Object-oriented Programming in Java

Session: 8

Multithreading and Concurrency

Objectives

- Define multithreading
- Differentiate between multithreading and multitasking
- Explain the use of `isAlive()` and `join()` method
- Explain race conditions and ways to overcome them
- Describe intrinsic lock and synchronization
- Describe atomic access
- Describe the use of `wait()` and `notify()` methods
- Define deadlock and the ways to overcome deadlock
- Explain `java.util.concurrent`

Introduction

- A thread performs a certain task and is the smallest unit of executable code in a program.
- Multitasking is the ability of the operating system to perform two or more tasks.
- Multitasking can be process-based or thread-based.
- In process-based multitasking, two or more programs run concurrently.
- In thread-based multitasking, a program performs two or more tasks at the same time.
- Thus, multithreading can be defined as the concurrent running of the two or more parts of the same program.

Types of Multitasking

There are basically two types of multitasking is in use among operating systems.

These are:

- Preemptive
  - In this case, the operating system controls multitasking by assigning the CPU time to each running program.
  - This approach is used in Windows 95 and 98.
- Cooperative
  - In this approach, related applications voluntarily surrender their time to one another.
  - This was used in Windows 3.x.
- Multithreading is a technique similar to multitasking and involves the creation of one or more threads within a program to enable number of tasks to run concurrently or in parallel.

Features of Multithreading

- Multithreading also supports the following features:

1. Managing many tasks concurrently.
2. Distinguishing between tasks of varying priority.
3. Allowing the user interface to remain responsive, while allocating time to background tasks.

Differences Between Multithreading and Multitasking

Multithreading	Multitasking
In a multithreaded program, two or more threads can run concurrently.	In a multitasking environment, two or more processes run concurrently.
Multithreading requires less overhead. In case of multithreading, threads are lightweight processes. Threads can share same address space and inter-thread communication is less expensive than inter-process communication.	Multitasking requires more overhead. Processes are heavyweight tasks that require their own address space. Inter-process communication is very expensive and the context switching from one process to another is costly.

Need for Multithreading

Multithreading is needed for the following reasons:

1. Multithreading increases performance of single-processor systems, as it reduces the CPU idle time.
2. Multithreading encourages faster execution of a program when compared to an application with multiple processes, as threads share the same data whereas processes have their own sets of data.
3. Multithreading introduces the concept of parallel processing of multiple threads in an application which services a huge number of users.

Implementing Multithreading [1-3]

- The code in the following example creates multiple threads, displays the count of the threads, and displays the name of each running child thread within the `run()` method:

```
/**
* Creating multiple threads using a class derived from Thread
    class
*/
package test;
 /**
* MultipleThreads is created as a subclass of the class Thread
*/
public class MultipleThreads extends Thread {
    /* Variable to store the name of the thread */
    String name;
     /**
      * This method of Thread class is overridden to specify the
        action that will be done when the thread begins execution.
      */
```

Implementing Multithreading [2-3]

```
public void run() {
   while(true) {
          name = Thread.currentThread().getName();
          System.out.println(name);
           try
                  {
                  Thread.sleep(500);
                  }
           catch( InterruptedException e)
                  {
                   break;
                  }
                }
          }
    /**
    * This is the entry point for the MultipleThreads class.
     */
```

Implementing Multithreading [3-3]

```
    public static void main(String args[]) {
       MultipleThreads t1 = new MultipleThreads();
       MultipleThreads t2 = new MultipleThreads();
       t1.setName("Thread2");
       t2.setName("Thread3");
       t1.start();
       t2.start();
    System.out.println("Number of threads running: " + Thread.
    activeCount());
    }
}
```

- In the code, the `main()` method creates two child threads by instantiating the `MultipleThreads` class which has been derived from the `Thread` class.
- When the `start()` method is invoked on the child thread objects, the control is transferred to the `run()` method which will begin thread execution.
- As soon as the child threads begin to execute, the number of active threads is printed in the `main()` method.

isAlive() Method

- A thread is considered to be alive when it is running.
- If the thread is alive, then the boolean value true is returned. If the `isAlive()` method returns false, it is understood that the thread is either in new state or in terminated state.
- **Syntax:** `public final boolean isAlive()`

Code Snippet

```
. . .
 public static void main(String [] args)
{
ThreadDemo Obj = new ThreadDemo();
Thread t = new Thread(Obj);   System.out.println("The thread is
alive :" + t.isAlive());
}
 . . .
```

`join()` Method [1-3]

- The `join()` method causes the current thread to wait until the thread on which it is called terminates.
- The `join()` method performs the following operations:
  - This method allows specifying the maximum amount of time that the program should wait for the particular thread to terminate.
  - It throws `InterruptedException` if another thread interrupts it.
  - The calling thread waits until the specified thread terminates.

join() Method [2-3]

- **Syntax:** `public final boolean isAlive()`

Code Snippet

```
try
{
System.out.println("I am in the main and waiting for the thread
to finish");
// objTh is a Thread object
objTh.join();
}
catch( InterruptedException e)
{
System.out.println("Main thread is interrupted");
}
. . .
```

join() Method [3-3]

The join() method of the Thread class has two other overloaded versions:

- void join(long timeout):
  - In this type of join() method, an argument of type long is passed.
  - The amount of timeout is in milliseconds.
  - This forces the thread to wait for the completion of the specified thread until the given number of milliseconds elapses.

- void join(long timeout, int nanoseconds):
  - In this type of join() method arguments of type long and integer are passed.
  - The amount of timeout is given in milliseconds in addition to a specified amount of nanoseconds.
  - This forces the thread to wait for the completion of the specified thread until the given timeout elapses.

Different Methods of `Thread` Class for Multithreading [1-4]

- The following example displays the use of the different methods of the `Thread` class:

```
/*
* Using the isAlive and join methods
*/ package test;
/** ThreadDemo inherits from Runnable interface */
class ThreadDemo implements Runnable {
        String name;
        Thread objTh;
        /* Constructor of the class */
        ThreadDemo(String str) {
                name = str;
                objTh = new Thread(this, name);
                System.out.println("New Threads are starting : " + objTh);
                objTh.start();
        }
        public void run() {
                 try {
```

Different Methods of `Thread` Class for Multithreading [2-4]

```
                for (int count = 0;count < 2;count++)
                {
                        System.out.println(name + " : "+count);
                        objTh.sleep(1000);
                }
                }
                catch(InterruptedException e)
                {
                        System.out.println(name + "  interrupted");
                }
                System.out.println(name + "  exiting");
        }

        public static void main(String [] args)
        {
        ThreadDemo objNew1 = new ThreadDemo("one");
        ThreadDemo objNew2 = new ThreadDemo ("two");
        ThreadDemo objNew3 = new ThreadDemo ("three");
        System.out.println("First thread is alive :" + objNew1.objTh.
                isAlive());
        System.out.println("Second thread is alive :" + objNew2.objTh.
                isAlive());
        System.out.println("Third thread is alive :" + objNew3.objTh.
                isAlive());
```

Different Methods of `Thread` Class for Multithreading [3-4]

```
try {
        System.out.println("In the main method, waiting for the
        threads to finish");
        objNew1.objTh.join();
        objNew2.objTh.join();
        objNew3.objTh.join();
}
catch(InterruptedException e) {
System.out.println("Main thread is interrupted");
}
System.out.println("First thread is alive :" + objNew1.objTh.
isAlive());
System.out.println("Second thread is alive :" + objNew2.objTh.
isAlive());
System.out.println("Third thread is alive :" + objNew3.objTh.
isAlive());
System.out.println("Main thread is over and exiting");
} }
```

Different Methods of `Thread` Class for Multithreading [4-4]

- In the code, three thread objects are created in the `main()` method.
- The `isAlive()` method is invoked by the three thread objects to test whether they are alive or dead.
- Then the `join()` method is invoked by each of the thread objects.
- The `join()` method ensures that the main thread is the last one to terminate.
- Finally, the `isAlive()` method is invoked again to check whether the threads are still alive or dead.

Race Conditions

- In multithreaded programs, several threads may simultaneously try to update the same resource, such as a file.
- This leaves the resource in an undefined or inconsistent state. This is called race condition.
- In general, race conditions in a program occur when:

1. Two or more threads share the same data between them.
2. Two or more threads try to read and write the shared data simultaneously.

Synchronized Blocks [1-3]

- The synchronized block contains code qualified by the `synchronized` keyword.
- A lock is assigned to the object qualified by `synchronized` keyword.
- When a thread encounters the `synchronized` keyword, it locks all the doors on that object, preventing other threads from accessing it.
- A lock allows only one thread at a time to access the code.
- When a thread starts to execute a synchronized block, it grabs the lock on it.
- Any other thread will not be able to execute the code until the first thread has finished and released the lock.
- The lock is based on the object and not on the method.

Synchronized Blocks [2-3]

- **Syntax**:

```
//synchronized block
synchronized(object)
{
        // statements to be synchronized
}
//synchronized method
synchronized method(...)
{
        // body of method
}
```

Synchronized Blocks [3-3]

Code Snippet

```
. . .
class Account
{
    float balance = 0.0;
    public synchronized void deposit(float value)
    {
        balance = balance + value;
    }
}
. . .
```

- In this snippet, the `deposit()` method of `Account` class has been `synchronized` by using a `synchronized` keyword.
- This method can be accessed by a single thread at a time from the several threads in a program.
- The synchronized method allows threads to access it sequentially.

Synchronized Method

- The synchronized method obtains a lock on the class object.
- This means that at a single point of time only one thread obtains a lock on the method, while all other threads need to wait to invoke the synchronized method.

Example of Synchronized Methods [1-4]

- The following example shows how to use a `synchronized` method:

```
/**
* Demonstrating synchronized methods.
*/
package test;
class One {
// This method is synchronized to use the thread safely
synchronized void display(int num) {
 num++;
System.out.print("" + num);
try {
 Thread.sleep(1000);
 } catch(InterruptedException e) {
  System.out.println("Interrupted");
 }
  System.out.println(" done");
 }
}
```

Example of Synchronized Methods [2-4]

```
class Two extends Thread {
int number;
One objOne;
public Two(One one_num, int num) {
 objOne = one_num;
 number = num;
}
public void run() {
// Invoke the synchronized method
objOne.display(number);
 }
}
class SynchMethod {
public static void main(String args[]) {
One objOne = new One();
int digit = 10;
```

Example of Synchronized Methods [3-4]

```
// Create three thread objects
Two objSynch1 = new Two(objOne);
Two objSynch2 = new Two(objOne);
Two objSynch3 = new Two(objOne);
objSynch1.start();
objSynch2.start();
objSynch3.start();
}
}
```

- The class `One` has a method `display()` that takes an `int` parameter.
- This number is displayed with a suffix '`done`'.
- The `Thread.sleep(1000)` method pauses the current thread after the method `display()` is called.

Example of Synchronized Methods [4-4]

- The constructor of the class `Two` takes a reference to an object `t` of the class `One` and an integer variable.
- Here, a new thread is also created. This thread calls the method `run()` of the object `t`.
- The main class `SynchDemo` instantiates the class `One` as a object `objOne` and creates three objects of the class `Two`.
- The same object `objOne` is passed to each `Two` object.
- The method `join()` makes the caller thread wait till the calling thread terminates.

Issues in Synchronization

- It is not always possible to achieve synchronization by creating `synchronized` methods within classes.
- The reason for this is as follows:

Consider a case where the programmer wants to synchronize access to objects of a class, which does not use synchronized methods.

Also assume that the source code is unavailable because either a third party created it or the class was imported from the built–in library.

- In such a case, the keyword `synchronized` cannot be added to the appropriate methods within the class.
- Therefore, the problem here would be how to make the access to an object of this class `synchronized`.
- This could be achieved by putting all calls to the methods defined by this class inside a `synchronized` block.

Intrinsic Locks and Synchronization

- Synchronization is built around the concept of an in-built monitor, which is also referred to as intrinsic lock or monitor lock.
- The monitor lock enforces exclusive access to the thread objects and creates a relationship between the thread action and any further access of the same lock.
- Every object is connected to an intrinsic lock.
- Typically, a thread acquires the object's intrinsic lock before accessing its fields, and then releases the intrinsic lock.
- In this span of acquiring and releasing the intrinsic lock, the thread owns the intrinsic lock.
- No other thread can acquire the same lock.
- The other thread will block when it attempts to acquire the lock.
- When a thread invokes a synchronized method, the following occurs:
  - It automatically acquires the intrinsic lock for that method's object.
  - It releases when the method returns.

Reentrant Synchronization

- This occurs when a thread acquires a lock that it already owns.
- In this event, the synchronized code invokes a method directly or indirectly that also contains synchronized code.
- Both sets of code use the same lock.
- With reentrant synchronization, it is easy for synchronized code to avoid a thread cause itself to block.

Atomic Access

- In programming, an atomic action occurs all at once.
- Following are the features of an atomic action:
  - It occurs completely or it doesn't occur at all.
  - Effects of an atomic action are visible only after the action is complete.
- Following are the actions that can be specified as atomic:
  - Reads and writes are atomic for reference variables and for most primitive variables (all types except long and double).
  - Reads and writes are atomic for all variables declared volatile.
- Atomic actions cannot be interleaved.
- There will no thread interference when atomic actions are used.

Wait-notify Mechanism

- The wait-notify mechanism acts as the traffic signal system in the program.
- It allows the specific thread to wait for some time for other running thread and wakes it up when it is required to do so.
- In other words, the wait-notify mechanism is a process used to manipulate the `wait()` and `notify()` methods.
- This mechanism ensures that there is a smooth transition of a particular resource between two competitive threads.
- It also oversees the condition in a program where one thread is:

Allowed to wait for the lock of a synchronized block of resource currently used by another thread.

Notified to end its waiting state and get the lock of that synchronized block of resource.

`wait()` Method [1-2]

- The `wait()` method causes a thread to wait for some other thread to release a resource.
- It forces the currently running thread to release the lock or monitor, which it is holding on an object.
- Once the resource is released, another thread can get the lock and start running.
- The `wait()` method can only be invoked only from within the synchronized code.
- The following points should be remembered while using the `wait()` method:

1. The calling thread gives up the CPU and lock.
2. The calling thread goes into the waiting state of monitor.

wait() Method [1-2]

- **Syntax :** `public final void wait()`

Code Snippet

```
. . .
public synchronized void takeup()
{
while (!available)  {
try      {
         System.out.println("Philosopher is waiting for the other
         chopstick");
         wait();
         }
         catch( InterruptedException e)
         {
         }
}
available = false;
}
. . .
```

`notify()` Method [1-2]

- The `notify()` method alerts the thread that is waiting for a monitor of an object.
- This method can be invoked only within a synchronized block.
- If several threads are waiting for a specific object, one of them is selected to get the object.
- The scheduler decides this based on the need of the program.
- The `notify()` method functions in the following ways:

The waiting thread moves out of the waiting space of the monitor and into the ready state.

The thread that was notified is now eligible to get back the monitor's lock before it can continue.

notify() Method [2-2]

- **Syntax:** `public final void notify()`

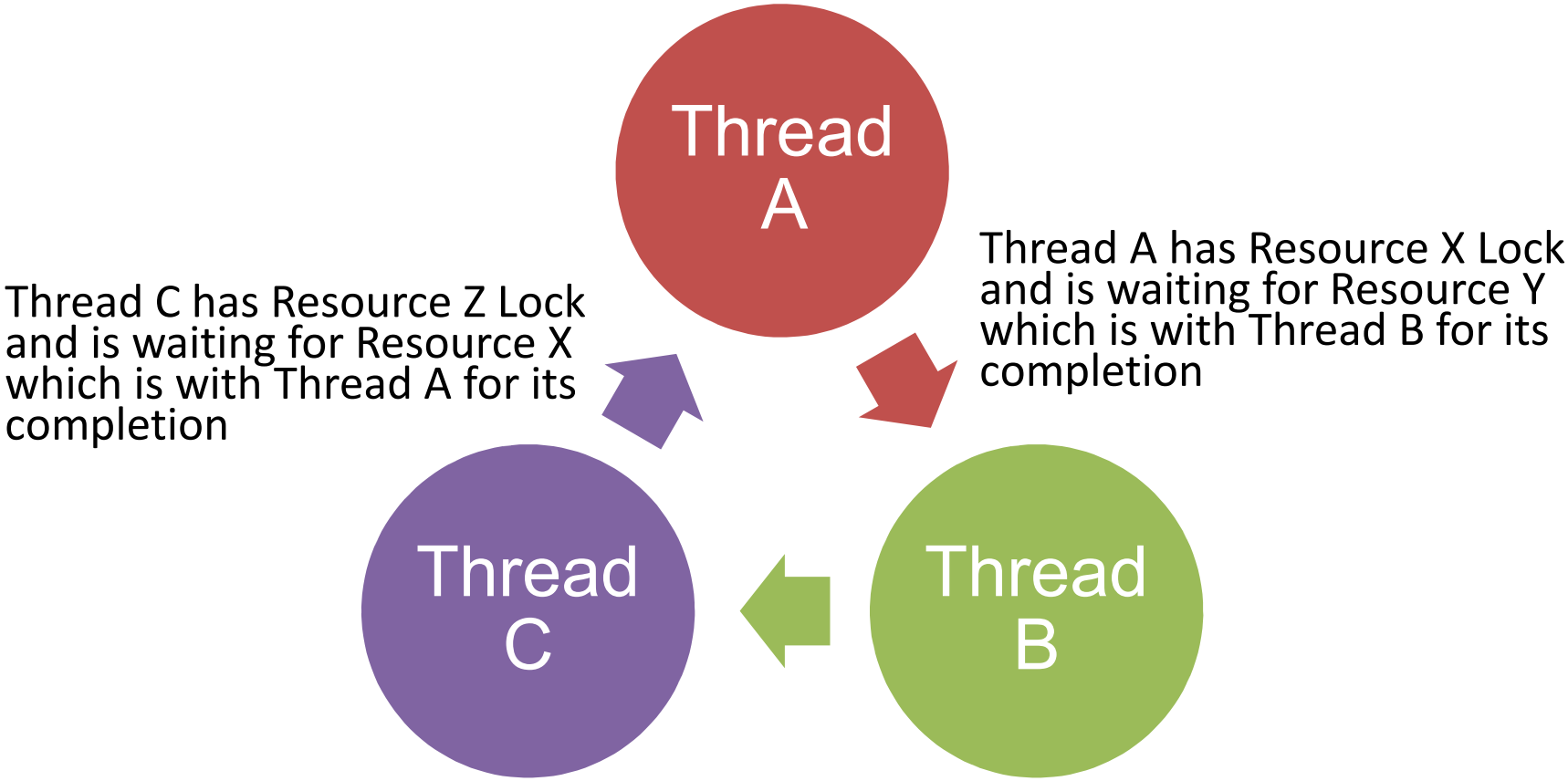
Code Snippet

```
. . .
public synchronized void putdown()
{
    available = true;
    notify();
}
. . .
```

Deadlocks [1-2]

- Deadlock describes a situation where two or more threads are blocked forever, waiting for the others to release a resource.
- At times, it happens that two threads are locked to their respective resources, waiting for the corresponding locks to interchange the resources between them.
- In that situation, the waiting state continues forever as both are in a state of confusion as to which one will leave the lock and one will get into it.
- This is the deadlock situation in a thread based Java program.
- The deadlock situation brings the execution of the program to a halt.

Deadlocks [2-2]



Example of Deadlock Condition [1-4]

- The following example demonstrates a deadlock condition:

```
/**
* Demonstrating Deadlock.
*/
package test;
/**
* DeadlockDemo implements the Runnable interface.
*/
public class DeadlockDemo implements Runnable
{
    public static void main(String args[])
    {
    DeadlockDemo objDead1 = new DeadlockDemo();
    DeadlockDemo objDead2 = new DeadlockDemo();
    Thread objTh1 = new Thread (objDead1);
    Thread objTh2 = new Thread (objDead2);
    objDead1.grabIt = objDead2;
    objDead2.grabIt = objDead1;
```

Example of Deadlock Condition [2-4]

```
            objTh1.start();
            objTh2.start();
            System.out.println("Started");
            try {
                    objTh1.join();
                    objTh2.join();
                }
            catch( InterruptedException e) {
            System.out.println("error occurred");
            }
            System.exit(0);
    }
    DeadlockDemo grabIt;
    public synchronized void run() {
            try {
                    Thread.sleep(500);
```

Example of Deadlock Condition [3-4]

```
                } catch( InterruptedException e) {
                        System.out.println("error occurred");
                 }
                grabIt.syncIt();
    }
    public synchronized void syncIt() {
        try {
                Thread.sleep(500);
                System.out.println("Sync");
        }
        catch(InterruptedException e) {
                System.out.println("error occurred");
         }
     System.out.println("In the syncIt() method");
    }
}// end class
```

Example of Deadlock Condition [4-4]

- The program creates two child threads.
- Each thread calls the synchronized `run()` method.
- When thread `objTh1` wakes up, it calls the method `syncIt()` of the `DeadlockDemo` object `objDead1`.
- Since the thread `objTh2` owns the monitor of `objDead2`, thread `objTh1` begins waiting for the monitor.
- When thread `objTh2` wakes up, it tries to call the method `syncIt()` of the `DeadlockDemo` object `objDead2`.
- At this point, `objTh2` also is forced to wait since `objTh1` owns the monitor of `objDead1`.
- Since both threads are waiting for each other, neither will wake up. This is a deadlock condition.
- The program is blocked and does not proceed further.

Overcoming the Deadlock

- The following things in a program can be done to avoid deadlock situations in it:

1. Avoid acquiring more than one lock at a time.
2. Ensure that in a Java program, you acquire multiple locks in a consistent and defined order.

java.util.concurrent Collections

- Following are some of these collections that are categorized by the collection interfaces:
  - **`BlockingQueue`**: This defines a FIFO data structure that blocks or times out when data is added to a full queue or retrieved from an empty queue.
  - **`ConcurrentMap`**: This is a subinterface of java.util.Map that defines useful atomic operations.
  - **`ConcurrentNavigableMap`**: This is a subinterface of ConcurrentMap that supports approximate matches.

Atomic Variables [1-2]

- The `java.util.concurrent.atomic` package defines classes that support atomic operations on single variables.
- All classes include get and set methods that functions similar to reads and writes on volatile variables.
- Therefore, a set has a happens-before relationship with any successive get on the same variable.
- The following Code Snippet displays the use of `AtomicVariableApplication` class:

```
public class AtomicVariableApplication {
 private final AtomicInteger value = new AtomicInteger(0);
 public int getValue() {
  return value.get();
}
```

Atomic Variables [2-2]

```
public int getNextValue() {
 return value.incrementAndGet();
}
public int getPreviousValue() {
 return value.decrementAndGet();
}
public static void main(String[] args) {
 AtomicVariableApplication obj = new AtomicVariableApplication();
 System.out.println(obj.getValue());
 System.out.println(obj.getNextValue());
 System.out.println(obj.getPreviousValue());
}
}
```

Executors and Executor Interface

- Objects that separate thread management and creates them from the rest of the application are called executors.
- The `java.util.concurrent` package defines the following three executor interfaces:
  - **`Executor`**: This helps launch new tasks.
  - **`ExecutorService`**: This is a subinterface of `Executor` and helps manage the development of the executor tasks and individual tasks.
  - **`ScheduledExecutorService`**: This is a subinterface of `ExecutorService` and helps periodic execution of tasks.

ThreadPools

- Thread pools have worker threads that help create threads and thus minimize the overhead.
- Certain executor implementations in `java.util.concurrent` use thread pools.
- Thread pools are often used to execute multiple tasks.
- Allocating and deallocating multiple thread objects creates a considerable memory management overhead in a large-scale application.
- Fixed thread pool is a common type of thread pool that includes the following features:
  - There are a specified number of threads running.
  - When in use if a thread is terminated, it is automatically replaced with a new thread.
  - Applications using fixed thread pool services HTTP requests as quickly as the system sustains.

For/Join Framework [1-5]

- This is an implementation of the `ExecutorService` interface.
- The framework helps work with several processors to boost the performance of an application.
- It uses a work-stealing algorithm and is used when work is broken into smaller pieces recursively.
- The Fork/Join framework allocates tasks to worker threads in a thread pool.
- There is the `ForkJoinPool` class in the fork/join framework.
- The class is an extension of the `AbstractExecutorService` class.
- The `ForkJoinPool` class implements the main work-stealing algorithm and executes ForkJoinTask processes.

For/Join Framework [2-5]

- The following Code Snippet displays the use of Fork/Join functionality:

```
package threadapplication;
import java.util.Random;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveTask;
public class ForkJoinApplication extends RecursiveTask<Integer> {

 private static final int SEQUENTIAL_THRESHOLD = 5;
 private final int[] data;
 private final int startData;
 private final int endData;


 public ForkJoinApplication(int[] data, int startValue, int
endValue) {
  this.data = data;
  this.startData = startValue;
  this.endData = endValue;
```

For/Join Framework [3-5]

```
 }
 public ForkJoinApplication(int[] data) {
  this(data, 0, data.length);
 }
 //recursive method which forks all small work units and then
joins them
 @Override
 protected Integer compute() {
   final int length = endData - startData;
   if (length < SEQUENTIAL_THRESHOLD) {
    return computeDirectly();
   }
   final int midValue = length / 2;
   final ForkJoinApplication leftValues = new
ForkJoinApplication(data, startData,
       startData + midValue);
     //forks all the small work units
     leftValues.fork();
```

For/Join Framework [4-5]

```
final ForkJoinApplication rightValues = new
ForkJoinApplication(data,
startData + midValue, endData);
//joins them all again using the join method
  return Math.max(rightValues.compute(), leftValues.join());

 }


private Integer computeDirectly() {
  System.out.println(Thread.currentThread() + " computing: " +
startData
+ " to " + endData);
   int max = Integer.MIN_VALUE;
   for (int i = startData; i < endData; i++) {
    if (data[i] > max) {
      max = data[i];
    }
   }
   return max;
```

For/Join Framework [5-5]

```
}
public static void main(String[] args) {
  // create a random object value set
 final int[] value = new int[20];
 final Random randObj = new Random();
 for (int i = 0; i < value.length; i++) {
  value[i] = randObj.nextInt(100);
 }

 // submit the task to the pool
 final ForkJoinPool pool = new ForkJoinPool(4);
 final ForkJoinApplication maxFindObj = new
ForkJoinApplication(value);
 //invokes the compute method
 System.out.println(pool.invoke(maxFindObj));
 }

}
```

Summary

- Multithreading is nothing but running of several threads in a single application.
- The isAlive() method tests whether the thread is in runnable, running, or terminated state.
- The join() method forces a running thread to wait until another thread completes its task.
- Race condition can be avoided by using synchronized block.
- The wait() method sends the running thread out of the lock or monitor to wait.
- The notify() method instructs a waiting thread to get in to the lock of the object for which it has been waiting.
- Deadlock describes a situation where two or more threads are blocked forever, waiting for each to release a resource.