here, the call( ) method is not modified by synchronized. Instead, the synchronized statement is used inside Caller's run( ) method. This causes the same correct output as the preceding example, because each thread waits for the prior one to finish before proceeding.

Sample Output:

```
[Hello]
[World]
[Synchronized]
```