D Y PATIL UNIVERSITY
NAVI MUMBAI

Subject Name : Java Programming

Unit No: 4

Unit Name : Java Input / Output & Multithreading

Faculty Name : Dr. Ritu Jain

D Y PATIL UNIVERSITY
NAVI MUMBAI

Unit No.: 4

Faculty Name :

Unit No:4

Lecture No: 1

Multithreading

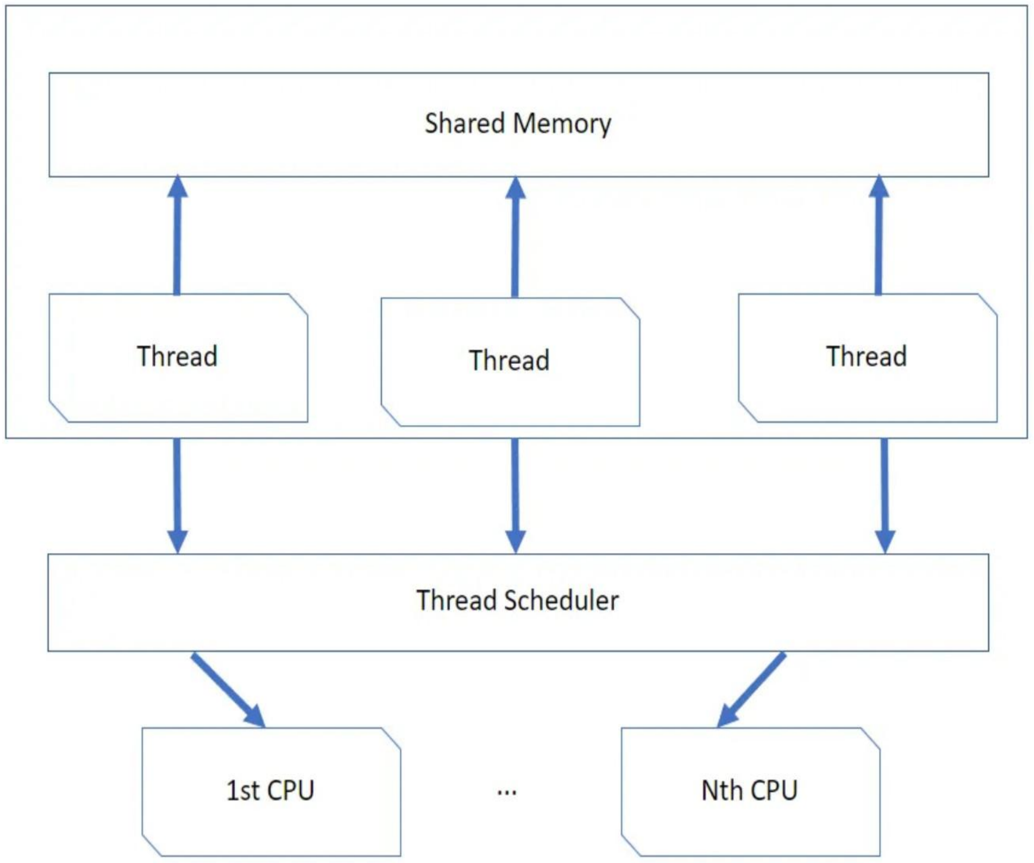
- Multithreading in Java is a programming concept that allows multiple parts of a program, known as threads, to execute concurrently within a single process.
- Each thread represents an independent path of execution.
- Single threaded process: one that contain exactly one thread.
- Multi threaded process: one that contains more than one threads.
- A process is a group of associated threads that execute in the same, shared environment.
- By shared environment, we mean that the threads in the same process share the same memory space and can communicate directly with one another.
- Thread is the smallest unit of execution that can be scheduled by Operating system.
- Single unit of work which is executed by thread is called as task. Even though a thread can complete multiple tasks, it can execute only one task a time.

D Y PATIL UNIVERSITY NAVI MUMBAI

Multithreading

- Threads share the same address space. They are lightweight and cost of communication between the threads are low.
- The Java Virtual Machine allows an application to have multiple threads of execution running concurrently.
- A thread is a separate path of execution of code for each task within a program. In thread- based multitasking, a program uses multiple threads to perform one or more tasks at the same time by a processor.
- That is, thread-based multitasking feature allows you to execute several parts of the same program at once. Each thread has a different path of execution.

Process Model



Process model

Realtime Example of Multithreading in Java

- Consider you are using Microsoft Word on your PC/laptop. In this application, say you are writing an article. So, Microsoft Word is itself a process because it is a program under execution. In Microsoft Word only, there are multiple threads that are running together. For instance, you are typing and the typed content is being shown on the screen. This can be considered as a thread. Another thread can be spell check while you are typing. Auto-saving is another thread running simultaneously.
- Another familiar example is a browser that starts rendering a web page while it is still downloading the rest of page.
- So, this is what threads are. Threads are the smallest parts or units of processes and threads combine to form a process

Realtime Example of Multithreading in Java

- Consider a program as shown in the below figure. The program is divided into two parts. These parts may represent two separate blocks of code or two separate methods that can perform two different tasks of the same program.
- Hence, a processor will create two separate threads to execute these two parts simultaneously. Each thread acts as an individual process that will execute a separate block of code.
- Thus, a processor has two threads that will perform two different tasks at a time. This multitasking is called thread-based multiple tasking.
-

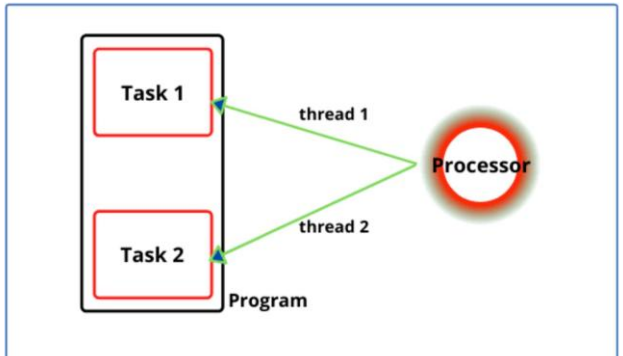


Fig: Thread-based Multitasking

What is the need for Multithreading?

- **Advantages:**
- **Improved CPU Utilization:** Threads can utilize available CPU cores more effectively by running different tasks in parallel.
- **Enhanced Responsiveness:** Applications can remain responsive to user input while performing background tasks.
- **Better Resource Utilization:** Resources like memory and I/O can be used more efficiently.
- **Simplified Problem Solving:** Certain problems, such as handling multiple client requests in a server application, are naturally suited for a multithreaded approach.

Life cycle of a thread

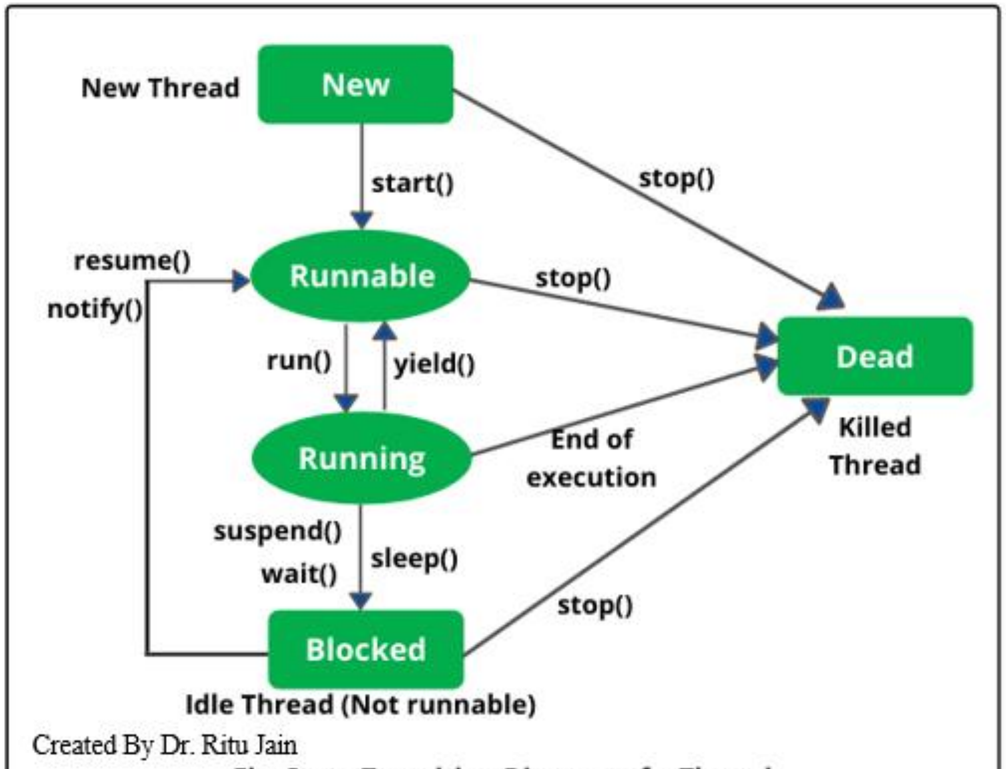


Fig: State Transition Diagram of a Thread

Life cycle of a thread: Newborn state

- **Newborn**: When we create a thread object using the Thread class, the thread is born and enters the New state. This means the thread is created, but the start() method has not yet been called on the instance.
- In this state, the Thread object exists but it cannot execute any statement because the execution of thread has not started. Only after calling the start() method does the thread move to the Runnable state. In other words, the thread remains in this state until you start its execution by calling the start() method.
- System.out.println("Thread state before start: " + thread.getState());

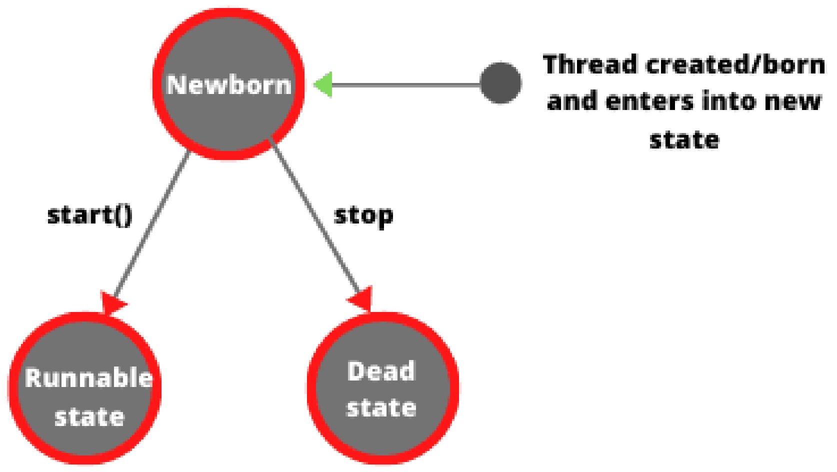


Fig: Scheduling a newborn thread

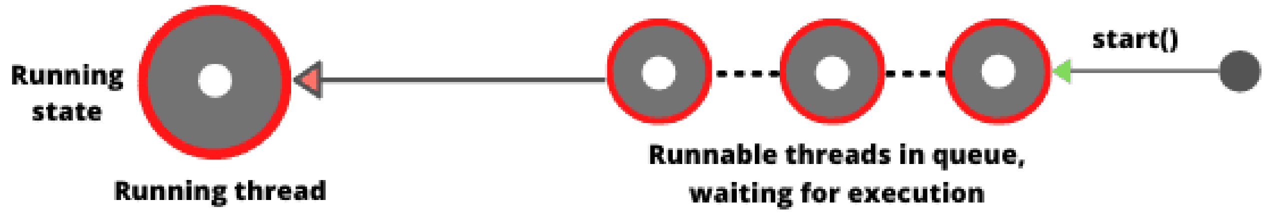


Fig: Runnable and Running states

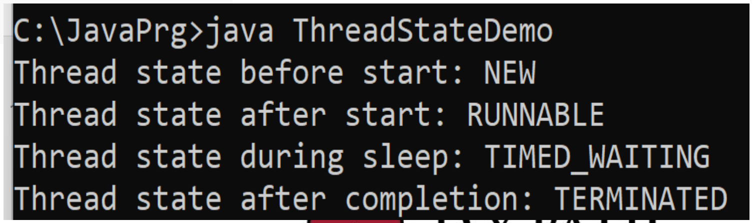
Life cycle of a thread: Runnable state

- **Runnable** : The Runnable state means that a thread is ready to run and is awaiting for the control of the processor, or in other words, threads are in this state in a queue and wait their turns to be executed.

- If all the threads have equal priority, then they are given time slots for execution in round robin fashion, i.e. first come, first serve manner. The thread that relinquishes control joins the queue at the end and again waits for its turn. This process of assigning time to threads is known as *time slicing*.

However, if we want a thread to relinquish control to another thread, we can do so by using yield() method. The yield() method of thread class causes the currently executing thread object to move to Runnable state and allow other threads to execute.
If any thread executes the yield method, the thread scheduler checks if there is any thread with the same or high priority as this thread. If the processor finds any thread with higher or same priority then it will move the current thread to Ready/Runnable state and give the processor to another thread and if not – the current thread will keep executing.

```java
class MyThread extends Thread {
    public void run() {
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

public class ThreadStateDemo {
    public static void main(String[] args) throws Exception {
        MyThread thread = new MyThread();
        // NEW state
        System.out.println("Thread state before start: " + thread.getState());
        thread.start();
        // RUNNABLE state
        System.out.println("Thread state after start: " + thread.getState());
        Thread.sleep(10);
        // Likely TIMED_WAITING or RUNNABLE
        System.out.println("Thread state during sleep: " + thread.getState());
        thread.join();
        // TERMINATED state
        System.out.println("Thread state after completion: " + thread.getState());
    }
}
```



```
C:\JavaPrg>java ThreadStateDemo
Thread state before start: NEW
Thread state after start: RUNNABLE
Thread state during sleep: TIMED_WAITING
Thread state after completion: TERMINATED
```

Life cycle of a thread: Running state

- **Running** : Running means that the thread has control of the processor, its code is currently being executed and thread will continue in this state until it get preempted by a higher priority thread, or until it relinquishes control on its own.
- A running thread may relinquishes its control in one of the following situations:
  1. When sleep() method is invoked on a thread to sleep for specified time period, the thread is out of queue during this time period. The thread again reenters into the runnable state as soon as this time period is elapsed.
  2. When a thread is suspended using suspend() method for some time in order to satisfy some conditions. A suspended thread can be revived by using resume() method. **Deprecated**
  3. When wait() method is called on a thread to wait for some time. The thread in wait state can be run again using notify() or notifyAll() method.

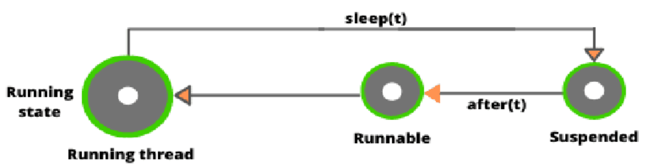


Fig: Relinquishing control using sleep() method

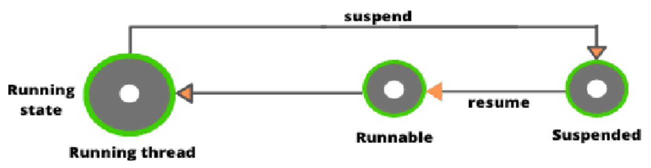


Fig: Relinquishing control using suspend() method

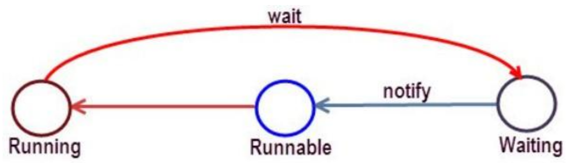


Fig. Control with wait() method

Life cycle of a thread

- **Blocked** : A thread is Blocked means that it is being prevented from the Runnable (or Running) state and is waiting for some event in order for it to reenter the scheduling queue.

Dead : A thread is Dead when it finishes its execution or is stopped (killed) by another thread. stop() method is deprecated

D Y PATIL UNIVERSITY NAVI MUMBAI

Unit No:4

Lecture No: 2

Constructors of Thread class

- **public Thread():** Allocates a new Thread object. Automatically generated names are of the form "Thread-"+n, where n is an integer.
- **public Thread(Runnable target):** Allocates a new Thread object. Target is the object whose run method is invoked when this thread is started.
- **public Thread(String name):** Allocates a new Thread object. Parameters: name is the name of the new thread.
- **public Thread(ThreadGroup group, Runnable task, String name):** Allocates a new Thread object so that it has target as its run object, has the specified name as its name, and belongs to the thread group referred to by group.

Methods of Threadclass

- **static Thread currentThread():** Returns a reference to the currently executing thread object.
- **long getId()**: Returns the identifier of this Thread. The thread ID is a positive long number generated when this thread was created. The thread ID is unique and remains unchanged during its lifetime. When a thread is terminated, this thread ID may be reused.
- **String getName():** Returns this thread's name.
- **final void setName(String name):** Changes the name of this thread to be equal to the argument name.
- **public static void sleep (long millis) throws InterruptedException:** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
- **final int getPriority():** Returns this thread's priority. (MIN_PRIORITY: 1, MAX_PRIORITY: 10, NORM_PRIORITY: 5)

D Y PATIL UNIVERSITY NAVI MUMBAI

Methods of Threadclass (cont'd)

- **public void start():** Causes this thread to begin execution; the Java Virtual Machine calls the run method of this thread.
- **final boolean isAlive():** Tests if this thread is alive. A thread is alive if it has been started and has not yet died. Returns: true if this thread is alive; false otherwise.
- **public String toString():** Returns a string representation of this thread, including the thread's name, priority, and thread group.

Main Thread

- When a Java program starts up, one thread begins running immediately. This is usually called the *main thread* of your program, because it is the one that is executed when your program begins.
- Although the main thread is created automatically when your program is started, it can be controlled through a **Thread** object. To do so, you must obtain a reference to it by calling the method **currentThread(**
  ), which is a **public static** member of **Thread**. Its general form is shown here:

  ```
  static Thread currentThread( )
  ```

- This method returns a reference to the thread in which it is called. Once you have a reference to the main thread, you can control it just like any other thread.

Main Thread

```
public class Ex1CurrentThread {
        public static void main(String[] args) { Thread
                t=Thread.currentThread(); System.out.println("Current
                thread= "+t);
                System.out.println("It's name= "+t.getName());
        }
}
```

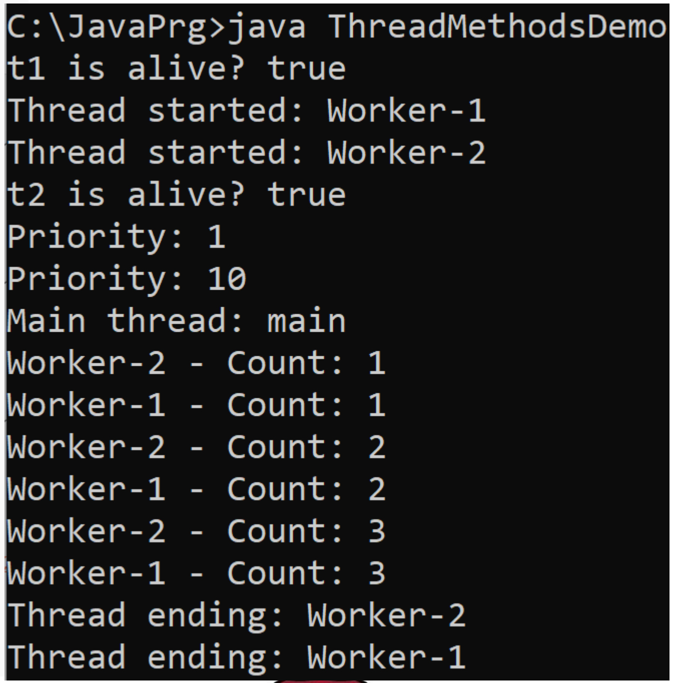
Output:

```
Current thread= Thread[main,5,main] It's name= main
```

Explanation:

[main,5,main]: name of the thread, its priority, and the name of its group.

```java
class MyThread extends Thread {
    public void run() {
        try {
            // Display thread running information
            System.out.println("Thread started: " + Thread.currentThread().getName());
            System.out.println("Priority: " + Thread.currentThread().getPriority());
            // Make thread sleep
            for (int i = 1; i <= 3; i++) {
                System.out.println(Thread.currentThread().getName() + " - Count: " + i);
                Thread.sleep(1000);  // sleep for 1 second
            }
            System.out.println("Thread ending: " + Thread.currentThread().getName());
        } catch (InterruptedException e) {
            e.printStackTrace();
        }}}

public class ThreadMethodsDemo {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        MyThread t2 = new MyThread();
        // Set thread names
        t1.setName("Worker-1");
        t2.setName("Worker-2");
        // Set priorities
        t1.setPriority(Thread.MIN_PRIORITY);  // 1
        t2.setPriority(Thread.MAX_PRIORITY);  // 10
        t1.start();
        t2.start();
        // Check if threads are alive
        System.out.println("t1 is alive? " + t1.isAlive());
        System.out.println("t2 is alive? " + t2.isAlive());
        // Main thread info
        System.out.println("Main thread: " + Thread.currentThread().getName());
    }
}
```



```
C:\JavaPrg>java ThreadMethodsDemo
t1 is alive? true
Thread started: Worker-1
Thread started: Worker-2
t2 is alive? true
Priority: 1
Priority: 10
Main thread: main
Worker-2 - Count: 1
Worker-1 - Count: 1
Worker-2 - Count: 2
Worker-1 - Count: 2
Worker-2 - Count: 3
Worker-1 - Count: 3
Thread ending: Worker-2
Thread ending: Worker-1
```

Unit No:4

Lecture No: 3

Creation of Threads

- Java’s multithreading system is built upon the **Thread** class, its methods, and its companion interface, **Runnable:**
  - **public class Thread extends Object implements Runnable**
- Threads can be created by using two mechanisms :
  1. Extending the Thread class
  2. Implementing the Runnable Interface

Creating Thread

- **One is to declare a class to be a subclass of `Thread`.** This subclass should override
  the `run` method of class `Thread`. An instance of the subclass can then be allocated and started. For example:

```
class PrimeThread extends Thread {

        public void run() {
             . . .
        }
    }
```

In main method, following code would then create a thread and start it running:

```
    PrimeThread p = new
    PrimeThread(); p.start();
```

-

Steps for Thread creation by extending the Thread class

1. Create a class that extends the **java.lang.Thread** class. This class overrides the run() method available in the Thread class.
2. A thread begins its life with run() method. It is the entry point for new thread.
3. We create an object of our new class and call start() method to start the execution of a thread.
4. start() invokes the run() method on the Thread object.

Creating Thread by extending Thread class

```java
class NewThread1 extends Thread{
    NewThread1(){
      super("Demo thread");
    }
    public void run(){
        System.out.println("Child thread "+
        Thread.currentThread());
    }
}
public class Ex2ThreadCreateThruExtends1 {
    public static void main(String[] args) {
        NewThread1 nt=new NewThread1();
        nt.start();
        System.out.println("Main thread "+
        Thread.currentThread());
    }
}
```

Output(Order may vary):

```
Child thread Thread[Demo thread,5,main]
Main thread Thread[main,5,main]
```

Ex: Creating thread by extending Thread class

```
class Class1Ex1 extends Thread {
    public void run(){

        try{
        for(int i=0;i<5;i++){
            System.out.println("Class1  "+i);
            Thread.sleep(500);
        }
    }catch(InterruptedException e){
        System.out.println("Class1 thread interrupted");
    }
 }
}
class Class2Ex1 extends Thread {
    public void run(){

        try{
         for(int i=0;i<5;i++){
            System.out.println("Class2  "+i);
            Thread.sleep(500);
        }
    }catch(InterruptedException e){
        System.out.println("Class2 thread interrupted");
    }
 }
}
public class MultiThrEx1CreateThr {
    public static void main(String[] args) {
        Class1Ex1 obj1=new Class1Ex1();
        Class2Ex1 obj2=new Class2Ex1();
        obj1.start();
        obj2.start();
}
}
```

Output: (Your output may vary)

```
Class1 0
Class2 0
Class2 1
Class1 1
Class1 2
Class2 2
Class1 3
Class2 3
Class2 4
Class1 4
```

Creating Thread by implementing Runnable

- The other way to **create a thread is to declare a class that implements Runnable interface**. That class then implements the run method. An instance of the class can then be allocated, passed as an argument when creating Thread, and started. The same example in this other style looks like the following:

```
class PrimeRun implements Runnable {

    public void run() {
        . . .
  } }
```

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

```
PrimeRun p = new PrimeRun();
new Thread(p).start();
```

Every thread has a name for identification purposes. More than one thread may have the same name. If a name is not specified when a thread is created, a new name is generated for it.

Steps for Thread creation by implementing the Runnable Interface

- We create a new class which implements java.lang.Runnable interface and override run() method.
- Then we instantiate a Thread object and call start() method on this object.

Ex: Thread creation by implementing the Runnable Interface

```
class NewThreadRunn implements Runnable{
   public void run(){
      System.out.println("Current thread "+Thread.currentThread());
   }
}
public class Ex4ThreadCreateRunnnable {
   public static void main(String[] args) {

      Thread th1=new Thread(new NewThreadRunn());
      th1.start();
      System.out.println("Main thread "+ Thread.currentThread());
   }
}
```

Output:

```
Current thread Thread[Thread-0,5,main] Main
thread Thread[main,5,main]
```

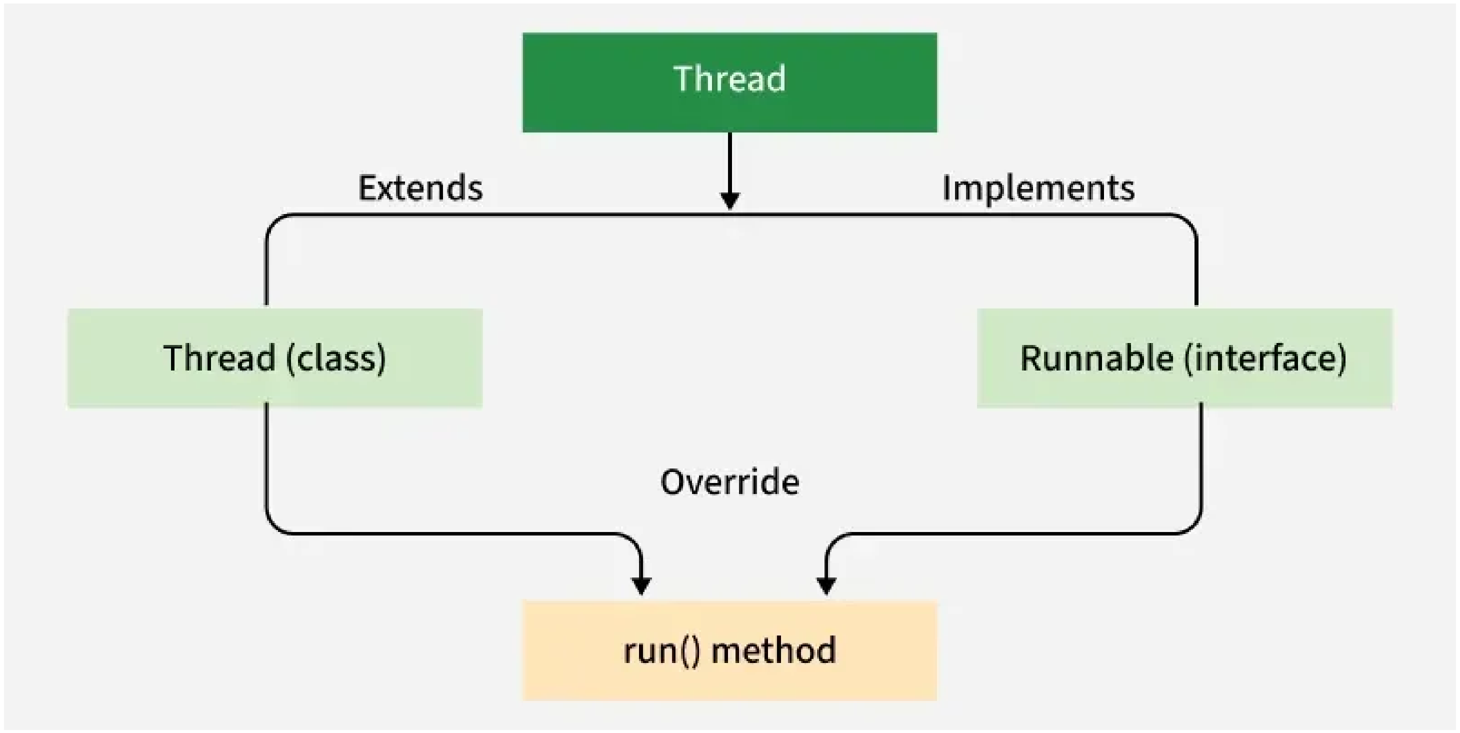
Ex: Thread creation by implementing the Runnable Interface

```
class Class1Ex2 implements Runnable {
    public void run(){

        try{
        for(int i=0;i<5;i++){
            System.out.println("Class1 "+i);
            Thread.sleep(500);
        }
    }catch(InterruptedException e){
        System.out.println("Class1 thread interrupted");
    }
 }
}
class Class2Ex2 implements Runnable {
    public void run(){

        try{
         for(int i=0;i<5;i++){
            System.out.println("Class2 "+i);
            Thread.sleep(500);
        }
    }catch(InterruptedException e){
        System.out.println("Class2 thread interrupted");
    }
 }
}
public class MultiThrEx2CreateThr {
    public static void main(String[] args) {
        Class1Ex2 obj1=new Class1Ex2();
        Class2Ex2 obj2=new Class2Ex2();
        Thread t1= new Thread(obj1);
        Thread t2= new Thread(obj2);
        t1.start();
        t2.start();
}
}
```

Output:

```
Class2 0
Class1 0
Class2 1
Class1 1
Class2 2
Class1 2
Class1 3
Class2 3
Class2 4
Class1 4
```

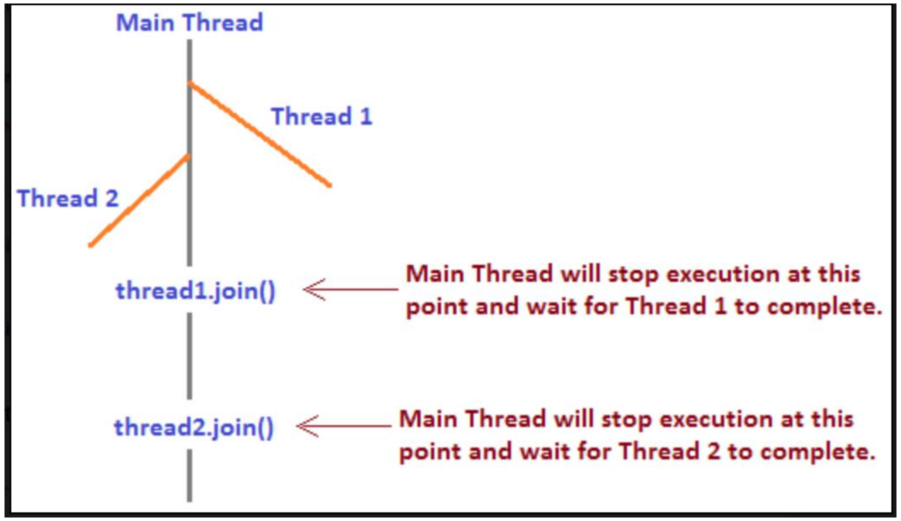
Two ways to Create a Thread



Unit No:4

Lecture No: 4

join() method of Thread class

- **java.lang.Thread** class provides the join() method which allows one thread to wait until another thread completes its execution.
- If **thread1** is a Thread object whose thread is currently executing, then **thread1.join()** will make sure that **thread1** is terminated before the next instruction of main thread is executed by the program.
- If **thread2** is a Thread object whose thread is currently executing, then **thread2.join()** will make sure that **thread2** is terminated before the next instruction of main thread is executed by the program.
- 
Main Thread
Thread 1
Thread 2
thread1.join()
Main Thread will stop execution at this point and wait for Thread 1 to complete.
thread2.join()
Main Thread will stop execution at this point and wait for Thread 2 to complete.

join() method

- The join() method in the Thread class is a mechanism for inter-thread synchronization, allowing one thread to wait for the completion of another thread. When a thread calls the join() method on another thread object, the calling thread will pause its execution and wait until the target thread (the one on which join() was called) finishes its execution.
- **Key aspects of the join() method:**
  - **Synchronization:** It ensures that the current thread waits until the specified thread completes its execution, guaranteeing sequential execution between threads when needed.
  - **Blocking behavior:** When join() is invoked on a thread, the calling thread enters a waiting state and remains there until the target thread terminates.

Synchronization

- Synchronization in Java is a mechanism that ensures that only one thread can access a shared resource (like a variable, object, or method) at a time. It prevents concurrent threads from interfering with each other while modifying shared data.
- **Why is Synchronization Needed?**
- **Prevents Data Inconsistency:** Ensures that multiple threads don't corrupt shared data when accessing it simultaneously.
- **Avoids Race Conditions:** Allows only one thread to execute a critical section at a time, maintaining predictable results.
- **Maintains Thread Safety:** Protects shared resources from concurrent modification by multiple threads.
- **Ensures Data Integrity:** Keeps shared data accurate and consistent throughout program execution.

Analogy for Synchronization: Bathroom Key

- Imagine a **single bathroom** in an office with **one key**.
- When one person enters, they **lock** the bathroom (takes the key).
- Others must **wait outside** until the key is returned.
- Only after the person leaves (releases the lock), the next person can enter.
- **In Java:**
- Bathroom → **shared resource (critical section)**
- Key → **monitor lock**
- Person → **thread**
- Locking bathroom → **entering synchronized block**

Synchronization

- You can synchronize your code in either of two ways. Both involve the use of the
- **synchronized** keyword:
  - synchronized Methods
  - synchronized block
  - Static Synchronized Method

- **Synchronized Methods:** A synchronized method ensures that only one thread can execute it at a time on the same object instance.
- **Synchronized block:** Instead of synchronizing an entire method, Java allows synchronization on specific blocks of code. This improves performance by locking only the necessary section.
- **Static synchronized method** is used to synchronize static methods

synchronized methods

- While a thread is inside a synchronized method, all other threads that try to call it (or any other synchronized method) on the same instance have to wait.
- Any time that you have a method, or group of methods, that manipulates the internal state of an object in a multithreaded situation, you should use the **synchronized** keyword to guard the state from race conditions.
- Remember, once a thread enters any synchronized method on an instance, no other thread can enter any other synchronized method on the same instance. However, nonsynchronized methods on that instance will continue to be callable.

Ex: Without synchronized, without Multiple threads

```
class CounterEx1{
    int count;
    public void increment(){
        count++;
    }

}
public class WOSyncEx1 {
    public static void main(String[] args)
        { CounterEx1 c=new CounterEx1();
        for(int i=0;i<1000;i++)
            c.increment();

        System.out.println(c.count);
    }
}
```

Ex: Without synchronized, with Multiple threads

```
class CounterEx2{
    int count;
    public void increment(){
        count++;
    }
}
class Class1WO implements Runnable{
    CounterEx2 c;
    Class1WO(CounterEx2 c){
        this.c=c;
    }
    public void run(){
        for(int i=0;i<1000;i++)
            c.increment();
    }
   }

class Class2WO implements Runnable{
    CounterEx2 c;
    Class2WO(CounterEx2 c){
        this.c=c;
    }
    public void run(){
        for(int i=0;i<1000;i++)
            c.increment();
    }

}
public class WOSyncEx2 {
    public static void main(String[] args) throws InterruptedException{
        CounterEx2 c=new CounterEx2();
        Class1WO obj1=new Class1WO(c);
        Class2WO obj2=new Class2WO(c);
        Thread t1=new Thread(obj1);
        Thread t2=new Thread(obj2);
        t1.start();
        t2.start();
        t1.join();
        t2.join();
        System.out.println(c.count);
    }
}
```

What will be the output?

Created By Dr. Ritu Jain

- **Explanation of the Race Condition:**
- **Shared Resource:** The count variable in the Counter class is a shared resource accessed by multiple threads (t1 and t2).
- ~~**Non-Atomic Operation:** The count++ operation, while appearing as a single line of~~ code, is not atomic at the CPU level. It typically involves three steps:
  - Read the current value of count into a register.
  - Increment the value in the register.
  - Write the new value back to count in memory.
- **Interleaving of Operations:** When t1 and t2 execute concurrently, their operations can interleave in an unpredictable order.
- **Scenario:** Suppose count is 0.
  - t1 reads count (value 0).
  - t2 reads count (value 0) before t1 writes back its incremented value.
  - t1 increments its local copy to 1 and writes it back to count. Now count is 1.
  - t2 increments its local copy (which was 0) to 1 & write it to count. Now count is still 1.
- In this scenario, one increment is lost, and the final count is 1 instead of the expected 2.
- **Unpredictable Results:** Because the exact timing and interleaving of thread execution can vary, running this program multiple times will likely produce different "Final Count" values, all of which will be less than the expected 2000 (1000 increments from each thread). This inconsistent and incorrect behavior is the hallmark of a race condition.

Ex: Synchronized method with Multiple threads

```
class CounterEx3{
    int count;
    public synchronized void increment(){
        count++;
    }
}
class Class1S implements Runnable{
    CounterEx3 c;
    Class1S(CounterEx3 c){
        this.c=c;
    }
    public void run(){
        for(int i=0;i<1000;i++)
            c.increment();
    }
}
class Class2S implements Runnable{
    CounterEx3 c;
    Class2S(CounterEx3 c){
        this.c=c;
    }
    public void run(){
        for(int i=0;i<1000;i++)
            c.increment();
    }


}
public class SyncEx3 {
    public static void main(String[] args) throws InterruptedException
        { CounterEx3 c=new CounterEx3();
        Class1S obj1=new Class1S(c);
        Class2S obj2=new Class2S(c);
        Thread t1=new Thread(obj1);
        Thread t2=new Thread(obj2);
        t1.start();
        t2.start();
        t1.join();
        t2.join();
        System.out.println(c.count);
    }
}
```

With the help of synchronized method, only one thread can access them at a time, ensuring the correct final count.

synchronized block

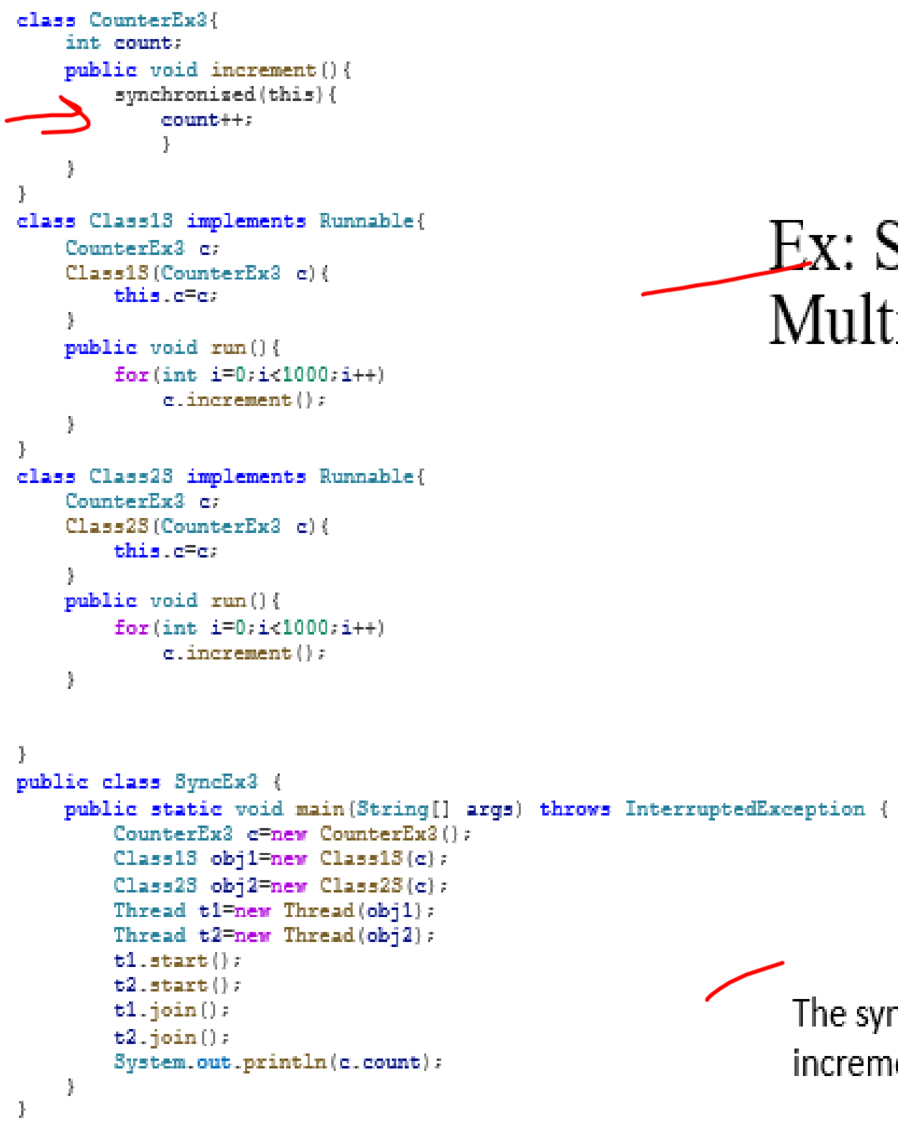
- While creating **synchronized** methods within classes that you create is an easy and effective means of achieving synchronization, it will not work in all cases. Consider the following. Imagine that you want to synchronize access to objects of a class that was not designed for multithreaded access. That is, the class does not use **synchronized** methods. Further, this class was not created by you, but by a third party, and you do not have access to the source code. Thus, you can't add **synchronized** to the appropriate methods within the class. How can access to an object of this class be synchronized?
- Fortunately, the solution to this problem is quite easy: You simply put calls to the methods defined by this class inside a **synchronized** block.
- This is the general form of the **synchronized** statement:

```
synchronized(objRef) {
  // statements to be synchronized
}
```

-

Here, *objRef* is a reference to the object being synchronized. A synchronized block ensures that a call to a synchronized method that is a member of *objRef*'s class occurs only after the current thread has successfully entered *objRef*'s monitor.

Ex: Synchronized block with Multiple threads



```
class CounterEx3{
    int count;
    public void increment(){
        synchronized(this){
            count++;
            }
    }
}
class Class1S implements Runnable{
    CounterEx3 c;
    Class1S(CounterEx3 c){
        this.c=c;
    }
    public void run(){
        for(int i=0;i<1000;i++)
            c.increment();
    }
}
class Class2S implements Runnable{
    CounterEx3 c;
    Class2S(CounterEx3 c){
        this.c=c;
    }
    public void run(){
        for(int i=0;i<1000;i++)
            c.increment();
    }


}
public class SyncEx3 {
    public static void main(String[] args) throws InterruptedException {
        CounterEx3 c=new CounterEx3();
        Class1S obj1=new Class1S(c);
        Class2S obj2=new Class2S(c);
        Thread t1=new Thread(obj1);
        Thread t2=new Thread(obj2);
        t1.start();
        t2.start();
        t1.join();
        t2.join();
        System.out.println(c.count);
    }
}
```

The synchronized block ensures mutual exclusion only for the increment statement, reducing the locking overhead.

```
class Table{
  synchronized static void printTable(int n){
    for (int i = 1; i <=3; i++){
                System.out.println(n * i);
      try {
      } catch (Exception e) {
        System.out.println(e);
      }}}}

class Thread1 extends Thread{
  public void run() {
    Table.printTable(1);
  }
}

class Thread2 extends Thread {
  public void run() {
    Table.printTable(10);
  }
}
public class SynB{
  public static void main(String[] args){
    Thread1 t1 = new Thread1();
    Thread2 t2 = new Thread2();
    t1.start();
    t2.start();
  }
}
```

Static synchronization is used to synchronize static methods. In this case, the lock is placed on the class object rather than the instance.

Explanation: Both threads t1 and t2 call the static synchronized method printTable(). The lock is applied to the Table.class object, ensuring that only one thread can access the method at a time, even if no object instance is shared.

D Y PATIL
UNIVERSITY
NAVI MUMBAI

Inter-thread communication

- Inter-thread communication in Java is a mechanism that enables threads to **cooperate and coordinate** with each other while sharing common resources.
- When multiple threads work together on shared data, simply using synchronization is not enough, because synchronization only ensures **mutual exclusion**—it does not provide a way for threads to **message each other** or **wait for certain conditions**.
- Inter-thread communication solves this problem by allowing threads to **pause**, **wait**, and **resume** based on specific conditions.

Inter-thread communication

- Java supports inter-thread communication through three important methods defined in the **Object** class:
  - **wait()**
  - **notify()**
  - **notifyAll()**
- These methods help threads communicate efficiently without causing **busy waiting**, where a thread repeatedly checks a condition in a loop and wastes CPU resources.

Polling

- **What is Polling, and What are the Problems with it?**
- The process of testing a condition repeatedly till it becomes true is known as polling. Polling is usually implemented with the help of loops to check whether a particular condition is true or not. If it is true, a certain action is taken. For example, in a classic queuing problem where one thread is producing data, and the other is consuming it.
- **Problem with Polling:** This wastes many CPU cycles and makes the implementation inefficient. This slows down the execution, and it keeps on checking the condition repeatedly.

How does Java Multi-Threading Tackle this problem?

- To avoid polling, Java uses three methods, namely, wait(), notify(), and notifyAll(). All these methods belong to the object class, so all classes have them. They must be used within a synchronized block only.

How does Java Multi-Threading Tackle Polling problem: wait(), notify(), and notifyAll() Methods

- **public final void wait() throws InterruptedException:** Causes the current thread to **release the lock** and enter the **waiting state.** The thread stays waiting until another thread calls notify() or notifyAll(). Must be used inside a synchronized block/method.
- **void notify()**: It wakes up one single thread called wait() on the same object. The choice of which waiting thread is awakened is not controlled by Java.
- **void notifyAll():** It wakes up all the threads called wait() on the same object. All awakened threads must acquire the lock again to proceed. One of the threads will be granted access.

Rules for Inter-Thread Communication

- These methods must be called from within a **synchronized block or method**.
- A thread calling wait() **releases the lock.**
- After notify() or notifyAll(), awakened threads become runnable only after acquiring the lock.
- They work with the **intrinsic lock (monitor lock)** of the object.
- These rules ensure proper coordination and avoid race conditions.

D Y PATIL UNIVERSITY NAVI MUMBAI

Need for Inter-Thread Communication

- In multithreaded applications, one thread often depends on another. Example:
  - A **Producer** thread produces data.
  - A **Consumer** thread consumes the data.
- If the consumer tries to consume before the producer produces, it must wait.
  If the producer tries to produce when the buffer is full, it must wait.
- Inter-thread communication provides a clean mechanism to achieve this coordination.

Advantages of wait(), notify() and notifyAll()

- **Provides Thread Communication (Inter-Thread Coordination):** wait() and notify() allow threads to **communicate and coordinate** with each other, especially in producer–consumer scenarios. One thread can pause until another thread notifies it to continue.
- **Efficient Resource Utilization:** Instead of constantly checking a condition (busy waiting or polling), wait() **releases the lock** and puts the thread into a **waiting state**, saving CPU time.
- **Avoids Continuous Polling:** Without wait(), a thread might repeatedly check a condition in a loop—wasting CPU cycles. wait() puts the thread to sleep until a condition changes.
-

D Y PATIL UNIVERSITY NAVI MUMBAI

Advantages of wait(), notify() and notifyAll()

- **Helps in Implementing Synchronization Patterns:** These methods help implement:
  - Producer–consumer
  - Reader–writer
  - Blocking queues
  - Resource allocation systems

- **Works at the Monitor Level (Object Lock)**
- Since these methods work with **intrinsic locks**, they integrate naturally with synchronized methods and blocks.
-

Example of Inter Thread Communication: Solving Producer Consumer Problem using Wait(), notify()

```
class Data {
   int value;
   boolean isAvailable = false;
   public synchronized void produce(int v) throws InterruptedException {
      while (isAvailable) {
         wait();  // wait until consumer consumes
      }
      value = v;
      System.out.println("Produced: " + value);
      isAvailable = true;
      notify();  // notify consumer
   }
   public synchronized int consume() throws InterruptedException {
      while (!isAvailable) {
         wait();  // wait until producer produces
      }
      System.out.println("Consumed: " + value);
      isAvailable = false;
      notify();  // notify producer
      return value;
   }
}
```

Cont'd on next slide…

Example of Inter Thread Communication: Solving Producer Consumer Problem using Wait(), notify()

```
class Producer implements Runnable {
   Data data;

   Producer(Data d) {
      this.data = d;
   }

   public void run() {
      for (int i = 1; i <= 5; i++) {
         try {
            data.produce(i);
            Thread.sleep(500);  // slow down production
         } catch (InterruptedException e) {
            e.printStackTrace();
         }
      }
   }
}
```

Cont'd on
next slide…

Example of Inter Thread Communication: Solving Producer Consumer Problem using Wait(), notify()

```
class Consumer implements Runnable {
  Data data;

  Consumer(Data d) {
    this.data = d;
  }

  public void run() {
    for (int i = 1; i <= 5; i++) {
      try {
        data.consume();
        Thread.sleep(500);  // slow down consumption
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
  }
}
```

Cont'd on next slide…

Example of Inter Thread Communication: Solving Producer Consumer Problem using Wait(), notify()

```
public class ProducerConsumerDemo {
  public static void main(String[] args) {
    Data data = new Data();

    Thread producerThread = new Thread(new Producer(data), "Producer");
    Thread consumerThread = new Thread(new Consumer(data), "Consumer");

    producerThread.start();
    consumerThread.start();
  }
}
```

D Y PATIL
UNIVERSITY
NAVI MUMBAI

Thank You

D Y PATIL
UNIVERSITY
NAVI MUMBAI