4.4 synchronizing threads

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

When we want to share the resource with multiple threads, we can use the Java synchronization concept. So there is a need to synchronize the action of multiple threads and make sure that only one thread can access the resource at a given point in time. This is implemented using a concept called **monitors**. Each object in Java is associated with a monitor, which a thread can lock or unlock. Only one thread at a time may hold a lock on a monitor.

4.4.1 Purpose of using synchronization

1. To prevent thread interference
2. To prevent consistency problem

4.4.2 Types of Thread synchronization

There are two types of thread synchronization mutual exclusive and inter-thread communication.

1. Mutual Exclusive
    1. Synchronized method.
    2. Synchronized block.
    3. static synchronization.
2. Cooperation (Inter-thread communication in java)

4.4.3 Mutual Exclusive

Mutual Exclusive helps keep threads from interfering with one another while sharing data. This can be done by three ways in java:

1. by synchronized method
2. by synchronized block
3. by static synchronization

4.4.3.1 Synchronized Method

If you declare any method as synchronized, it is known as synchronized method. Synchronized method is used to lock an object for any shared resource. When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

Example Program: Without Synchronized Method class Table

```
{
void printTable(int n)
{//method not synchronized
for(int i=1;i<=5;i++) {
System.out.println(n*i);
try
{
Thread.sleep(400);
}
catch(Exception e){System.out.println(e);}
}
}
}
class MyThread1 extends Thread
{
Table t;
MyThread1(Table t)
{ this.t=t;
}
public void run()
{
t.printTable(5);
}
}

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

```
t.printTable(100);
}
}
class TestSynchronization1
{
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();
}
}
```

Output:

```
5
100
10
200
15
300
20
400
25
500
```

Example Program: With Synchronized Method class Table

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

```
System.out.println(e);
} }
} } class MyThread1 extends Thread {
Table t;
MyThread1(Table t)
{
this.t=t;
}
public void run()
{
t.printTable(5);
}
}
class MyThread2 extends Thread {
Table t;
MyThread2(Table t)
{ this.t=t;
}
public void run()
{
t.printTable(100);
}
}
public class TestSynchronization2
{
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();
}
}
```

Output:

```
5
10
15
```

```
20
25
100
200
300
400
500
```

4.4.3.2 Synchronized Block

Synchronized block can be used to perform synchronization on any specific resource of the method.Suppose you have 50 lines of code in your method, but you want to synchronize only 5 lines, you can use synchronized block.If you put all the codes of the method in the synchronized block, it will work same as the synchronized method.

Purpose of Synchronized Block

1. Synchronized block is used to lock an object for any shared resource.
2. Scope of synchronized block is smaller than the method.

Syntax:

```
synchronized (object reference expression)
{
//code block
}
```

Example program:

```
class Table
{
void printTable(int n)
{
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);
}
}
}
}//end of the method
}
class MyThread1 extends Thread
{
Table t;
MyThread1(Table t)
{ this.t=t;
}
public void run()
{
t.printTable(5);
}
}
class MyThread2 extends Thread
{
Table t;
MyThread2(Table t)
{
this.t=t;
}
public void run()
{
t.printTable(100);
}
}
public class TestSynchronizedBlock1
{
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();
}
}
```

Output:

```
5
10
15
20
25
100
200
300
400
500
```

4.4.3.3 Static Synchronization Block

If you make any static method as synchronized, the lock will be on the class not on object.

Example Program:

```
class Table
{
synchronized static void printTable(int n)
{
for(int i=1;i<=10;i++)
{
System.out.println(n*i);
try
{
Thread.sleep(400);
}
catch(Exception e){}
}
}
}
class MyThread1 extends Thread
{
public void run()
{
Table.printTable(1);
}
}
class MyThread2 extends Thread
{
public void run()
{
Table.printTable(10);
}
}
class MyThread3 extends Thread
{
public void run()
{
Table.printTable(100);
}
}
class MyThread4 extends Thread
{
public void run()
{
Table.printTable(1000);
}
}
public class TestSynchronization4
{
public static void main(String t[])
{
MyThread1 t1=new MyThread1();
MyThread2 t2=new MyThread2();
MyThread3 t3=new MyThread3();
MyThread4 t4=new MyThread4();
t1.start();
t2.start();
t3.start();
t4.start();
}
}
```

Output:

```
1
2
3
4
5
6
7
8
9
10
10
20
30
40
50
60
70
80
90
100
100
200
300
400
500
600
700
800
900 1000
1000
2000 3000
4000 5000
6000 7000
```

```
8000
9000
10000
```

4.5 Inter-thread Communication

Polling is a process in which the condition is repeatedly checked. If the condition is true, appropriate action is taken. This wastes CPU time.

For example, consider the classic queuing problem, where one thread is producing some data and another is consuming it. To make the problem more interesting, suppose that the producer has to wait until the consumer is finished before it generates more data. In a polling system, the consumer would waste many CPU cycles while it waited for the producer to produce. Once the producer was finished, it would start polling, wasting more CPU cycles waiting for the consumer to finish, and so on.

To avoid polling, Java includes an elegant interprocess communication mechanism via the **wait( )**, **notify( )**, and **notifyAll( )** methods. These methods are implemented as **final** methods in **Object**, so all classes have them. All three methods can be called only from within a **synchronized** context.

- **wait( )** tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls **notify( )**.
- **notify( )** wakes up a thread that called **wait( )** on the same object.
- **notifyAll( )** wakes up all the threads that called **wait( )** on the same object. One of the threads will be granted access.

The following methods are declared within Object:

```
final void wait( ) throws InterruptedException
final void notify( )
final void notify All( )
```

Example Program: class Q

```
{
```

```
int n;
synchronized int get()
{
System.out.println("Got: " + n);
return n;
}
synchronized void put(int n)
{
this.n = n;
System.out.println("Put: " + n);
}
}
class Producer implements Runnable
{
Q q;
Producer(Q q)
{
this.q = q;
new Thread(this, "Producer").start();
}
public void run()
{
int i = 0;
while(true)
{
q.put(i++);
}
}
}
class Consumer implements Runnable
{
Q q;
Consumer(Q q)
{
this.q = q;
new Thread(this, "Consumer").start();
}
public void run()
```

```
{
while(true)
{
q.get();
}
}
}
class PC
{
public static void main(String args[])
{
Q q = new Q();
new Producer(q);
new Consumer(q);
System.out.println("Press Control-C to stop.");
}
}
```

Output:

```
Put: 1
Got: 1
Got: 1
Got: 1
Got: 1
Got: 1
Put: 2
Put: 3
Put: 4
Put: 5
Put: 6
Put: 7
Got: 7
```

4.6 Daemon Threads

Daemon thread in java is a service provider thread that provides services to the user thread. Its life depend on the mercy of user threads i.e. when all the user threads dies, JVM terminates this thread automatically.

There are many java daemon threads running automatically e.g. gc, finalizer etc.
You can see all the detail by typing the jconsole in the command prompt. The jconsole tool provides information about the loaded classes, memory usage, running threads etc.

4.6.1 Purpose of Daemon thread:

- It provides services to user threads for background supporting tasks. It has no role in life than to serve user threads.
- Its life depends on user threads.
- It is a low priority thread.

4.6.2 Methods of Daemon Thread

Method	Description
public void setDaemon(boolean status)	is used to mark the current thread as daemon thread or user thread.
public boolean isDaemon()	is used to check that current is daemon.

Table 4.2 Methods of Daemon Thread

Example Program:

```
public class TestDaemonThread1 extends Thread
{
public void run()
{
if(Thread.currentThread().isDaemon())
{
//checking for daemon thread
System.out.println("daemon thread work");
}
else
{
System.out.println("user thread work");
}
}
public static void main(String[] args)
{
TestDaemonThread1 t1=new TestDaemonThread1();//creating th read
TestDaemonThread1 t2=new TestDaemonThread1();
TestDaemonThread1 t3=new TestDaemonThread1();
t1.setDaemon(true);//now t1 is daemon thread t1.start();//starting
threads
t2.start();
t3.start();
}
}
```

Output:

```
daemon thread work user thread work user thread work
```