join() in Java

java.lang.Thread class provides the **join() method which allows one thread to wait until another thread completes its execution**. If **t** is a Thread object whose thread is currently executing, then **t.join()** will make sure that **t** is terminated before the next instruction is executed by the program.

If there are multiple threads calling the join() methods that means overloading on join allows the programmer to specify a waiting period. However, as with sleep, join is dependent on the OS for timing, so **you should not assume that join will wait exactly as long as you specify.**

Let's see one example:


```
class Demo1 extends Thread
{
    public void run()
    {
        for(int i=1;i<=5;i++)
        {
            try
            {
                System.out.println("JAVA");
                Thread.sleep(2000);
            }
            catch (Exception e)
            {
                System.out.println("Some problem occured.");
            }
        }
    }
}
class Demo
{
    public static void main(String[] args) throws InterruptedException
    {
        System.out.println("Main thread started");
        Demo1 d1 = new Demo1();
        d1.start();
        d1.join();//d1 thread is making main thread to wait until it completes exceution
        System.out.println("Main thread completed");
    }
}
```

Output:

```
Main thread started
JAVA
JAVA
JAVA
JAVA
JAVA
Main thread completed
Press any key to continue . . .
```

In the above output we see that main thread is waiting for d1 thread to complete execution.

There are three overloaded join functions.

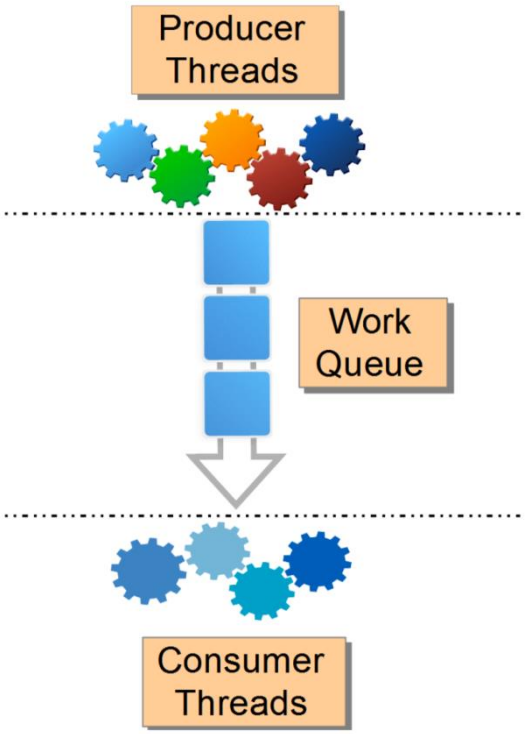
- **join():** It will put the current thread on wait until the thread on which it is called is dead. If thread is interrupted then it will throw InterruptedException.
- **join(long millis)** :It will put the current thread on wait until the thread on which it is called is dead or wait for specified time (milliseconds).
- **join(long millis, int nanos):** It will put the current thread on wait until the thread on which it is called is dead or wait for specified time (milliseconds + nanos).

understood

Producer-Consumer problem in java

In computing, **the producer-consumer problem (also known as the bounded-buffer problem) is a classic example of a multi-process synchronization problem**. The problem describes two processes, the producer and the consumer, which **share a common, fixed-size buffer used as a queue.**

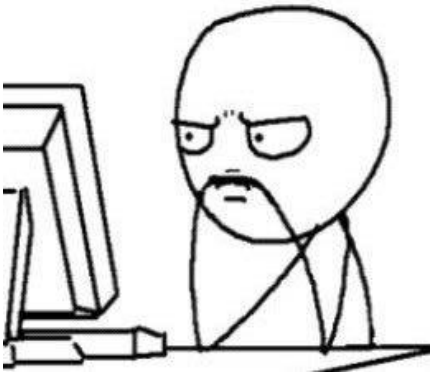
- The producer's job is to generate data, put it into the buffer, and start again.
- At the same time, the consumer is consuming the data (i.e. removing it from the buffer), one piece at a time.



Let us see a case where producer is producing n number of values and Consumer is consuming the values produced.

```
class Queue
{
    int x;
    void store(int j)
    {
        x = j;
        System.out.println("Produced"+x);
    }
    void retrieve()
    {
        System.out.println("Consumed"+x);
    }
}
```

```
class Producer extends Thread
{
    Queue a; // refrence to queue class which is pointing to queue object
    Producer (Queue q)//constructor taking queue type reference as input
    {
        a = q;
    }
    public void run()
    {
        int i=1;
        for(;;)
        {
            a.store(i++);
        }
    }
}
class Consumer extends Thread
{
    Queue b;
    Consumer(Queue q)
    {
        b = q;
    }
    public void run()
    {
        for(;;)
        {
            b.retrieve();
        }
    }
}
```

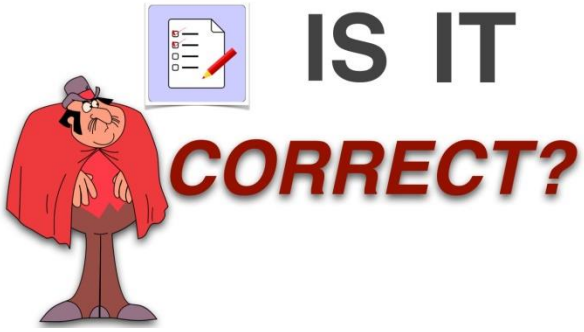


```
class Demo
{
    public static void main(String[] args)
    {
        Queue q = new Queue();
        Producer p = new Producer(q);
        Consumer c = new Consumer(q);
        p.start();
        c.start();
    }
}
```

Output:

We see in the below output that the process is so fast that sometimes we see a value getting consumed before the produced value is printed. And also all the values that are getting produced are not getting consumed.

```
Produced1921
Consumed1921
Consumed1922
Consumed1922
Consumed1922
Consumed1922
Produced1922
Consumed1922
Produced1923
Produced1924
Produced1925
Produced1926
Produced1927
Consumed1923
Consumed1928
Consumed1928
Terminate batch job (Y/N)?
```



The above code is partially correct let us see how to overcome the problems in the below example:

In the above code you will have to make changes in class Queue, so let's see what are those changes,

```
class Queue
{
    int x;
    boolean is_data_present=false;
    synchronized void store(int j)
    {
        try
        {
            if(is_data_present==false)
            {
                x = j;
                System.out.println("Produced"+x);
                is_data_present = true;
                notify();
            }
            else
            {
                wait();
            }
        }
        catch (Exception e)
        {
            System.out.println("Some problem occured");
        }
    }
```

```
    synchronized void retrieve()
    {
        try
        {
            if(is_data_present==true)
            {
                System.out.println("Consumed"+x);
                is_data_present = false;
                notify();
            }
            else
            {
                wait();
            }
        }
        catch (Exception e)
        {
            System.out.println("Some problem occured");
        }
    }
}
```

Let us now see the output for the same code after making the above changes:

```
Produced2567
Consumed2567
Produced2569
Consumed2569
Produced2571
Consumed2571
Produced2573
Consumed2573
Produced2575
Consumed2575
Produced2577
Consumed2577
Produced2579
Consumed2579
Produced2581
Consumed2581
Produced2583
Consumed2583
Produced2585
Terminate batch job (Y/N)?
```

We can see that only after the value is produced, consumer is able to consume. And because of the fast execution some values are not printing but that doesn't mean they are not produced.