Module: Core Java

Session 16: Ad. Concepts of Threading

* Thread Priorities and Thread Scheduling
* Synchronization
* Using Synchronized Methods
* The Synchronized Statement
* Inter Thread Communication
* Deadlock

**Objective:**

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

* Define Thread Priorities and Thread Scheduling
* Implement Synchronized methods and statements
* Gather information about inter thread communication
* Understand about Suspend, Resume, and Stop methods .
* Define and understand about Deadlock

# Thread Priorities and Thread Scheduling

# Every Java thread has a priority that helps the operating system to determine the order in which

# threads are scheduled. Java priorities are in the range between MIN\_PRIORITY (a constant

# with value of 1) and MAX\_PRIORITY (a constant with value of 10). Generally threads with

# higher priority are more important to a program and should be allocated processor time before

# lower-priority threads. However, thread priorities cannot guarantee the order in which threads

# execute. By default, every thread is given priority NORM\_PRIORITY (a constant of 5). Each

# new thread inherits the priority of the thread that created it.

Most Java platforms support time slicing, which enables threads of equal priority to share a processor. Without time slicing, each thread in a set of equal-priority threads runs to completion (unless it leaves the runnable state and enters the waiting or timed waiting state, or gets interrupted by a higher-priority thread) before other threads of equal priority get a chance to execute. Even if the thread has not finished executing at the time when the quantum expiration takes place, the processor is taken away from that thread and given to the next thread of equal priority (if one is available) with time slicing.

Which thread runs next will be determined by an operating system’s thread scheduler. The highest priority thread is kept running all the time by one of the simple implementations of the thread of a scheduler. But what if there are more than one highest-priority threads? In this case the thread scheduler ensures that such threads execute for a quantum each in a round-robin manner. The following figure illustrates a multi-level priority queue for threads. Assume that it is a single processor computer. Threads A and B execute for quantum in round-robin manners until both of them complete execution. This means that first A gets a quantum time to run, then B gets a quantum, then again A gets another quantum, then B and so on till one thread is completed. The entire power of the processor is then devoted to the remaining thread, unless another thread of that priority becomes ready. After that, the thread C runs to completion, assuming that no higher-priority threads arrive. D, E and F each execute for a quantum in round-robin fashion till all the executions are completed. Again you have to keep in mind the occurrence of higher-priority threads. Until all the threads run to completion, this process continues.

The following figure illustrates Thread-priority scheduling:

# 

# Fig. 5: Thread-priority scheduling

# Note: Thread scheduling is platform dependent—an application that uses multithreading could behave differently on separate Java implementations.

# The following example demonstrates how to control the execution of thread by setting its priority:

# Source:

class CounterThread extends Thread {

String nm;

public CounterThread(String nm) {

super();

this.nm = nm;

}

public void run() {

int count = 0;

while (true) {

try {

sleep(100);

} catch (InterruptedException e) {

}

if (count == 50)

count = 0;

System.out.println(nm+":" + count++);

}

}

}

public class PriorityDemo{

public static void main(String[] args) {

CounterThread t1 = new CounterThread("Thread1");

t1.setPriority(10);

CounterThread t2 = new CounterThread("Thread2");

t2.setPriority(1);

t2.start();

t1.start();

}

}

# In the above program the “Thread1” starts first even if “Thread2” is started first, because the

# priority of “Thread1” is higher than “Thread2”;

## Synchronizing Threads

# So far we have concentrated on cases where all the threads have run independent of each

# other. One thread did not need to know what another thread was doing. Sometimes, however,

# threads need to share data. In this case it is important to ensure that one thread is writing to a

# file while another thread is reading the file, it is likely that the thread that is reading the file will

# get corrupt data.

# The Java programming language supports the coding of programs that, though the use of

# concurrents, still exhibit deterministic behavior. They provide mechanisms for synchronizing the

# concurrent activity of threads. To synchronize threads, the Java Programming language uses

# monitors, which are high-level mechanisms for allowing only one thread at a time to execute a

# region of code protected by the monitor. The behavior of monitors is explained in terms of locks;

# there is a lock associated with each object.

# For example, suppose you have a thread that writes data to a file while and at the same time, another thread is reading data from that same file. When your threads need to share information you need to synchronize the threads to get the desired results.

# Infact, the entire Thread is not likely to be synchronized, but rather it is a method representing a segment of code, which must be synchronized to all Threads. Synchronization is supported in two types:

# Mutual Exclusion: Implemented with Object locks. Allowing multiple threads to run independently on shared data, with the ability to lock each other to prevent conflicts.

# Cooperation: Implemented with the wait() and notify() methods of class Object. This method of synchronization is meant to be used when all threads work towards a single result.

# It is important to understand synchronization locks. When a Thread wishes to run code which is synchronized, it must have the lock. A monitor (aka semaphore) is an object that provides a mutually exclusive lock (mutex). Java provides the synchronized keyword as the key that locks/unlocks an object. Having the lock essentially means that no other Thread may run the synchronized code. It can be used as a class or method modifier or as a statement (very localized). No long running method should not be synchronized, as it would become a traffic bottleneck. Once the current Thread finishes with the synchronized code, it releases the lock and allows another Thread to run the synchronized code.

# To guarantee that a variable is thread safe (i.e. not shared between threads) it can be marked as volatile.

# This effectively allows code to run in parallel as long as they do not cause a conflict on another Thread. The code placed in the synchronize method is that which may cause such a conflict.

# Implementing the synchronized code is simple. It is a special block of code within a method:

# public void aMethod() { synchronized(variable)     {                      }     }

# Here variable is an Object which must not be modified by any other synchronized Thread.

# public synchronized void anotherMethod()     {              }

# As with Threads themselves, synchronization has it’s downfalls due to the potential for deadlock. Java has implemented ways to avoid this, by allowing a synchronized method to call another, even if they both require a lock on the same Object.

**Source Code:**

public class SecondThread{

public SecondThread()

{

FirstThread t1 = new FirstThread();

FirstThread t2= new FirstThread();

t1.start();

t2.start();

}

private class FirstThread extends Thread

{

public void run()

{

myMethod();

}

}

public synchronized void myMethod()

{

{

System.out.println("Entering Synchronized Code");

try

{

Thread.sleep(5000);

}catch(Exception e){ }

}

}

public static void main(String[] args)

{

new SecondThread();

}

}

# Note that there is a 5 second wait between displaying the first line and the second. Yet if the myMethod() is made non-synchronized, they both appear at once This is happening since there is no restriction on the threads in parallel.

# When objects are waiting to execute, due to a lock and synchronized code associated to it, the state of the object can be either of the following-i.e. entering the monitor, or entering the wait queue. Both terms are common becasu once the Thread is in waiting, it has become a monitor that is waiting for signals, and also waiting for the current locked Thread to finish.

# If the current Thread requires an operation by another Thread to complete before continuing, you can reorder the wait queue, by calling wait(long timeout) on the running Thread.

# The other option is to use a constant wait(), which waits until the notify() or notifyAll() method is called to release the wait on a specific Object, it can be any Object - a Java supplied one, or your own.

# Consider this Example having Synchronization problem

**Source Code:**

//demonstrates synchronization problems with threads

class Counter{

int i = 0;

public void count(){

while (i<30){

System.out.println(i);

i++;

}

}

}

public class CounterThread extends Thread {

Counter c;

public static void main(String args[]){

Counter d = new Counter();

CounterThread ct1 = new CounterThread (d);

CounterThread ct2 = new CounterThread(d);

ct1.start();

ct2.start();

}

public CounterThread(Counter d){

this.c = d;

}

public void run() {

c.count();

}

}

Since there is no guarantee when a particular thread will be interrupted by the scheduler you cannot guarantee that one will not be interrupted in between the statements System.out.println(i) and i++. If this happens it is possible for the same value to be printed out twice. Other problems can be associated with the thread being interrupted between reading a variable's value and writing a new value back.

Consider this example solving Synchronization problem

Java solves this problem by implementing the monitor approach to concurrency. A monitor is a special purpose object, which applies the principle of mutual exclusion to groups of procedures. Monitors in Java enforce mutually exclusive access to synchronized methods. When a monitor automatically checks to see whether any other thread is currently executing a synchronized method on that object. If not then the current thread is allowed to enter the monitor, the current thread must wait until the other thread leaves the monitor.

To demonstrate how monitors operate, we will rewrite the Counter example from above to take advantage of monitors:

**Source Code**

// demonstrates synchronization with threads

class SyncCounter {

int i = 0;

public synchronized void count(){

while (i<30){

System.out.println(i);

i++;

}

}

public synchronized void count(){

while (i <30){

System.out.println(i);

i++;

}

}

}

public class SyncThread extends Thread{

SyncCounter c;

public static void main(String args[]){

SyncCounter d = new SyncCounter();

SyncThread ct1 = new SyncThread(d);

SyncThread ct2 = new SyncThread(d);

ct1.start();

ct2.start();

}

public SyncThread(SyncCounter d){

this.c = d;

}

public void run() {

C.count ();

}

}

**The Synchronized Statement**

Suppose, you want to synchronize access to objects of a class that was not designed for multithreaded access. That is, the class does not use synchronized methods. Further, these are third-party classes, and you do not have access to the source code. Thus the question of adding the keyword synchronized to the appropriate methods in such classes does not arise. How can the access to an object of this class be synchronized? The answer is to put the calls to methods that you want to synchronize inside a synchronized block. The general form of the synchronized method is as follows:

synchronized(object)

{

// statements to be synchronized

}

Here, object is a reference to the object being synchronized, i.e., the object whose methods you need to synchronize. If you want to synchronize only a single statement, then the curly braces are not required. 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 the object's monitor.

When you use the synchronized modifier to specify that a method is synchronized, you are locking on the particular object whose method is invoked. You can synchronize at a lower level than a method.

For example:

public void count(){

synchronized(this){

int i = this.i;

}

while (i ! = 100) {

System.out.println(i);

i++;

}

synchronized (this){

this.i = i;

}

}

# Interthread communication using wait(), notify() and notifyAll()

# The following example explains you about the Java thread synchronization through a simple producer/consumer example. For example, imagine a Java application where one thread (the producer) writes data to a file while a second thread (the consumer) reads data from the same file. Or, as you type characters on the keyboard, the producer thread places key events in an event queue and the consumer thread reads the events from the same queue. Both of these examples use concurrent threads that share a common resource: the first shares a file, the second shares an event queue. Because the threads share a common resource, they must be synchronized in some way.

A very simple example of wait() and notify() is described in the following three classes.

The first class is named PingPong and consists of a single synchronized method and a state variable. The method is hit() and the only parameter it takes is the name of the player who will go next.

The algorithm is essentially this:

If it is my turn,

note whose turn it is next,

then PING,

and then notify anyone waiting.

otherwise,

wait to be notified.

**Source Code:**

1 public class PingPong {

2 // state variable identifying whose turn it is.

3 private String whoseTurn = null;

4

5 public synchronized boolean hit(String opponent) {

6

7 String x = Thread.currentThread().getName();

8

9 if (whoseTurn == null) {

10 whoseTurn = x;

11 return true;

12 }

13

14 if (x.compareTo(whoseTurn) == 0) {

15 System.out.println("PING! ("+x+")");

16 whoseTurn = opponent;

17 notifyAll();

18 } else {

19 try {

20 long t1 = System.currentTimeMillis();

21 wait(2500);

22 if ((System.currentTimeMillis() - t1) > 2500) {

23 System.out.println("\*\*\*\*\*\* TIMEOUT! "+x+

24 " is waiting for "+whoseTurn+" to play.");

25 }

26 } catch (InterruptedException e) { }

27 }

28 return true; // keep playing.

29 }

30

}

In line 3 we declare our state variable, whoseTurn. This is declared private since the users of the class do not need to know it. Line 5 declares our method and it must have the synchronized keyword or the call to wait() will fail.

In line 7 we get our own name from the thread object. As you will see later, we set this after the thread is created. This helps in debugging since our thread is named something useful and is a convenient way to identify the players.

Lines 9 through 12 solve the problem of whose turn it is before anyone has gone. The policy implemented is that the first thread to invoke this method will get the honor of going first.

Lines 14 through 17 execute when it is the current thread's turn to go. When executed, the thread updates the state variable with the next thread's turn. This is done before the notify, since the notify may cause another thread to start running immediately before it knows its turn to run. Then notifyAll() is called to notify all threads that are waiting on this object, the one that they can run. If you are using only two threads, simply call notify() since that call will awaken exactly one thread from the set waiting to run. With two threads, only one thread can be waiting, so the correct thread will wake up. If you extend this to three or more threads, however, the notify call may not wake up the correct thread and the system will stop until that thread's wait times out.

Lines 19 through 26 execute when it is not the current thread's turn to go. Line 21 simply calls wait() and goes to sleep. However, you will notice that in line 20 the code notes the current time. It does this because when execution continues after the wait call returns. The reason for continuing could be either that the wait timed out or that our thread was awakened with a call to notify(). The only way to tell the difference is to measure how long the thread was asleep.

This timeout test is performed in line 22. If a timeout occurs, an informative message is printed to the console. In practice this will happen only when the time spent in lines 14 through 17 is greater than 2.5 seconds.

Line 26 is where we catch the InterruptedException. This would be thrown if the thread in the wait() call stops prematurely.

Really, that is all there is to in this part of the code. However, add some additional code (shown below) between lines 8 and 9 to allow a third thread to cause the threads using this class to exit.

8.01 if (whoseTurn.compareTo("DONE") == 0)

8.02 return false;

8.03

8.04 if (opponent.compareTo("DONE") == 0) {

8.05 whoseTurn = opponent;

8.06 notifyAll();

8.07 return false;

* 1. }

As you can see, this is done by setting the special opponent DONE in the call to hit(). When the opponent is done, line 8.02 makes sure the code returns the boolean false. Once we have the class of type PingPong, any thread with a reference to an instance of class PingPong can synchronize itself with other threads holding that same reference. To illustrate this, consider the following Player class designed for use in the instantiation of a couple of threads:

1 public class Player implements Runnable {

2 PingPong myTable; // Table where they play

3 String myOpponent;

4

5 public Player(String opponent, PingPong table) {

6 myTable = table;

7 myOpponent = opponent;

8 }

9

10 public void run() {

11 while (myTable.hit(myOpponent))

12 ;

13 }

14 }

As you can see, this code is even simpler. All we really need is a class that implements the Runnable interface. The Thread class provides a constructor that takes a reference to an object implementing Runnable.

The two instance variables in this class are the reference holding the PingPong object and the name of this player's opponent. This latter field is used in the hit() method to tell the object which player should go next.

There is a single constructor taking a PingPong object and the name of an opponent. To satisfy the Runnable interface, there is the method run in lines 10 through to 13.

The run method runs an infinite loop, calling hit() until it returns false. This method returns true until some thread calls it with the opponent name DONE.

To complete our example, we have an application class that will create a couple of threads using the Player class and pit them against each other. This is shown below in the Game class:

1 public class Game {

2

3 public static void main(String args[]) {

4 PingPong table = new PingPong();

5 Thread alice = new Thread(new Player("bob", table));

6 Thread bob = new Thread(new Player("alice", table));

7

8 alice.setName("alice");

9 bob.setName("bob");

10 alice.start(); // alice starts playing

11 bob.start(); // bob starts playing

12 try {

13 // Wait 5 seconds

14 Thread.currentThread().sleep(5000);

15 } catch (InterruptedException e) { }

16

17 table.hit("DONE"); // cause the players to quit their

//threads.

18 try {

19 Thread.currentThread().sleep(100);

20 } catch (InterruptedException e) { }

21 }

22 }

Because we want to execute this class from the command line, it must include a public static method named main that takes a single argument which is an array of strings. This is the method signature that the java command keys of when instantiating a class from the command line.

Line 4 is where the code instantiates a copy of our PingPong class and stores the reference in the local variable table. Line 5 and line 6 are compound object creations, first creating new Player objects and then using those objects in the creation of new Thread objects. At the time when it is created, the name of the opponent is specified. So Alice's opponent is Bob and Bob's opponent is Alice. These new threads are named using the setName method in lines 8 and 9, and then they are started in lines 10 and 11.

After line 11 is executed, there are three user threads running, one named alice, one named bob, and the main thread. On the system console you will start seeing messages of the form:

PING! (alice)

PING! (bob)

PING! (alice)

...

and so on. The threads alternate which one runs by the state in the PingPong object. This object forces them to run one after another, however it also ensures that they run as rapidly after one another as possible since as soon as one is finished, it calls notifyAll() and the other thread begins to run.

Finally, in lines 12 through to 15, you will see that the main thread goes to sleep for five seconds or so, and when it wakes up, it calls hit() with the magic bullet name DONE. This will cause the alice and bob threads to exit. The short sleep in lines 18 through 20 cover this case and allow our program to exit normally on all systems.

So we have managed to get two threads to share the processor equally by synchronizing the use of a common object instance.

### Fairness, Starvation, and Deadlock

You can call a system fair if each thread gets enough access to limited resources to make reasonable progress. You need to take precaution in order to ensure fairness if you write a program in which several concurrent threads are competing for fairness. Deadlocks and starvations are not allowed in a fair system. When does starvation take place? When one or more threads in the program are blocked from gaining access to a resource and hence, cannot progress. The ultimate form a starvation is a deadlock. Deadlock happens when two or more threads in the program are on a condition, which cannot be satisfied.

### Suspend() and resume() and stop()

Thread suspend and resume are two more thread management features. When a thread executes a thread suspend to suspend the execution of itself or another thread, the indicated thread will be suspended ***until*** the execution of a thread resume that releases the indicated thread. For example, suppose we have three threads ***A***, ***B*** and ***C*** running concurrently. Then, thread ***A*** execute a thread suspend to suspend the execution of thread ***B***. After this, we have only two threads ***A*** and ***C*** running concurrently. Note that even though both ***A*** and ***C*** are waiting for the completion of their own I/O activities and no thread is running, the suspended thread ***B*** cannot run. To run thread ***B*** again, one of the other threads must execute a corresponding thread resume. For example, thread ***C*** may execute a thread resume to resume thread ***B***'s execution. After this, all three threads are running concurrently.

Note that Sun has removed Thread.stop(), Thread.suspend(), and Thread.resume(). These methods are dangerous to use and you do not require them as it is inherently unsafe. Invoking stop() methods on a thread causes it to unlock all the monitors that it had locked. (The monitors are unlocked as the ThreadDeath exception propagates up the stack.) If any of the objects previously protected by these monitors were in an inconsistent state, other threads may now view these objects in an inconsistent state. Such objects are said to be damaged. When threads operate on damaged objects, arbitrary behavior can result. This behavior may be subtle and difficult to detect, or it may be pronounced. Unlike other unchecked exceptions, ThreadDeath kills threads silently; thus, the user has no warning that his program may be corrupted. The corruption can manifest itself at any time after the actual damage occurs, even hours or days in the future. In a similar fashion suspend() and resume() methods may lead to deadlock situations.

**Suspending, Resuming, and Stopping Threads - Java 1.1 and Earlier Versions**

// Using suspend() and resume().

class NewThread implements Runnable {

String name; // name of thread

Thread t;

NewThread(String threadname) {

name = threadname;

t = new Thread(this, name);

System.out.println("New thread: " + t);

t.start(); // Start the thread

}

// This is the entry point for thread.

public void run() {

try {

for(int i = 15; i > 0; i--) {

System.out.println(name + ": " + i);

Thread.sleep(200);

}

} catch (InterruptedException e) {

System.out.println(name + " interrupted.");

}

System.out.println(name + " exiting.");

}

}

class SuspendResume {

public static void main(String args[]) {

NewThread ob1 = new NewThread("One");

NewThread ob2 = new NewThread("Two");

try {

Thread.sleep(1000);

ob1.t.suspend();

System.out.println("Suspending thread One");

Thread.sleep(1000);

ob1.t.resume();

System.out.println("Resuming thread One");

ob2.t.suspend();

System.out.println("Suspending thread Two");

Thread.sleep(1000);

ob2.t.resume();

System.out.println("Resuming thread Two");

} catch (InterruptedException e) {

System.out.println("Main thread Interrupted");

}

// wait for threads to finish

try {

System.out.println("Waiting for threads to finish.");

ob1.t.join();

ob2.t.join();

} catch (InterruptedException e) {

System.out.println("Main thread Interrupted");

}

System.out.println("Main thread exiting.");

}

}

**Output:**

New thread: Thread[One,5,main]

One: 15

New thread: Thread[Two,5,main]

Two: 15

One: 14

Two: 14

One: 13

Two: 13

One: 12

Two: 12

One: 11

Two: 11

Suspending thread One

Two: 10

Two: 9

Two: 8

Two: 7

Two: 6

Resuming thread One

Suspending thread Two

One: 10

One: 9

One: 8

One: 7

One: 6

Resuming thread Two

Waiting for threads to finish.

Two: 5

One: 5

Two: 4

One: 4

Two: 3

One: 3

Two: 2

One: 2

Two: 1

One: 1

Two exiting.

One exiting.

Main thread exiting.

Using Java 2

// Suspending and resuming a thread for Java 2

class NewThread implements Runnable {

String name; // name of thread

Thread t;

boolean suspendFlag;

NewThread(String threadname) {

name = threadname;

t = new Thread(this, name);

System.out.println("New thread: " + t);

suspendFlag = false;

t.start(); // Start the thread

}

// This is the entry point for thread.

public void run() {

try {

for(int i = 15; i > 0; i--) {

System.out.println(name + ": " + i);

Thread.sleep(200);

synchronized(this) {

while(suspendFlag) {

wait();

}

}

}

} catch (InterruptedException e) {

System.out.println(name + " interrupted.");

}

System.out.println(name + " exiting.");

}

void mysuspend() {

suspendFlag = true;

}

synchronized void myresume() {

suspendFlag = false;

notify();

}

}

class SuspendResume {

public static void main(String args[]) {

NewThread ob1 = new NewThread("One");

NewThread ob2 = new NewThread("Two");

try {

Thread.sleep(1000);

ob1.mysuspend();

System.out.println("Suspending thread One");

Thread.sleep(1000);

ob1.myresume();

System.out.println("Resuming thread One");

ob2.mysuspend();

System.out.println("Suspending thread Two");

Thread.sleep(1000);

ob2.myresume();

System.out.println("Resuming thread Two");

} catch (InterruptedException e) {

System.out.println("Main thread Interrupted");

}

// wait for threads to finish

try {

System.out.println("Waiting for threads to finish.");

ob1.t.join();

ob2.t.join();

} catch (InterruptedException e) {

System.out.println("Main thread Interrupted");

}

System.out.println("Main thread exiting.");

}

}

System.out.println("Resuming thread One");

ob2.mysuspend();

System.out.println("Suspending thread Two");

Thread.sleep(1000);

ob2.myresume();

System.out.println("Resuming thread Two");

} catch (InterruptedException e) {

System.out.println("Main thread Interrupted");

}

// wait for threads to finish

try {

System.out.println("Waiting for threads to finish.");

ob1.t.join();

ob2.t.join();

} catch (InterruptedException e) {

System.out.println("Main thread Interrupted");

}

System.out.println("Main thread exiting.");

}

}

**Thread Deadlock**

In multiple threading, following problems may occur.

Deadlock or deadly embrace occurs when two or more threads are trying to gain control of the same object, and each one has a lock on another resource that they need in order to proceed.

For example, When thread A waiting for lock on Object P while holding the lock on Object Q and at the same time, thread B holding a lock on Object P and waiting for lock on Object Q, deadlock occurs.

* Please note that if the thread is holding a lock and went to a sleeping state, it does not loose the lock. However, when thread goes in blocked state, it normally releases the lock. This eliminates the potential of deadlocking threads.
* Java does not provide any mechanisms for detection or control of deadlock situations, so the programmer is responsible for avoiding them.

In detail A deadlock occurs when two threads have a circular dependency on a pair of synchronized objects. In other words deadlock when two or more threads are waiting for a mutual condition that can never be satisfied; they are starving each other.

To understand a deadlock situation, imagine the case of a round-table dinner of five friends. Five friends once sat around a table with a bowl of Chinese noodles in front of each of them and a chopstick between each of them. The problem is that each of the friends needs two chopsticks, one each for left and right hands to take a chunk of noodles. Also remember that they are allowed to pick up the left chopstick first and then the right one. Chopsticks are limited shared resources and the friends are the threads here. Each of them grabs one chopstick from the left, resulting in all of them having a single chopstick. Now if they look for the right hand chopstick what will happen? They will find it is already being held by the friend sitting to his right. So the wait for the chopstick will be an indefinite one. If one of them starts to eat, then others will have to wait for their turn. This approach does help, but the potential for starvation and therefore, the possibility for deadlock is still there.

You have two approaches to solve a deadlock in a situation like this: prevention or detection. Prevention means designing the system in a way makes deadlock impossible. Detection, on the other hand, means allowing for deadlock but detecting it and dealing with its consequences when they arise. Prevention is better than cure, particulrly because trying to detect deadlocks can often be a daunting task in itself.

Getting back to the five friends, the root of the problem is the fact that there is no order imposed on the selection of chopsticks. By assigning a priority order to the chopsticks, you can easily solve the deadlock problem: just assign increasing numbers to the chopsticks. Then force the friends to always pick up the chopstick with the lower number first. This results in the friend sitting between chopsticks 1 and 2 and the friend sitting between chopsticks 1 and 5 going for chopstick 1. Whoever gets it first is then able to get the remaining chopsticks, while the other philosopher is left waiting. When the lucky friend with two chopsticks finishes his bite and returns the chopsticks, the process repeats itself, allowing all the philosophers to eat. Deadlock has been successfully avoided.

**Summary**

Threads can contend with each other for a shared resource

* Looping is eliminated in multi-threading.
* A thread priority, defined in terms of integers, decides when to switch from one running thread to the next.
* Synchronization of threads is essential.
* Java supports inter-threaded messaging and communications.

To coordinate activity between different threads, we use the wait(), notify(), and notifyAll() methods.

We have learnt to create Multiple Threads. Multiple threads may give problems like Deadlocks