UNIT 3: UTILITIES & MULTITHREADING

Unit Structure

3.0 Learning Objectives

After learning this unit, you will be able to:

- Identify utility package
- Describe arrays and hash table
- Explain the thread lifecycle
- Define thread class and runnable interface
- State thread priorities
- Point out synchronisation
- Illustrate deadlock
- Explain suspending, resuming and stopping threads

3.1 Introduction

Java Utility package is one of the most commonly used packages in the java program. The Utility Package of Java consists of the following components:

- Collections framework
- Legacy collection classes
- Event model
- Date and time facilities
- Internationalisation

Miscellaneous utility classes such as string tokeniser, random-number generator and bit array

In this unit, we will be studying about details of these utility packages as well as about multithreading.

3.2 Comparing Arrays: Java Util

This section shows you how to determine the given arrays are same or not. The given program illustrates you how to compare arrays according to the content of that in this section, you can see that the given program initializes two arrays and input five numbers from user through the keyboard. And then program checks whether the given taken both arrays are same or not. This comparison operation is performed by using the equals () method of Arrays class.

public class ComparingArrays{
public Arrays.equals():
Above method compares two arrays.

Arrays is the class of the *java.util.*;* package. This class and it's methods are used for manipulating arrays.

Here is the code of the program:

```
import java.io.*;
import java.util.*;


classArrayDemo{
static void main(String[] args) throws IOException{
   int[] array1 = new int[5];
   int[] array2 = new int[5];
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
 try{
System.out.println("Enter 5 numbers for the first Array : ")
for(int i = 0; i < array1.length; i++){
array1[i] = Integer.parseInt(br.readLine());
}
System.out.println("Enter 5 numbers for the second Array : ")
for(int i = 0; i < array2.length; i++){
array2[i] = Integer.parseInt(br.readLine());
}
}
catch(NumberFormatException ne){
ne.printStackTrace();
}
```

```
boolean check = Arrays.equals(array1, array2);
if(check == false)
System.out.println("Arrays are not same.")
else
System.out.println("Both Arrays are same").
}
}
```

Check your progress 1

1. Write a note on Arrays.equals().
2. Explain arrays class.

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

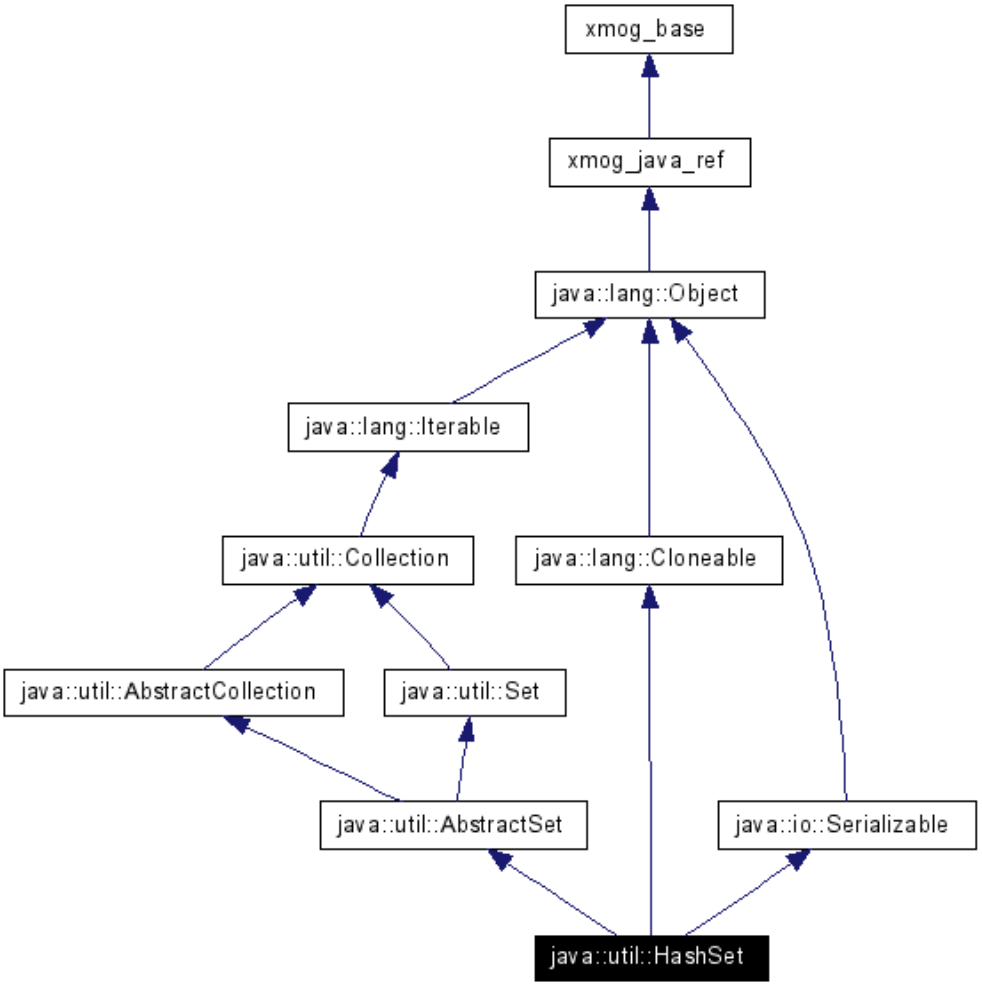
.....

.....

.....

.....

3.3 Creating a Hash Table: Java Util



This section explains the implementation of the hash table. “What is the hash table and how to create that? Hash Table holds the records according to the unique key value. It stores the non-contiguous key for several values. Hash Table is created using an algorithm (hashing function) to store the key and value regarding to the key in the hash bucket. If you want to store the value for the new key and if that key is already exists in the hash bucket then the situation known as collision” occurs which is the problem in the hash table i.e. maintained by the hashing function. The drawback is that hash tables require a little bit more memory and that you cannot use the normal list procedures for working with them.

This program simply asks you for the number of entries which have to enter into the hash table and takes one-by-one key and its value as input. It shows all the elements with the separate key.

Code Description:

Hashtable<Integer, String> hashTable = new Hashtable<Integer, String>():

Above code creates the instance of the **Hashtable** class. This code is using the type checking of the elements which will be held by the hash table.

hashTable.put(key, in.readLine()):

Above method puts the values in the hash table regarding to the unique key. This method takes two arguments in which, one is the key and another one is the value for the separate key.

Map<Integer, String> map = new TreeMap<Integer, String>(hashTable):

Above code creates an instance of the TreeMap for the hash table which name is passed through the constructor of the TreeMap class. This code creates the Map with the help of the instance of the TreeMap class. This map has been created in the program for showing several values and it's separate key present in the hash table. This code has used the type checking.

Check your progress 2

1. What is the drawback for hash tables?
2. Which method puts the values in hash table?

.....

.....

.....

.....

.....

.....

.....

3.4 Multi-threading

Java provides built-in support for multithreaded programming. A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread and each thread defines a separate path of execution.

A multithreading is a specialised form of multitasking. Multitasking threads require less overhead than multitasking processes.

There is another term to be defined related to threads: process: A process consists of the memory space allocated by the operating system that can contain one or more threads. A thread cannot exist on its own; it must be a part of a process. A process remains running until all of the non-daemon threads are done executing.

Multithreading enables you to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum.

Check your progress 3

1. What do you mean by thread?
2. Explain a process.

.....

.....

.....

.....

.....

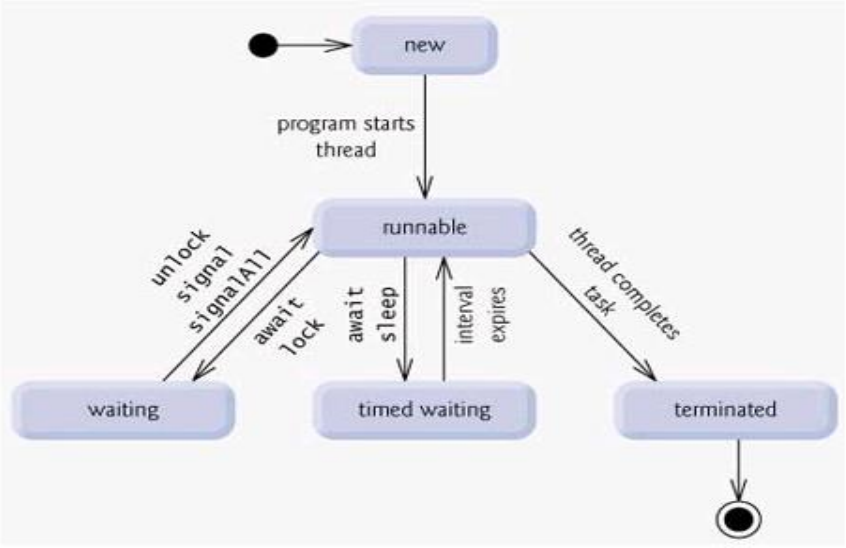
.....

.....

.....

3.5 Thread Life Cycle

The thread passes many stages in its life cycle including being born, starting, running and dying. Following diagram shows complete life cycle of a thread.



Life Cycle of Thread

Above mentioned stages are explained here:

- **New -** At this stage the thread is also referred to as a born thread and it is where the life cycle of the thread begins. A thread shall remain in this state until the program starts it.
- **Runnable -** at this stage a thread is executing its task. After it is started a thread becomes runnable.
- **Waiting -** While waiting for another thread to perform its task, a thread tends to transition into the waiting stage. A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.
- **Timed waiting -** A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transition back to the runnable state when that time interval expires or when the event it is waiting for occurs.
- **Terminated -** A runnable thread enters the terminated state when it completes its task or otherwise terminates.

Check your progress 4

1. Draw a diagram showing complete life cycle of a thread.
2. Explain multithreading.

.....

.....

.....

.....

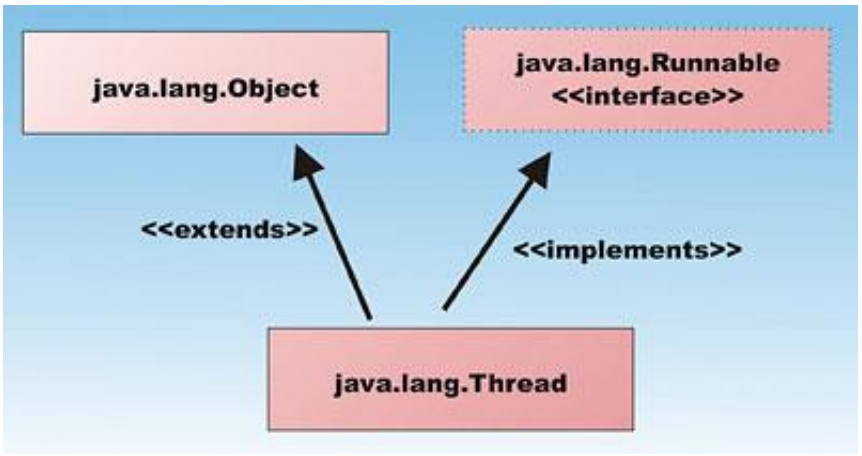
.....

.....

.....

.....

3.6 The Thread Class and the Runnable Interface



The easiest way to create a thread is to create a class that implements the Runnable interface.

To implement Runnable, a class need only implement a single method called run ( ), which is declared like this:

public void
run( )

You will define the code that constitutes the new thread inside run() method. It is important to understand that run() can call other methods, use other classes and declare variables, just like the main thread can.

After you create a class that implements Runnable, you will instantiate an object of type Thread from within that class. Thread defines several constructors. The one that we will use is shown here:

Thread(Runnable threadOb, String threadName)

Here *threadOb* is an instance of a class that implements the Runnable interface and the name of the new thread is specified by *threadName*.

After the new thread is created, it will not start running until you call its start ( ) method, which is declared within Thread. The start ( ) method is shown here:

Check your progress 5

1. What is the easiest way to create a thread?
2. Explain how to implement runnable interface.

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

3.7 Thread Priorities

Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled.

Java priorities are in the range between MIN_PRIORITY (a constant of 1) and MAX_PRIORITY (a constant of 10). By default, every thread is given priority NORM_PRIORITY (a constant of 5).

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 and are very much platform dependent.

Creating a Thread:

Java defines two ways in which this can be accomplished:

- You can implement the runnable interface.
- You can extend the thread class, itself

Check your progress 6

1. Define the two ways to create a thread.
2. Give the range of Java priorities?

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

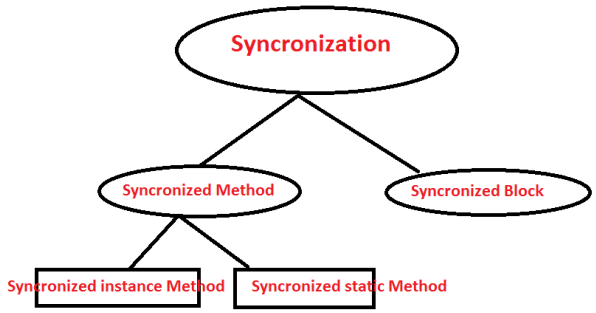
.....

.....

.....

.....

3.8 Synchronisation



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 synchronisation is achieved is called thread synchronisation.

The synchronised keyword in Java creates a block of code referred to as a critical section. Every Java object with a critical section of code gets a lock associated with the object. To enter a critical section, a thread needs to obtain the corresponding object's lock.

This is the general form of the synchronised statement:

synchronised(object)

{

//statements to be synchronised

}

Here, object is a reference to the object being synchronised. A synchronised 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 example, using a synchronised block within the run ( ) method:

//The given below program uses a synchronised block

//File Name Call.java

classCallme

```
{
voidmessage(String msg)
{
System.out.println (“[“ +msg);
Try
{
Thread. sleep (1000);
}
catch (InterruptedException e)
{
System.out.println (“Interrupted”);
}
System.out.println (“]”);
}
}
```

//File Name: Caller java

```
class Caller implements Runnable
{
String msg
Callme target;
Thread t;
public Caller (Callmetarg, String s)
{
target=targ
msg=s;
t=new thread (this);
t.start ( )
```

```
}
//synchronize calls to call ( )
public void run ( )
{
synchronised  (target)          //synchronised  block
{
target.call(msg)
}
}
}
//File Name: Synch.java
Class synch
{
public static void main (String args[])
{
Callme target=new Callme( );
Caller ob1=new Caller (target, ”Hello”);
Caller ob2=new Caller (target, ”Synchronised ”);
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”)
```

```
}
}
}
```

The above program will produce the given output:

[Hello]

[World]

[Synchronised]

Check your progress 7

1. Write the general form of synchronised statement.
2. What do you mean by critical section?

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

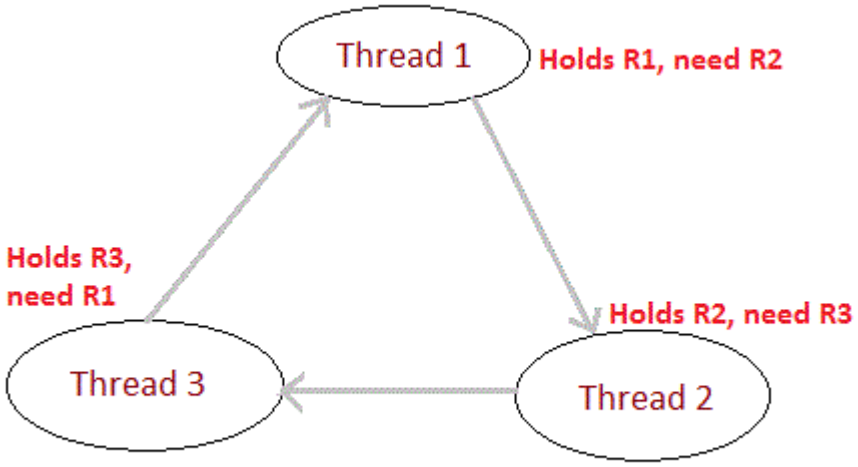
.....

.....

.....

.....

3.9 Deadlock



Deadlock Condition

A special type of error that you need to avoid that relates specifically to multitasking is deadlock, which occurs when two threads have a circular dependency on a pair of synchronised objects.

For example, suppose one thread enters the monitor on object X and another thread enters the monitor on object Y. If the thread in X tries to call any synchronised method on Y, it will block as expected. However, if the thread in Y, in turn, tries to call any synchronised method on X, the thread waits forever, because to access X, it would have to release its own lock on Y so that the first thread could complete.

Example:

To understand deadlock fully, it is useful to see it in action. The next example creates two classes, A and B, with methods foo ( ) and bar( ), respectively, which pause briefly before trying to call a method in the other class.

The main class, named Deadlock, creates an A and a B instance and then starts a second thread to set up the deadlock condition. The foo ( ) and bar( ) methods use sleep( ) as a way to force the deadlock condition to occur.

```
Class A
{
Synchronised void foo (B b)
{
String name = Thread.currentThread ( ). getName ( );
System.out.println (name + “entered a.foo”);
try
{
Thread.sleep (2000);
} catch (Exception e)
{
System.out.println (“A Interrupted”);
}
System.out.println (name + “trying to call B.last ( )”);
b.last ( );
}
Synchronised void last ( )
{
System.out.println (“Inside A.last”);
}
}

Class B
{
Synchronised void bar (A a)
{
String name = Thread.currentThread ( ).getName( );
```

```
System.out.println (name + "entered B.bar");
try
{
Thread.sleep (1000);
} catch (Exception e)
{
System.out.println ("B Interrupted");
}
System.out.println (name + "trying to call A.last ( )");
a.last ( );
}
Synchronised void last ( )
{
System.out.println ("Inside A.last");
}
}

Class Deadlock implements Runnable
{
A a= new A ( );
B b = new B ( );
Deadlock ( );
{
Thread.currentThread ( ).setName("Main Thread");
Thread t = new Thread (this, "Racing Thread");
t.start ( );
a.foo (b);        //get lock on a in this thread
System.out.println ("Back in main thread");
```

```
}
public void run ( )
{
b.bar (a);          //get lock on b in other thread
System.out.println ("Back in other thread");
}
public static void main (String args [ ])
{
new Deadlock ( );
}
}
```

The output of the above program will be

```
MainThread entered A.foo
RacingThread entered B.bar
MainThread try to call B.last ( )
RacingThread trying to call
A.last ( )
```

As the program is deadlocked, you need to press CTrl-C to end the program. You can see a full thread and monitor cache dump by pressing Ctrl-Break.

You will see that Racing Thread owns the monitor on b, while it is waiting for the monitor on a. At the same time, Main Thread owns a and is waiting to get b. This program will never complete.

As this example illustrates, if your multithreaded program locks up occasionally, deadlock is one of the first condition that you should check for.

Ordering Locks

A common threading trick to avoid the deadlock is to order the locks. By ordering the locks, it gives threads a specific order to obtain multiple locks.

Check your progress 8

1. What is the trick to avoid deadlock?
2. What should be checked if the multithreaded program is locked occasionally?

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

3.10 Suspending, Resuming and Stopping Threads

While the suspend ( ), resume ( ) and stop( ) methods defined by Thread class seem to be a perfectly reasonable and convenient approach to managing the execution of threads, they must not be used for new Java programs and obsolete in newer versions of Java.

The following example illustrates how the wait ( ) and notify ( ) methods that are inherited from Object can be used to control the execution of a thread.

This example is similar to the program in the previous section. However, the deprecated method calls have been removed. Let us consider the operation of this program.

The NewThread class contains a boolean instance variable named suspendFlag, which is used to control the execution of the thread. It is initialised to false by the constructor.

The run( ) method contains a synchronised statement block that checks suspendFlag. If that variable is true, the wait( ) method is invoked to suspend the execution of the thread. The mysuspend( ) method sets suspendFlag to true. The myresume( ) method sets suspendFlag to false and invokes notify( ) to wake up the thread. Finally, the main( ) method has been modified to invoke the mysuspend( ) and myresume( ) methods.

Check your progress 9

1. Write a multithreaded program to create threads and use suspend and resume methods to transfer control.
2. Explain the execution of suspend, resume and stop threads.

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

3.11 Let Us Sum Up

This section shows you how to determine the given arrays are same or not. The given program illustrates you how to compare arrays according to the content of that.in this section, you can see that the given program initializes two arrays and input five numbers from user through the keyboard. And then the program checks whether the given taken both arrays are same or not. This comparison operation is performed by using the equals () method of Arrays class. This section explains the implementation of the hash table. What is the hash table and how to create that? Hash Table holds the records according to the unique key value. It stores the non-contiguous key for several values. Hash Table is created using an algorithm (hashing function) to store the key and value regarding to the key in the hash bucket. If you want to store the value for the new key and if that key is already exists in the hash bucket then the situation known as collision. It occurs when there is the problem in the hash table maintained by the hashing function.

Java provides built-in support for multithreaded programming. A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread and each thread defines a separate path of execution.

A multithreading is a Specialized form of multitasking. Multitasking threads require less overhead than multitasking processes. There is another term to be defined related to threads: process: A process consists of the memory space allocated by the operating system that can contain one or more threads. A thread cannot exist on its own; it must be a part of a process. A process remains running until all of the non-daemon threads are done executing.

There is also understanding related to Thread life cycle. A thread goes through various stages in its life cycle. For example, a thread is born, started, runs and then dies. Following diagram shows complete life cycle of a thread.

There is a mention about Thread Priority, every Java thread has a priority that helps the operating system determine the order in which threads are scheduled.

Java priorities are in the range between MIN_PRIORITY (a constant of 1) and MAX_PRIORITY (a constant of 10). By default, every thread is given priority NORM_PRIORITY (a constant of 5).

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 and are very much platform dependent. There is a learning related to 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 synchronization is achieved is called thread synchronization.

The synchronized keyword in Java creates a block of code referred to as a critical section. Every Java object with a critical section of code gets a lock associated with the object. To enter a critical section, a thread needs to obtain the corresponding object's lock.

There is understanding related to a Dead lock, a special type of error that you need to avoid that relates specifically to multitasking is deadlock, which occurs when two threads have a circular dependency on a pair of synchronised objects.

About Suspending, Resuming and Stopping Threads

Further there is a understanding about While the suspend ( ), resume ( ) and stop( ) methods defined by Thread class seem to be a perfectly reasonable and convenient approach to managing the execution of threads, they must not be used for new Java programs and obsolete in newer versions of Java.

3.12 Suggested Answer for Check Your Progress

Check your progress 1

Answers: See Section 3.2

Check your progress 2

Answers: See Section 3.3

Check your progress 3

Answers: See Section 3.4

Check your progress 4

Answers: See Section 3.5

Check your progress 5

Answers: See Section 3.6

Check your progress 6

Answers: See Section 3.7

Check your progress 7

Answers: See Section 3.8

Check your progress 8

Answers: See Section 3.9

Check your progress 9

Answers: See Section 3.10

3.13 Glossary

1. **Collision -** If you want to store the value for the new key and if that key is already exists in the hash bucket then the situation known as **collision** occurs which is the problem in the hash table maintained by the hashing function.
2. **Multithreading -** A multithreading is a Specialized form of multitasking. Multitasking threads require less overhead than multitasking processes.
3. **Synchronized keyword -** The synchronized keyword in Java creates a block of code referred to as a critical section. Every Java object with a critical section of code gets a lock associated with the object.

3.14 Assignment

Write a program to create multiple threads

3.15 Activities

1. Describe the important methods available in thread class.
2. Write a program to remove deadlock

3.16 Case Study

Every program has at least one thread. Programs without multithreading executes sequentially. That is, after executing one instruction the next instruction in sequence is executed. If a function is called then until the completion of the function the next instruction is not executed. Similarly if there is a loop then instruction safer loop only gets executed when the loop gets completed. Consider the following java program having three loops.

3.17 Further Reading

1. Core Java 2, 2 volumes, Cay S. Horstmann, Gary Cornell, The Sun Microsystems Press, 1999, Indian reprint 2000
2. Java 2, The Complete Reference, Patrick Naughton and Herbert Schildt, Tata McGraw Hill, 1999
3. Programming with Java, Ed. 2,E. Balagurusamy, Tata McGraw Hill, 1998, reprint, 2000
4. The Java Tutorial, Ed. 2, 2 volumes, Mary Campione and Kathy Walrath, Addison Wesley Longmans, 1998
5. The Java Language Specification, Ed. 2, James Gosling, Bill Joy, Guy Steele & Gilad Bracha, (Ed. 1 1996, Ed. 2 2000), Sun Microsystems, 2000
6. Using Java 2, Joseph L. Weber, Prentice Hall, Eastern Economy Edition, 2000