Week 11

1. **MULTITHREADING**

Multithreading is a programming concept in which the application can create a small unit of tasks to execute in parallel. If you are working on a computer, it runs multiple applications and allocates processing power to them. A simple program runs in sequence and the code statements execute one by one. This is a single-threaded application. But, if the programming language supports creating multiple threads and passes them to the operating system to run in parallel, it’s called multithreading.

**Multithreading vs Multiprocessing**

When we talk about multithreading, we don’t care if the machine has a 2-core processor or a 16-core processor. Our work is to create a multithreaded application and let the OS handle the allocation and execution part. In short, multithreading has nothing to do with multiprocessing.

A diagram of a computer process

Description automatically generated with medium confidence

**Fig 1: Multithreading vs Multiprocessing**

**How does Java Support Multithreading?**

Java has great support for multithreaded applications. Java supports multithreading through Thread class. Java Thread allows us to create a lightweight process that executes some tasks. We can create multiple threads in our program and start them. Java runtime will take care of creating machine-level instructions and work with OS to execute them in parallel.

1. **THREAD SYNCHRONIZATION**

Synchronization in Java is the process that allows only one thread at a particular time to complete a given task entirely. By default, the JVM gives control to all the threads present in the system to access the shared resource, due to which the system approaches race condition.

Imagine a classroom where three teachers are simultaneously present to teach the same class. In this scenario, the classroom acts as a shared resource, and the three teachers are the threads. All of them can’t teach at the same time. This scenario when looked at in the context of a computer language is referred to as Race Condition, where there are multiple threads present to do a given task.

To solve the above problem, we use Synchronization in Java. It is also a good practice especially when the threads update the data concurrently.

**Need to Use Synchronization in Java**

* To prevent thread interleaving or interference
* To provide consistency to the program

**How Is It Carried Out?**

Synchronization in Java is done by using the synchronized keyword. This keyword can be used on top of a method or a block.

**Types of Synchronization**

There are two types of Synchronization in Java, as mentioned below-

* **Process Synchronization**

When multiple threads are executed simultaneously process synchronization ensures that they reach a particular state and agree to a certain set of actions.

* **Thread Synchronization**

When multiple threads want to access the same resource, thread synchronization makes sure that only one thread gains access at a time.

**Syntax:**

|  |
| --- |
| synchronized(objectidentifier) {  // Access shared variables and other shared resources  } |

Here, the objectidentifier is a reference to an object whose lock associates with the monitor that the synchronized statement represents. Now we are going to see two examples, where we will print a counter using two different threads. When threads are not synchronized, they print counter value which is not in sequence, but when we print counter by putting inside synchronized() block, then it prints counter very much in sequence for both the threads.

**Multithreading Example without Thread Synchronization**

Here is a simple example which may or may not print counter value in sequence and every time we run it, it produces a different result based on CPU availability to a thread.

|  |
| --- |
| class PrintDemo {  public void printCount() {  try {  for(int i = 5; i > 0; i--) {  System.out.println("Counter --- " + i );  }  } catch (Exception e) {  System.out.println("Thread interrupted.");  }  }  }  class ThreadDemo extends Thread {  private Thread t;  private String threadName;  PrintDemo PD;  ThreadDemo( String name, PrintDemo pd) {  threadName = name;  PD = pd;  }    public void run() {  PD.printCount();  System.out.println("Thread " + threadName + " exiting.");  }  public void start () {  System.out.println("Starting " + threadName );  if (t == null) {  t = new Thread (this, threadName);  t.start ();  }  }  }  public class TestThread {  public static void main(String args[]) {  PrintDemo PD = new PrintDemo();  ThreadDemo T1 = new ThreadDemo( "Thread - 1 ", PD );  ThreadDemo T2 = new ThreadDemo( "Thread - 2 ", PD );  T1.start();  T2.start();  // wait for threads to end  try {  T1.join();  T2.join();  } catch ( Exception e) {  System.out.println("Interrupted");  }  }  }  **Output:**  Starting Thread - 1  Starting Thread - 2  Counter --- 5  Counter --- 4  Counter --- 3  Counter --- 5  Counter --- 2  Counter --- 1  Counter --- 4  Thread Thread - 1 exiting.  Counter --- 3  Counter --- 2  Counter --- 1  Thread Thread - 2 exiting. |

**Multithreading Example with Thread Synchronization**

Here is the same example which prints counter value in sequence and every time we run it, it produces the same result.

|  |
| --- |
| class PrintDemo {  public void printCount() {  try {  for(int i = 5; i > 0; i--) {  System.out.println("Counter --- " + i );  }  } catch (Exception e) {  System.out.println("Thread interrupted.");  }  }  }  class ThreadDemo extends Thread {  private Thread t;  private String threadName;  PrintDemo PD;  ThreadDemo( String name, PrintDemo pd) {  threadName = name;  PD = pd;  }    public void run() {  synchronized(PD) {  PD.printCount();  }  System.out.println("Thread " + threadName + " exiting.");  }  public void start () {  System.out.println("Starting " + threadName );  if (t == null) {  t = new Thread (this, threadName);  t.start ();  }  }  }  public class TestThread {  public static void main(String args[]) {  PrintDemo PD = new PrintDemo();  ThreadDemo T1 = new ThreadDemo( "Thread - 1 ", PD );  ThreadDemo T2 = new ThreadDemo( "Thread - 2 ", PD );  T1.start();  T2.start();  // wait for threads to end  try {  T1.join();  T2.join();  } catch ( Exception e) {  System.out.println("Interrupted");  }  }  }  **Output:**  Starting Thread - 1  Starting Thread - 2  Counter --- 5  Counter --- 4  Counter --- 3  Counter --- 2  Counter --- 1  Thread Thread - 1 exiting.  Counter --- 5  Counter --- 4  Counter --- 3  Counter --- 2  Counter --- 1  Thread Thread - 2 exiting. |

1. **INTERTHREAD COMMUNICATION**

Inter Thread Communication, as the name implies, is a method that allows many synchronized threads to communicate or interact with one another. In Java, there are two ways to implement inter-thread communication: using wait() and notify() methods and using the higher-level constructs of the java.util.concurrent package.

When a program is executed, it becomes a process, which can be further divided into threads. Threads are independent paths of execution within the same process, and they allow a program to perform multiple tasks simultaneously, which can improve performance and efficiency.

Inter-Thread Communication (Cooperation) is a mechanism that allows threads to exchange information or coordinate their execution. It enables threads to work together to solve a common problem or to share resources. ITC can involve synchronization mechanisms such as mutexes or semaphores, which ensure that critical sections of code are executed by only one thread at a time to avoid concurrency issues.

**What Is Polling And What Are The Problems With It?**

Polling is the process of continually testing a condition until it becomes true. Polling is usually done with the use of multiple loops, and it determines whether a condition is true or false. When we discover that a certain condition is true, we take a specific action. However, because this method uses numerous CPU cycles, it is wasteful, thus we employ inter thread communication to overcome the problem.

To avoid polling, Java provides three methods. These are the wait(), notify(), and notifyAll methods(). Because these methods are in the object class and are marked as final, they can be used in any class. They can only be utilized inside a synchronized block.

**Wait() Method**

Let's say we are currently running Thread1 and we want to run Thread2. Since inter thread communication can be done in a synchronized block only one thread can run at a time. So to run Thread2 we must "pause" Thread1. The wait() function helps us achieve this exact thing.

The wait() method aids inter thread communication by releasing the lock on the current or calling thread (Thread1 in the above example) and instructing it to sleep until another thread (Thread2 in the above example) enters the monitor and calls notify() or notifyAll(), or until a certain period of time has passed.

The current thread must own this object's monitor, hence the wait() function must be used from the synchronized method only; otherwise, an error will be thrown.

**Syntax:**

|  |
| --- |
| public final void wait()throws InterruptedException |

**Syntax:**

|  |
| --- |
| public final void wait(long timeout)throws InterruptedException |

**Notify() Method**

Now let's say after we completed our work in Thread2 and we want to return to Thread1. To do so, we need to inform Thread1 that it can now use the object to run itself. We can use the notify() method to do so. Because we used the wait() function in Thread1 and since only one thread can run at a time Thread1 is waiting for the object to run.

The notify method() aids inter thread communication by allowing a single thread that is waiting for the object's monitor to become a runnable thread by waking it up. The decision of which waiting thread to awaken is arbitrary and is determined by the implementation.

**Syntax:**

|  |
| --- |
| public final void notify() |

**NotifyAll() Method**

In a similar way to the notify() method, the notifyAll() method aids inter thread communication. The only difference is that it wakes up all of the object monitor waiting for threads, making them all runnable at the same time. The term "runnable" does not imply that the thread is currently running. It signifies the thread can obtain a lock before proceeding to execution.

**Syntax:**

|  |
| --- |
| public final void notifyAll() |

|  |
| --- |
| import java.util.\*;  public class ThreadExampleScalar  {  public static void main(String[] args) throws InterruptedException  {  //creating an Object on Game  final Game game = new Game();  // Create a thread object that calls game.csgo()  Thread t1 = new Thread(new Runnable()  {  @Override  public void run()  {  try  {  game.csgo();  }  catch(InterruptedException e)  {  e.printStackTrace();  }  }  });  // Create another thread object that calls  // game.valorant()  Thread t2 = new Thread(new Runnable()  {  @Override  public void run()  {  try  {  csgo.valorant();  }  catch(InterruptedException e)  {  e.printStackTrace();  }  }  });  // Start both threads  t1.start();  t2.start();  // t1 finishes before t2  t1.join();  t2.join();  }  // Game class  public static class Game  {  //csgo function to demonstrate wait() function  public void csgo()throws InterruptedException  {  // synchronized block so that only one thread runs at a time  synchronized(this)  {  System.out.println("csgo thread running.");  // releases the lock on shared resources and the csgo thread pauses  wait();  // and waits till some other method invokes notify().  System.out.println("Resumed csgo.");  }  }  //valorant function to demonstrate notify() function  public void valorant()throws InterruptedException  {  // this makes the csgo thread to run first.  Thread.sleep(1000);  Scanner s = new Scanner(System.in);  // synchronized block ensures only one thread  // running at a time.  synchronized(this)  {  System.out.println("Waiting for return key csgo paused valorant running.");  s.nextLine();  System.out.println("Return key pressed valorant finished.");  // notifies the csgo thread that it  // can wake up.  notify();  // Sleep  Thread.sleep(2000);  }  }  }  }  **Output:**  csgo thread running.  Waiting for return key csgo paused valorant running.  Return key pressed valorant finished.  Resumed csgo. |

In the main class, we create a new Game object. It runs csgo and valorant methods on two threads, t1 and t2, and then waits for them to finish. Let's look at how our csgo and valorant methods function together.

* First and foremost, using a synchronized block ensures that only one thread is active at any given time. The csgo thread also gets a kickstart because there is a sleep method right at the start of the consume loop.
* When we use the csgo method wait(), it performs two functions. It begins by releasing the lock it has on the Game object. Second, it puts the csgo thread in a waiting state until all other threads are finished. It can regain control of a Game object by using notify() or notifyAll() on the same object, if some other function wakes it up.
* As a result, as soon as the wait() function is invoked, the control transfers to valorant thread and prints -"Waiting for return key csgo paused valorant running." to show that the csgo thread is in a waiting state while the valorant thread is active.
* The valorant method calls notify() once we press the return key. It also accomplishes two goals: To begin with, unlike wait(), it does not release the lock on shared resources, therefore it's best to use notify just at the end of your method to get the desired outcome. Second, it informs the waiting threads that they are now free to wake up but only once the current procedure has finished.
* As you may have noticed, control does not instantly pass to the csgo thread even after notify(). The reason for this is that after notify(), we called Thread.sleep() . The valorant thread has a lock on a Game object, as we already know. It cannot be accessed by another thread until the lock has been released. As a result, the csgo thread can only reclaim control after the valorant thread has completed its sleep period and then terminated by itself.
* After a 2-second delay, the program finishes at the end.
* If we delete the notify() call from the valorant thread and run the program again, we will observe that it no longer terminates.

The reason behind this is easy to understand. When you asked for the csgo thread to wait, it just sat there waiting and never ended. A program continues to operate until all of its threads have ended. To avoid this, we can use wait(long timeout), which accepts a timeout parameter and terminates the program when the csgo thread has waited for the specified amount of time.

A screenshot of a computer

Description automatically generated

1. **LEGACY CLASSES**

Legacy classes and interfaces are the classes and interfaces that formed the collections framework in the earlier versions of Java and how now been restructured or re-engineered. They are fully compatible with the framework. All legacy classes and interface were redesign by JDK 5 to support Generics. In general, the legacy classes are supported because there is still some code that uses them.

The following are the legacy classes defined by java.util package

* Dictionary
* HashTable
* Properties
* Stack
* Vector
* There is only one legacy interface called Enumeration

**Enumeration interface**

Enumeration interface defines method to enumerate(obtain one at a time) through collection of objects.

* This interface is superseded(replaced) by Iterator interface.
* However, some legacy classes such as Vector and Properties defines several method in which Enumeration interface is used.
* It specifies the following two methods:
* boolean hasMoreElements() //It returns true while there are still more elements to extract, and returns false when all the elements have been enumerated.
* Object nextElement() //It returns the next object in the enumeration i.e. each call to nextElement() method obtains the next object in the enumeration. It throws NoSuchElementException when the enumeration is complete.

1. **VECTOR CLASS**

A vector is similar to a dynamic array whose size can be increased or decreased. Unlike arrays, it has no size limit and can store any number of elements. Since Java 1.2, it has been a part of the Java Collection framework. It's in the java.util package and implements the List interface, so we can use all of the List interface's methods here.

Vector is a data structure that is used to store a collection of elements. Elements can be of all primitive types like int, float, Object, etc. Vectors are dynamic in nature and accordingly, grow or shrink as per the requirement.

* Vector Class in Java is found in the java.util package.
* Vector class is a child class of the AbstractList class and implements the List interface. Therefore we can use all the methods of the List interface.
* Vectors are known to give ConcurrentModificationException when accessed concurrently at the time of modification.
* When a Vector is created, it has a certain capacity to store elements that can be defined initially. This capacity is dynamic in nature and can be increased or decreased.
* By definition, Vectors are synchronized, which implies that at a time, only one thread is able to access the code while other threads have to wait. Due to this, Vectors are slower in performance as they acquire a lock on a thread.

**Syntax:**

|  |
| --- |
| public class Vector<E> extends AbstractList<E> implements List<E>, RandomAccess, Cloneable, Serializable |

* Here, E denotes the Element Type
* Vector Class extends AbstractList and implements multiple interfaces like Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess interfaces.
* The directly known subclass is Stack.

In each allocation cycle, Vector will expand in accordance with the increment if one is supplied. However, if the increment is left unspecified, each allocation cycle doubles the vector's capacity. Three protected data members are defined by Vector:

* int capacityIncreament: Contains the value of the increment.
* int elementCount: Number of elements that are currently stored in the vector.
* Object elementData[]: The vector is kept in an array that is stored in it

**Common Errors in the Declaration of Vectors**

The following are typical mistakes made in the declaration of vectors:

* If the InitialSize of the vector defined is negative, Vector throws an IllegalArgumentException.
* It throws a NullPointerException if the specified collection is null.

**Constructors in Vectors**

* Vector(): A default vector of capacity 10 gets created while calling this constructor.
* Vector<E> defaultVector = new Vector<E>(); //Here E represents Element Type.
* Vector(int size): A vector is created with the given size as its capacity.
* Vector<E> initialCapacityVector = new Vector<E>(int size);
* Vector(int size, int increment): A vector is created with the given size as its initial capacity, and whenever the capacity needs to be increased, it is increased by the given increment count.
* Vector<E> incrementalVector = new Vector<E>(int size, int increment);
* Vector(Collection c): A Java vector is constructed from the given collection with the same order of elements as in the collection.
* Vector<E> vector = new Vector<E>(Collection c);

There are also three protected data members (Protected fields are the data members that can be accessed either within the class or from the derived class) in the Vector class.

* int capacityIncreament: Contains the increment value.
* int elementCount: Number of elements currently in vector stored in it.
* Object elementData[]: Array that holds the vector is stored in it.

While initializing vectors in Java, one should keep in mind some exceptions that can occur:

* IllegalArgumentException: This is thrown if the initial size of the vector defined is negative.
* NullPointerException: This is thrown if the specified collection passed in the constructor is null.

|  |
| --- |
| import java.util.Vector;  public class Main {  public static void main(String[] args) {  // creating default vector(with capacity equals to 10)  Vector<Integer> defaultVector = new Vector<Integer>();  // creating a vector with the Capacity equals to 100  Vector<Integer> fixedSizeVector = new Vector<Integer>(100);  fixedSizeVector.add(100);  fixedSizeVector.add(100);  fixedSizeVector.add(100);  // creating a vector of given Capacity = 30 and Increment=20  // Here vector capacity will increase by 20 when needed  Vector<Integer> incrementalVector = new Vector<Integer>(30, 20);  // creating a vector with the given collection  Vector<Integer> copyConstructorVector = new Vector<Integer>(  fixedSizeVector  );  System.out.println(  "Vector defaultVector has capacity : " +  defaultVector.capacity() +  " elements"  );  System.out.println(  "Vector fixedSizeVector of capacity : " +  fixedSizeVector.capacity() +  " elements"  );  System.out.println(  "Vector incrementalVector of capacity : " +  incrementalVector.capacity() +  " elements"  );  System.out.println(  "Vector copyConstructorVector of capacity : " +  copyConstructorVector.capacity() +  " elements"  );  }  }  **Output:**  Vector defaultVector has capacity : 10 elements  Vector fixedSizeVector of capacity : 100 elements  Vector incrementalVector of capacity : 30 elements  Vector copyConstructorVector of capacity : 3 elements |

* Firstly, we made a defaultVector with an initial capacity of 10.
* Then we created another vector, fixedSizeVector with Capacity = 100, and added 3 elements into it.
* Then we created a third vector, incrementalVector with Capacity 30 and its capacity will increase by 20 whenever we want to add elements more than its capacity.
* Then we create a vector, copyConstructorVector which is initialized from the elements of the fixedSizeVector.
* Notice that the size of fixedSizeVector and capacity of copyConstructorVector is the same.
* Then, we compare the capacity of all the vectors using the capacity() method.

**Increment of Vector Capacity**

By default, the vector increases its capacity by double. However, if an increment is specified in its constructor, Vector will grow in accordance with it in each allocation cycle.

1. **STACK CLASS**

A stack is a generic data structure that represents a LIFO (last in, first out) collection of objects allowing for pushing/popping elements in constant time.

**A diagram of a list

Description automatically generated**

**Fig 1: Hierarchy of stack class**

In stack, elements are stored and accessed in Last In First Out manner. That is, elements are added to the top of the stack and removed from the top of the stack.

**Creating a Stack**

In order to create a stack, we must import the java.util.Stack package first. Once we import the package, here is how we can create a stack in Java.

**Syntax:**

|  |
| --- |
| Stack<Type> stacks = new Stack<>(); |

**Methods of the Stack Class**

We can perform push, pop, peek and search operation on the stack. The Java Stack class provides mainly five methods to perform these operations. Along with this, it also provides all the methods of the Java Vector class.

* empty()- It returns true if nothing is on the top of the stack. Else, returns false.
* peek()- Returns the element on the top of the stack, but does not remove it.
* pop()-Removes and returns the top element of the stack. An ‘EmptyStackException’. An exception is thrown if we call pop() when the invoking stack is empty.
* push(Object element)- Pushes an element on the top of the stack.
* search(Object element)- It determines whether an object exists in the stack. If the element is found, It returns the position of the element from the top of the stack. Else, it returns -1.

|  |
| --- |
| import java.util.\*;  public class StackPushPopExample  {  public static void main(String args[])  {  //creating an object of Stack class  Stack <Integer> stk = new Stack<>();  System.out.println("stack: " + stk);  //pushing elements into the stack  pushelmnt(stk, 20);  pushelmnt(stk, 13);  pushelmnt(stk, 89);  pushelmnt(stk, 90);  pushelmnt(stk, 11);  pushelmnt(stk, 45);  pushelmnt(stk, 18);  //popping elements from the stack  popelmnt(stk);  popelmnt(stk);  //throws exception if the stack is empty  try  {  popelmnt(stk);  }  catch (EmptyStackException e)  {  System.out.println("empty stack");  }  }  //performing push operation  static void pushelmnt(Stack stk, int x)  {  //invoking push() method  stk.push(new Integer(x));  System.out.println("push -> " + x);  //prints modified stack  System.out.println("stack: " + stk);  }  //performing pop operation  static void popelmnt(Stack stk)  {  System.out.print("pop -> ");  //invoking pop() method  Integer x = (Integer) stk.pop();  System.out.println(x);  //prints modified stack  System.out.println("stack: " + stk);  }  }  **Output:**  stack: []  push -> 20  stack: [20]  push -> 13  stack: [20, 13]  push -> 89  stack: [20, 13, 89]  push -> 90  stack: [20, 13, 89, 90]  push -> 11  stack: [20, 13, 89, 90, 11]  push -> 45  stack: [20, 13, 89, 90, 11, 45]  push -> 18  stack: [20, 13, 89, 90, 11, 45, 18]  pop -> 18  stack: [20, 13, 89, 90, 11, 45]  pop -> 45  stack: [20, 13, 89, 90, 11]  pop -> 11  stack: [20, 13, 89, 90] |

**Reference links**

1. <https://www.linkedin.com/pulse/choice-between-multithreading-multi-processing-when-use-deepak-kumar-uxyvf/>
2. <https://www.shiksha.com/online-courses/articles/thread-synchronization-in-java/>
3. <https://www.tutorialspoint.com/java/java_thread_synchronization.htm>
4. <https://efaidnbmnnnibpcajpcglclefindmkaj/https://www.sietk.org/downloads/javabook.pdf>
5. <https://www.programiz.com/java-programming/stack>

**Discussion forum**

Explain interthread communication in Java.

**Multiple Choice Questions**

1. What will happen if two thread of the same priority are called to be processed simultaneously?

a) Anyone will be executed first lexographically

b) Both of them will be executed simultaneously

c) None of them will be executed

d) It is dependent on the operating system

Answer: d

2. Which of these statements is incorrect?

a) By multithreading CPU idle time is minimized, and we can take maximum use of it

b) By multitasking CPU idle time is minimized, and we can take maximum use of it

c) Two thread in Java can have the same priority

d) A thread can exist only in two states, running and blocked

Answer: d

3. What will be the output of the following Java code?

class multithreaded\_programing

{

public static void main(String args[])

{

Thread t = Thread.currentThread();

System.out.println(t);

}

}

a) Thread[5,main]

b) Thread[main,5]

c) Thread[main,0]

d) Thread[main,5,main]

Answer: d

4. What is the priority of the thread in the following Java Program?

class multithreaded\_programing

{

public static void main(String args[])

{

Thread t = Thread.currentThread();

System.out.println(t);

}

}

a) 4

b) 5

c) 0

d) 1

Answer: b

5. What is the name of the thread in the following Java Program?

class multithreaded\_programing

{

public static void main(String args[])

{

Thread t = Thread.currentThread();

System.out.println(t);

}

}

a) main

b) Thread

c) System

d) None of the mentioned

Answer: a

6. Which of these class object can be used to form a dynamic array?

a) ArrayList

b) Map

c) Vector

d) ArrayList & Vector

Answer: d

7. Which of these are legacy classes?

a) Stack

b) Hashtable

c) Vector

d) All of the mentioned

Answer: d

8. Which of these is the interface of legacy?

a) Map

b) Enumeration

c) HashMap

d) Hashtable

Answer: b

9. What is the name of a data member of class Vector which is used to store a number of elements in the vector?

a) length

b) elements

c) elementCount

d) capacity

Answer: c

10. Which of these methods is used to add elements in vector at specific location?

a) add()

b) set()

c) AddElement()

d) addElement()

Answer: d

**Long Answer:**

1. Explain legacy classes in Java.

**FAQs**

1. **What is the need to Use Synchronization in Java?**

To prevent thread interleaving or interference

To provide consistency to the program

1. **What are the types of Synchronization?**

There are two types of Synchronization in Java, as mentioned below-

* **Process Synchronization**

When multiple threads are executed simultaneously process synchronization ensures that they reach a particular state and agree to a certain set of actions.

* **Thread Synchronization**

When multiple threads want to access the same resource, thread synchronization makes sure that only one thread gains access at a time.

1. What is Inter-Thread Communication (Cooperation)?

It is a mechanism that allows threads to exchange information or coordinate their execution. It enables threads to work together to solve a common problem or to share resources. ITC can involve synchronization mechanisms such as mutexes or semaphores, which ensure that critical sections of code are executed by only one thread at a time to avoid concurrency issues.

1. Give the working of wait()?

This method aids inter thread communication by releasing the lock on the current or calling thread (Thread1 in the above example) and instructing it to sleep until another thread (Thread2 in the above example) enters the monitor and calls notify() or notifyAll(), or until a certain period of time has passed.

1. What is the use of notify?

This method() aids inter thread communication by allowing a single thread that is waiting for the object's monitor to become a runnable thread by waking it up. The decision of which waiting thread to awaken is arbitrary and is determined by the implementation.

**GLOSSARY:**

* **Multithreading:** This is a programming concept in which the application can create a small unit of tasks to execute in parallel. If you are working on a computer, it runs multiple applications and allocates processing power to them.
* **Synchronization :** The process that allows only one thread at a particular time to complete a given task entirely. By default, the JVM gives control to all the threads present in the system to access the shared resource, due to which the system approaches race condition.
* **NotifyAll() Method: A** similar way to the notify() method, the notifyAll() method aids inter thread communication. The only difference is that it wakes up all of the object monitor waiting for threads, making them all runnable at the same time. The term "runnable" does not imply that the thread is currently running. It signifies the thread can obtain a lock before proceeding to execution.
* **Legacy classes and interfaces:** These are the classes and interfaces that formed the collections framework in the earlier versions of Java and how now been restructured or re-engineered. They are fully compatible with the framework. All legacy classes and interface were redesign by JDK 5 to support Generics
* **Vector:** It is similar to a dynamic array whose size can be increased or decreased. Unlike arrays, it has no size limit and can store any number of elements. Since Java 1.2, it has been a part of the Java Collection framework. It's in the java.util package and implements the List interface, so we can use all of the List interface's methods here.