Threads

22) Write code to define instantiate and start new threads using both java.lang.Thread and java.lang.Runnable

Creating Threads :
There are two ways

- Define a class that extends the Thread class

```
public class aThread extends Thread{
public void run() {
}
}
```

Every class that extends the Thread class should have a run() method that specifies whatever tasks are required when this thread is executed.
Now to start a new thread, you use the start() method, which is defined in the Thread class

Starting a Thread :

```
aThread t = new aThread().
t.start();
```

when the new class is instantiated and its start() method is called a new thread of control is created. The new thread then calls the run() method, which in this case is defined in aThread class.

- Define a class that implements the Runnable Interface.

```
public class aThread implements Runnable {
public void run() {

}
}
```

The following code would then create a thread and start it running:

```
aThread t = new aThread();
new Thread(t).start();
```

This method of creating threads of control is very useful if the class we want to run in a thread already extends some other class.

Remember : Creating a new Thread instance puts the thread into the "new thread" state. It is not "alive" until someone invokes the thread's start() method.

23) Recognize conditions that might prevent a thread from executing.

Every thread in the JVM can be in 4 states :

- Initial: From the period when a thread object is created (ie. when its constructor is called) until the thread object's start() method is called, is said to be the initial state.
- Runnable : once a thread's start() method is called, it is said to be in runnable state. The Runnable state can be thought of as a default state; if a thread is not in any other state, it's in the runnable state
- Blocked : A thread is blocked when it is waiting for some specific event to occur and it can't run. Threads that are sleeping or waiting on an object lock are also considered blocked.
- Exiting : Once thread's run() method returns or its stop() method is called, it is said to be in the exiting state.

Conditions that might prevent a thread from executing :

- The thread is not the highest priority thread and so cannot get CPU time.
- The thread has been put to sleep using the sleep() method
- There is more than one thread with the same highest priority and JVM is switching between these threads, at the moment, the thread in question is awaiting CPU time.
- Someone has suspended the thread using the suspend() method. (deprecated in Java 2)
- The thread is waiting on a condition because someone invoked wait() for the thread.
- The thread has explicitly yielded control by invoking yield()
- It is blocked for file I/O

---

24) Write code using synchronized wait notify and notifyAll to protect against concurrent access problems and to communicate between threads. Define the interaction between threads and object locks when executing synchronized wait notify or notifyAll.

Synchronization :

- Threads in Java are running in the same memory space, they can share access to variables and methods in objects. In the case of an object being shared between two threads, where, for example, one thread stores data into the shared object and the other thread reads that data, there can be problems of synchronization if the first thread hasn't finished storing the data before the second one goes to read it. So they need to be taken care to only access the data one at a time.
- Java's synchronization features serve to prevent multiple threads from performing conflicting tasks on the same object by indicating that a method is synchronized. A class may have several synchronized methods.
- When a class with synchronized methods is instantiated, the new object is given a monitor. Think of a monitor as a very small box that can hold only one thread. Once a thread enters a monitor, all other threads must wait until the first exits. These monitors can be prescribed to protect a shared asset from being manipulated by more than one thread at a time.
- Below is an example where three unsynchronized threads calling the same method at the same time.

```
class Callme {
void call(String msg) {
System.out.print("[" + msg);
try {
Thread.sleep(1000);
} catch (Exception e){};
System.out.println("]");
}
}

class caller implements Runnable {
String msg;
Callme target;
public caller(Callme t, String s) {
target = t;
msg = s;
new Thread(this).start();
}
public void run() {
target.call(msg);
}
}

public class Synch {
public static void main(String args[]) {
Callme target = new Callme();
new caller(target, "Hello");
new caller(target, "Synchronized");
new caller(target, "World");
}
}
```

output :

```
[Hello[Synchronized[World]
]
]
```

- You can see from the output that the sleep in the call method allowed the threads to context switch and mix up the output of our three message strings. The three threads call the same method, on the same object, all at the same time. This is known as a race−condition, because the three threads are racing each other to complete the method.
- The example used sleep to make the effects repeatable and obvious. In most situations, a race condition is subtle because you aren't sure where the context switch occurs, and the effects might be less visible than the example we've constructed here.
- When a synchronized keyword is applied to a method, it indicates that the entire method is a critical section.
- For a synchronized class method (a static method), Java obtains an exclusive lock on the class before executing the method. For a synchronized instance method, Java obtains an exclusive lock on the class instance.
- The Callme class above can be fixed with the synchronized modifier used in the call method to force the other threads to wait until the first one has completed.

```
class Callme {
synchronized void call(String msg) {
System.out.print("[" + msg);
try Thread.sleep(1000);
catch (Exception e);
System.out.println("]");
}
}

class caller implements Runnable {
String msg;
Callme target;
public caller(Callme t, String s) {
target = t;
msg = s;
new Thread(this).start();
}
public void run() {
target.call(msg);
}
}

class Synch1 {
public static void main(String args[]) {
Callme target = new Callme();
new caller(target, "Hello");
new caller(target, "Synchronized");
new caller(target, "World");
}
}
```

output :

```
[Hello]
[Synchronized]
[World]
```

- You can wrap the call to the methods in a synchronized block.

```
synchronized (expression) statement;
```

- Here, expression is an expression that must resolve to an object or array, and statment is the code of the critical section, which is usually a block of statements (within { and }).
- The synchronized statement attempts to acquire an exclusive lock for the object or array specified by expression.
- It does not execute the critical section of code until it can obtain this lock, and in this way, ensures that no other threads can be executing the section at the same time.
- We can fix the previous example by re−coding the run method.

```
public void run() {
synchronized(target) {
target.call(msg);
}
}
```

Inter−thread communication :

- Java includes an elegant inter−process communication mechanism via the wait, notify, and notifyAll methods. These methods are implemented as final methods in Object, so all classes have them.
- wait/notify is used when synchronized methods in the same class need to communicate with each other.
- *wait:* tells the current thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls notify.

- *notify:* Wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation.
- *notifyAll:* wakes up all the threads that call wait on the same object. The highest priority thread that wakes up will run first.

A non−synchronous Producer/Consumer problem :

```
class Q {
int n;
synchronized int get() {
System.out.println("Got: " + n);
return n;
}
synchronized void put(int n) {
this.n = n;
System.out.println("Put: " + n);
}
}

class Producer implements Runnable {
Q q;
Producer(Q q) {
this.q = q;
new Thread(this, "Producer").start();
}
public void run() {
int i = 0;
while(true) {
q.put(i++);
}
}
}

class Consumer implements Runnable {
Q q;
Consumer(Q q) {
this.q = q;
new Thread(this, "Consumer").start();
}
public void run() {
while(true) {
q.get();
}
}
}

class PC {
public static void main(String args[]) {
Q q = new Q();
new Producer(q);
new Consumer(q);
}
}
```

- Although the put and get methods on Q are synchronized, there's still nothing stopping the producer from overrunning the consumer, nor is there anything to stop the consumer from consuming the same queue value twice.

output :

```
Put: 1
Got: 1
Got: 1
Got: 1
Got: 1
Got: 1
Put: 2
Put: 3
Put: 4
Put: 5
Put: 6
Put: 7
Got: 7
```

- The polling way to fix this would be to have the producer set some boolean variable after it had set the value, then spin in a tight loop evaluating the same boolean, waiting for the consumer to set it back once the value had been consumed.

```
class Q {
int n;
boolean valueSet = false;
int get() {
while (!valueSet)
;
System.out.println("Got: " + n);
valueSet = false;
return n;
}
void put(int n) {
while (valueSet)
;
this.n = n;
valueSet = true;
System.out.println("Put: " + n);
}
}
```

- This version gives the correct output but the two while loops consume enormous amount of CPU resources to accomplish almost nothing, embarrassingly inefficient!

Properly synchronized the queue via wait and notify :

```
class Q {
int n;
boolean valueSet = false;
synchronized int get() {
if (!valueSet) try {
wait();
} catch(InterruptedException e) {};
System.out.println("Got: " + n);
valueSet = false;
notify();
return n;
}
synchronized void put(int n) {
if (valueSet) try {
wait();
} catch(InterruptedException e) {};
this.n = n;
valueSet = true;
System.out.println("Put: " + n);
notify();
}
}
```

ouput :

```
Put: 1
Got: 1
Put: 2
Got: 2
Put: 3
Got: 3
Put: 4
Got: 4
Put: 5
Got: 5
```

Deadlock :

- Deadlock is a rare, but very hard to debug error condition where two threads have a circular dependency on a pair of synchronized objects.
- For example, if one thread enters the monitor on object X and another thread enters the monitor on object Y, then if X tries to call any synchronized method on Y it will block as expected.
- However, if Y in turn tries to call any synchronized method on X, then it will be waiting forever since in order to get the lock on X, it would have to release its own lock on Y first so that the first thread could complete.
- In the example of thread deadlock below, the main class named Deadlock creates an A and a B instance, then forks a second thread to set up the deadlock condition. The methods foo and bar use sleep as a way to force the deadlock condition to occur. Under real circumstances, it could be much harder to find in your code.

```
class A {
synchronized void foo(B b) {
String name = Thread.currentThread().getName();
System.out.println(name + " entered A.foo");
try {Thread.sleep(1000);} catch (Exception e) {};
System.out.println(name + " trying to call B.last()");
b.last();
}
synchronized void last() {
System.out.println("inside A.last");
}
}

class B {
synchronized 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(name + " trying to call A.last()");
a.last();
}
synchronized void last() {
System.out.println("inside B.last");
}
}

class Deadlock implements Runnable {
A a = new A();
B b = new B();
Deadlock() {
Thread.currentThread().setName("MainThread");
new Thread(this, "RacingThread").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();
}
}
```

<u>ouput :</u>

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

Full Thread Dump:
"RacingThread" (... state:MONITOR_WAIT) prio=5
A.last(Deadlock.java:8)
```

```
B.bar(Deadlock.java:17)
Deadlock:run(Deadlock.java:32)
java.lang.Thread.run(Thread.java:289)
"MainThread" (... state:MONITOR_WAIT) prio=5
B.last(Deadlock.java:19)
A.foo(Deadlock.java:6)
Deadlock.(Deadlock.java:28)
Deadlock,main(Deadlock.java:36)
Monitor Cache Dump:
A@1393760/142E330 (key=0x1393760): "MainThread"
B@1393778/142E340 (key=0x1393778): "RacingThread"
```

Thread Summary :

The interface Runnable is defined as,

```
public abstract interface Runnable {
// Public Instance Methods
public abstract void run();
}
```

The class Thread in Java is defined as,

```
public class Thread extends Object implements Runnable {
// Public Constructors
public Thread();
public Thread(Runnable target)
public Thread(ThreadGroup group, Runnable target);
public Thread(String name);
public Thread(ThreadGroup group, String name);
public Thread(Runnable target, String name);
public Thread(ThreadGroup group, Runnable target, String name);
// Constants
public final static int MAX_PRIORITY;
public final static int MIN_PRIORITY;
public final static int NORM_PRIORITY;
// Class Methods
public static int activeCount();
public static Thread currentThread();
public static sleep(long millis) throws InterruptedException;
public static sleep(long millis, int nanos) throws InterruptedException;
public static void yield();
....
....
// Public Instance Methods
public void destroy();
public final int getPriority();
public final ThreadGroup getThreadGroup();
public void boolean isAlive();
public void run();
public void setPriority(int newPriority) throws IllegalArgumentException;
public final void stop();
public synchronized void stop(Throwable o);
public final void suspend();
.....
.....
}
```

- *Class Methods :* These are static methods which shoild be called directly on the Thread class
- *currentThread :* The currentThread static method returns the Thread object which is the currently running thread.
- *yield :* The yield method causes the runtime to context switch from the current thread to the next available runnable thread. This is one way to ensure that threads at lower priority do not get starved.
- *sleep (int n) :* The sleep method causes the runtime to put the current thread to sleep for n milliseconds. After n milliseconds have expired, this thread will become eligible to run again. The clocks on most Java runtimes will not be able to be accurate to less than 10 milliseconds.
- *Instance Methods :*
- *start :* The start method tells the Java runtime to thread. This is the single method in the Runnable nterface. It is called by the start method after the proper system thread has been initialized. Whenever the run method returns, the current thread will stop.
- *stop :(deprecated)* The stop method causes this thread to stop immediately. This is often an abrupt way to end a thread, especially if this method is executed on the current thread. In this case, the line immediately after the stop method call is never executed because the thread context dies before stop returns. A cleaner way to stop a thread is to set some variable which will cause the run method to exit cleanly.
- *suspend :(deprecated)* The suspend method is different from stop. Suspend takes the thread and causes it to stop running without destroying the underlying system thread, or the state of the formerly running thread. If a thread is suspended, you can call resume on the same thread to cause it to start running again.
- *resume :(deprecated)* The resume methods is used to revive a suspended thread. There is no guarantee that the thread will start running right away, since there might be a higher−priority thread running already, but resume causes the thread to become eligible for running.
- setPriority (int p) The setPriority method sets the thread's priority to the integer value passed in. There are several predefined priority constants defined in class
- Thread:MIN_PRIORITY, NORM_PRIORITY and MAX_PRIORITY, which are 1, 5, and 10 respectively.
- Most user−level processes should stick to NORM_PRIORITY, plus or minus 1.
- Background tasks like network I/O or screen repainting should be set at or near MIN_PRIORITY. Use caution when starting threads at MAX_PRIORITY. If they don't sleep or yield, you may find the entire Java runtime unresponsive.
- *getPriority :* The get Priority method returns the thread's current priority, a value between one and ten.
- *setName (String name) :* The setName method identifies the thread with a human−readable name. This helps with debugging multi−threaded programs. This name will appear in all stack traces that display when the interpreter prints uncaught exceptions.
- *getName :* The getName method returns the current String value of the thread's name as set by setName.
- *destroy :* Stops and kills a thread. Java runtime cannot have chance to intervene. Not recommended for routine use.
- *interrupt :* Causes any kind of wait (sleep, wait, join, etc.) to abort with an InterruptedException.
- *wait :* Causes a thread to suspend until a later notifyAll or notify is made
- *notifyAll :* Wakes up all suspended activities that have been blocked via waits within any method of the same object
- *notify :* Wakes up at most one waiting thread

- *join :* Let an object wait for a thread to complete at a later time Or, Thread.join(long msec) can be used to obtain calls with time−out

Remember :

- When wait() is called , the Thread will block for x time or until notify() is called for the Thread. notifyAll() will call notify() for each object waiting for the object lock.
- yield() and sleep() are static methods. Affects only the currently running thread