Synchronized keyword can be used only with methods and code blocks.

Suppose that two threads are working on SharedObj. If two threads run on different processors, each thread may have its own local copy of sharedVariable.

**Synchronized**: It means that only one thread or process can execute a block of code (critical section) at a time.

**Volatile**: It means that changes made by one thread to shared data are visible to other threads.

private int[] arr;  
private Object lockObject;  
private intcapacity = 10;  
private volatile intsize = 0;  
  
// Dining Philosophers Problem  
public App()  
{  
 arr= new int[capacity];  
 lockObject= new Object();  
}  
  
public void Insert(intvalue)  
{  
synchronized (lockObject)  
 {  
 if(size == capacity) return;  
 arr[size] = value;  
 size++;  
 }  
}  
  
public intGetElement()  
{  
synchronized (lockObject)  
 {  
if(size == 0) throw new IllegalStateException();  
 return arr[size--];  
}  
}

**General Differences between Wait and Sleep**

Simply put, wait() is an instance method that’s used for thread synchronization.

It can be called on any object, as it’s defined right on java.lang.Object, but it can only be called from a synchronized block. It releases the lock on the object so that another thread can jump in and acquire a lock.

On the other hand, Thread.sleep() is a static method that can be called from any context. Thread.sleep() pauses the current thread and does not release any locks.

Here’s a very simplistic initial look at these two core APIs in action:

Running this example will produce the following output:

Thread ‘main’ is woken after sleeping for 1 second

Object ‘java.lang.Object@31befd9f’ is woken after waiting for 1 second

private static Object LOCK = new Object();  
private static void sleepWaitExamples()  
throws InterruptedException {  
Thread.sleep(1000);  
System.out.println(  
"Thread '" + Thread.currentThread().getName() +  
"' is woken after sleeping for 1 second");  
synchronized (LOCK) {  
LOCK.wait(1000);  
System.out.println("Object '" + LOCK + "' is woken after" +  
" waiting for 1 second");  
}  
 }

**Default Methods In Java**

Before Java 8, interfaces could have only abstract methods. The implementation of these methods has to be provided in a separate class. So, if a new method is to be added in an interface then its implementation code has to be provided in the class implementing the same interface. To overcome this issue, Java 8 has introduced the concept of default methods that allow the interfaces to have methods with implementation without affecting the classes that implement the interface.

// A simple program to Test Interface default  
// methods in java  
interface TestInterface  
{  
// abstract method  
public void square(inta);  
  
// default method  
default void show()  
 {  
System.*out*.println("Default Method Executed");  
}  
}  
  
class TestClassimplements TestInterface  
{  
// implementation of square abstract method  
public void square(inta)  
 {  
System.*out*.println(a\*a);  
}  
  
public static void main(String args[])  
 {  
TestClass d = new TestClass();  
d.square(4);  
  
// default method executed  
d.show();  
}  
}

**Nested Classes**

Inner classes are a security mechanism in Java. We know a class cannot be associated with the access modifier private, but if we have the class as a member of other class, then the inner class can be made private. And this is also used to access the private members of a class.

classOuter\_Demo{

// private variable of the outer class

privateintnum=175;

// inner class

publicclassInner\_Demo {

public int getNum() {

System.out.println("This is the get num method of the inner class");

returnnum;

}

}

}

Public class My\_class2{

Public static void main(String args[]){

// Instantiating the outer class

Outer\_Demo outer =new Outer\_Demo();

// Instantiating the inner class

Outer\_Demo.Inner\_Demo inner =outer.new Inner\_Demo();

System.out.println(inner.getNum());

}

}

**Java – Generics**

Code Reuse: We can write a method/class/interface once and use for any type we want.

Type Safety: Generics also provide compile-time type safety that allows programmers to catch invalid types at compile time.Generics make errors to appear compile time than at run time (It’s always better to know problems in your code at compile time rather than making your code fail at run time). Suppose you want to create an ArrayList that store name of students and if by mistake programmer adds an integer object instead of string, compiler allows it. However, when we retrieve this data from ArrayList, it causes problems at runtime.

Public class GenericMethodTest{

// generic method printArray

publicstatic<T>voidprintArray( T[]inputArray){

// Display array elements

for(T element :inputArray){

System.out.printf("%s ", element);

}

}

publicstaticvoid main(Stringargs[]){

// Create arrays of Integer, Double and Character

Integer[]intArray={1,2,3,4,5};

Double[]doubleArray={1.1,2.2,3.3,4.4};

Character[]charArray={'H','E','L','L','O'};

System.out.println("Array integerArray contains:");

printArray(intArray);// pass an Integer array

System.out.println("\nArraydoubleArray contains:");

printArray(doubleArray);// pass a Double array

System.out.println("\nArraycharacterArray contains:");

printArray(charArray);// pass a Character array

}

}

**Generic Classes**

Public class Box<T>{

private T t;

public void add(T t){

this.t = t;

}

public T get(){

return t;

}

publicstaticvoid main(String[]args){

Box<Integer>integerBox=new Box<Integer>();

Box<String>stringBox=new Box<String>();

integerBox.add(new Integer(10));

stringBox.add(new String("Hello World"));

System.out.printf("Integer Value :%d\n\n",integerBox.get());

System.out.printf("String Value :%s\n",stringBox.get());

}

}

Integer Value :10

String Value :Hello World

**What are the differences between ArrayList and Vector?**

1. Vectors are synchronized, ArrayLists are not.
2. Data Growth Methods

\* Use ArrayLists if there is no specific requirement to use Vectors.

**Synchronization**

If multiple threads access an ArrayList concurrently then we must externally synchronize the block of code, which modifies the list either structurally or simply modifies an element. Structural modification means addition or deletion of element(s) from the list. Setting the value of an existing element is not a structural modification.

Collections.synchronizedList is normally used at the time of creation of the list to avoid any accidental unsynchronized access to the list.

**Data growth**

Internally, both the ArrayList and Vector hold onto their contents using an Array. When an element is inserted into an ArrayList or a Vector, the object will need to expand its internal array if it runs out of room. A Vector defaults to doubling the size of its array, while the ArrayList increases its array size by 50 percent.

**Difference between == and .equals() method in Java**

In general both equals() and “==” operator in Java are used to compare objects to check equality but here are some of the differences between the two:

Main difference between .equals() method and == operator is that one is method and other is operator.

We can use == operators for reference comparison (address comparison) and .equals() method for content comparison. In simple words, == checks if both objects point to the same memory location whereas .equals() evaluates to the comparison of values in the objects.

If a class does not override the equals method, then by default it uses equals(Object o) method of the closest parent class that has overridden this method. See this for detail

Coding Example:

// Java program to understand

// the concept of == operator

// Java program to understand   
// the concept of == operator  
public class Test {  
public static void main(String[] args)  
 {  
 String s1 = new String("HELLO");  
String s2 = new String("HELLO");  
System.*out*.println(s1 == s2);  
System.*out*.println(s1.equals(s2));  
}  
 }

Output:

false

true

Explanation: Here we are creating two objects namely s1 and s2.

Both s1 and s2 refers to different objects.

When we use == operator for s1 and s2 comparison then the result is false as both have different addresses in memory.

Using equals, the result is true because it is only comparing the values given in s1 and s2.

**Delete a node from Binary Search Tree**

classTreeNode  
{  
intvalue;  
TreeNodeleft;  
TreeNoderight;  
  
 public TreeNode(intv)  
 {  
value = v;  
}  
}  
public class DeleteBSTNode {  
// Case 1: No child  
 // => This is a leaf node. Delete it and return  
 // Case 2: One child  
 // => Change pointer  
 // => head = node.left || head = node.right  
 // Case 3: Two child  
 // => Find min in right  
 // => Copy the value in targeted node  
 // => Delete the duplicate from the right subtree  
  
public TreeNodedelete(TreeNode root, intdata)  
 {  
if(root == null) return null;  
 else if (root.value> data) root.left= delete(root.left, data);  
 else if (root.value< data) root.right= delete(root.right, data);  
 else  
{  
// We found the data...  
 // CASE 1 : No child  
if(root.left== null &&root.right== null) root = null;  
// CASE 2 : One child  
else if(root.left== null) root = root.right;  
// CASE 2 : One child  
else if(root.right== null) root = root.left;  
// CASE 3 : Two child  
else  
{  
TreeNode temp = findMin(root.right);  
root.value= temp.value;  
root.right= delete(root.right, temp.value);  
}  
 }  
  
return root;  
}  
}

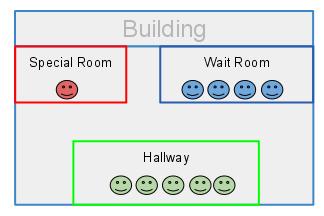
**Monitors – The Basic Idea of Java Synchronization**

A monitor is mechanism to control concurrent access to an object.

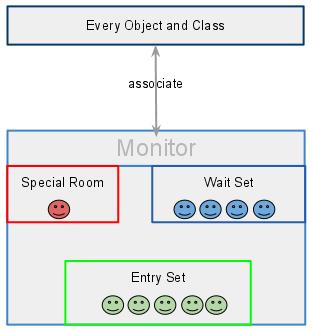
If you took operating system course in college, you might remember that monitor is an important concept of synchronization in operating systems. It is also used in Java synchronization. This post uses an analogy to explain the basic idea of "monitor".

1. What is a Monitor?

A monitor can be considered as a building, which contains a special room. The special room can be occupied by only one customer (thread) at a time. The room usually contains some data and code.



If a customer wants to occupy the special room, he has to enter the Hallway(Entry Set) to wait first. Scheduler will pick one based on some criteria(e.g. FIFO). If he is suspended for some reason, he will be sent to the wait room, and be scheduled to reenter the special room later. As it is shown in the diagram above, there are 3 rooms in this building.



In brief, a monitor is a facility, which monitors the threads' access to the special room. It ensures that only one thread can access the protected data or code.

When you wake up from wait you must test the condition predicate again, and go back to waiting (or fail) if it is not yet true. Since you can wake up repeatedly without your condition predicate being true, you must therefore always call wait from within a loop, testing the condition predicate in each iteration. The canonical form for a condition wait is shown below:

voidstateDependentMethod() throws InterruptedException {  
// condition predicate must be guarded by lock  
synchronized(lock) {  
while (!conditionPredicate())  
lock.wait();  
// object is now in desired state  
}  
}

**Double-checked locking**

In software engineering, double-checked locking (also known as "double-checked locking optimization") is a software design pattern used to reduce the overhead of acquiring a lock by first testing the locking criterion (the "lock hint") without actually acquiring the lock.

The only advantage might be performance: check in a non-thread-safe way, then do some locking operations to check the variable, which may be expensive.

public class Singleton {  
  
private static Singleton *singleton*= null;  
 private static Object *lock* = new Object();  
  
 private Singleton(){}  
  
public static Singleton getSingleton() {  
if(*singleton* == null) // 1st check  
synchronized (*lock*)  
 {  
if(*singleton* == null) // 2st check  
*singleton* = new Singleton();  
}  
  
return *singleton*;  
}  
}

**Dining Philosophers Problem**

****

In computer science, the dining philosophers problem is an example problem often used in concurrent algorithm design to illustrate synchronization issues and techniques for resolving them.

**Five silent philosophers sit at a round table with bowls of spaghetti. Forks are placed between each** pair of adjacent philosophers.

Each philosopher must alternately think and eat. However, a philosopher can only eat spaghetti when they have both left and right forks. Each fork can be held by only one philosopher and so a philosopher can use the fork only if it is not being used by another philosopher. After an individual philosopher finishes eating, they need to put down both forks so that the forks become available to others. A philosopher can take the fork on their right or the one on their left as they become available, but cannot start eating before getting both forks.

Eating is not limited by the remaining amounts of spaghetti or stomach space; an infinite supply and an infinite demand are assumed.

semaphore[] forks = {1,1,1,1,1};  
  
// INCORRECT: There is a circular wait condition  
void philosophers(inti)  
{  
while(true)  
 {  
 think();  
semWait(forks[i]);  
semWait(forks[(i+1) % 5]);  
eat();  
semSignal(forks[(i+1) % 5);  
semSignal(forks[i]);  
}  
}  
  
// CORRECT  
// Have a global ordering of the resources  
void philosophers(inti)  
{  
while(true)  
 {  
 think();  
semWait(forks[min(i, i+1) % 5)]);  
semWait(forks[max(i, i+1) % 5)]);  
eat();  
semSignal(forks[(i+1) % 5);  
semSignal(forks[i]);  
}  
}