Basic Java

Unit 10 - Threads

Pratian Technologies (India) Pvt. Ltd.
www.pratian.com

PRATIAN
TECHNOLOGIES

PRATIAN TECHNOLOGIES

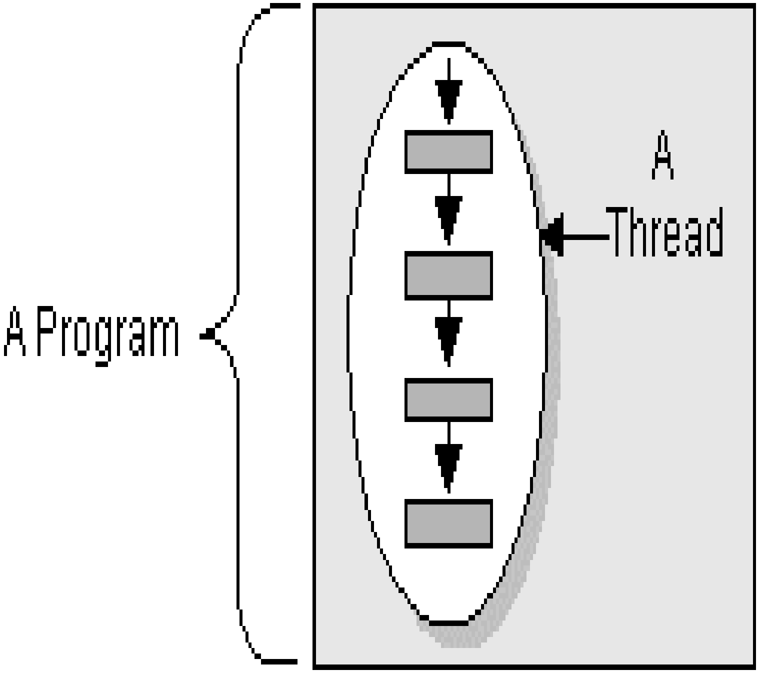
Topics

- What are threads ?
- Need for Multiple Threads
- Time Scheduling
- Creating multiple threads
- Thread class
- The Runnable interface
- Thread priorities
- sleep() and join()
- Daemon threads
- The problems that comes with parallelism
- What are race conditions?
- Thread synchronization
- Synchronizing critical code
- Synchronized method() Vs Synchronized block

What is a Thread ?

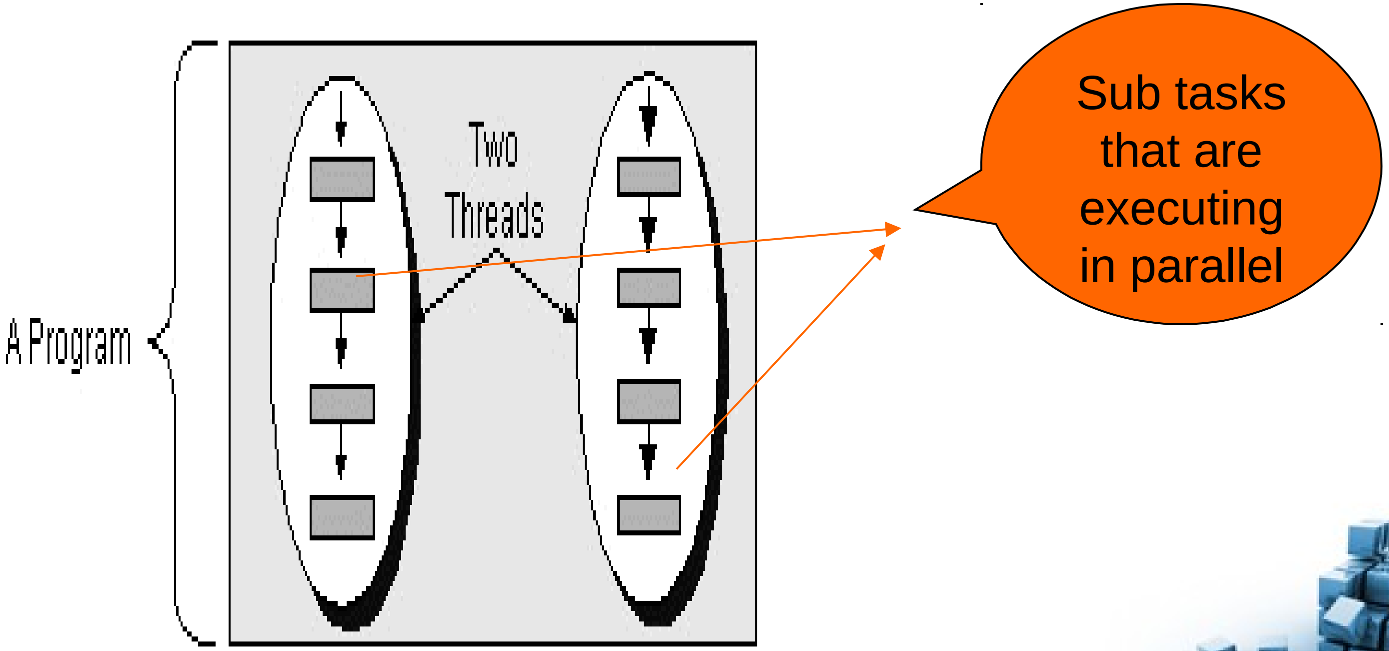
- A Thread is a single sequential flow of control within a program.

- It is an independently running subtask.



Multiple Threads

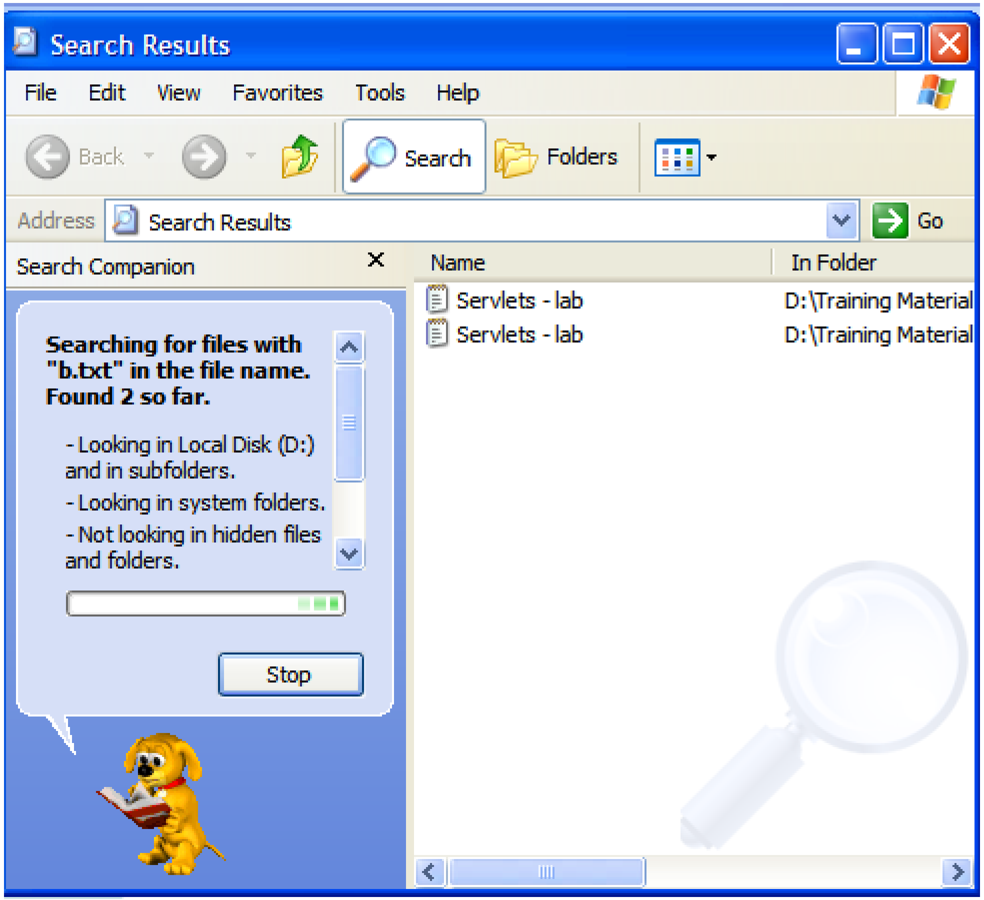
- A program having multiple threads implies that there are multiple flows of execution within the program



Sub tasks that are executing in parallel

Need for Multiple Threads

- Let us suppose that we have to develop an application that is similar to Windows search



Address: Search Results

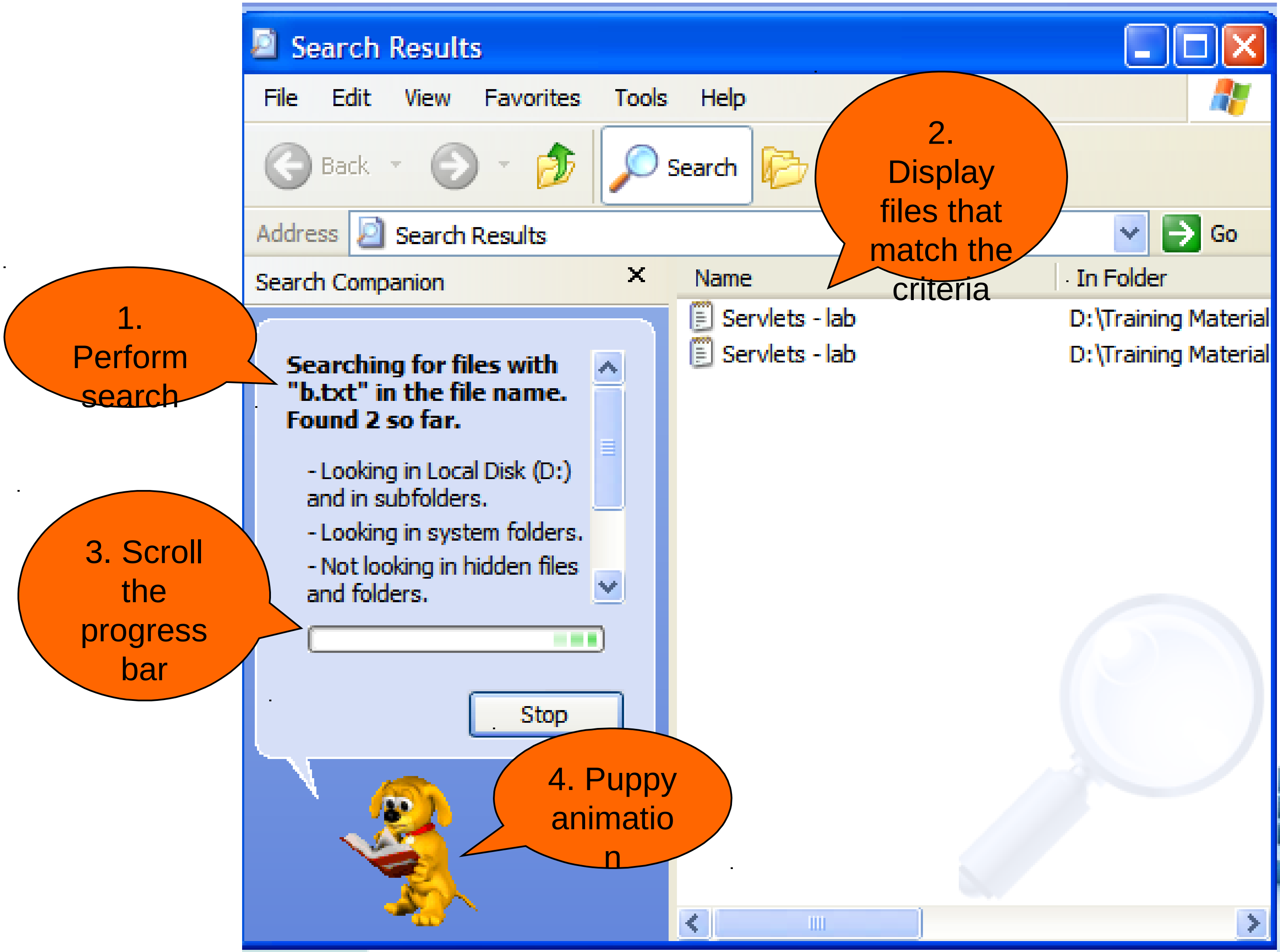
Searching for files with "b.txt" in the file name. Found 2 so far.

- Looking in Local Disk (D:) and in subfolders.
- Looking in system folders.
- Not looking in hidden files and folders.

Name	In Folder
Servlets - lab	D:\Training Material
Servlets - lab	D:\Training Material

What are the different functionalities that we identify ?

Need for Multiple Threads



Searching for files with "b.txt" in the file name. Found 2 so far.

- Looking in Local Disk (D:) and in subfolders.
- Looking in system folders.
- Not looking in hidden files and folders.

Name	In Folder
Servlets - lab	D:\Training Material
Servlets - lab	D:\Training Material

1. Perform search

2. Display files that match the criteria

3. Scroll the progress bar

4. Puppy animatio n

Need for Multiple Threads

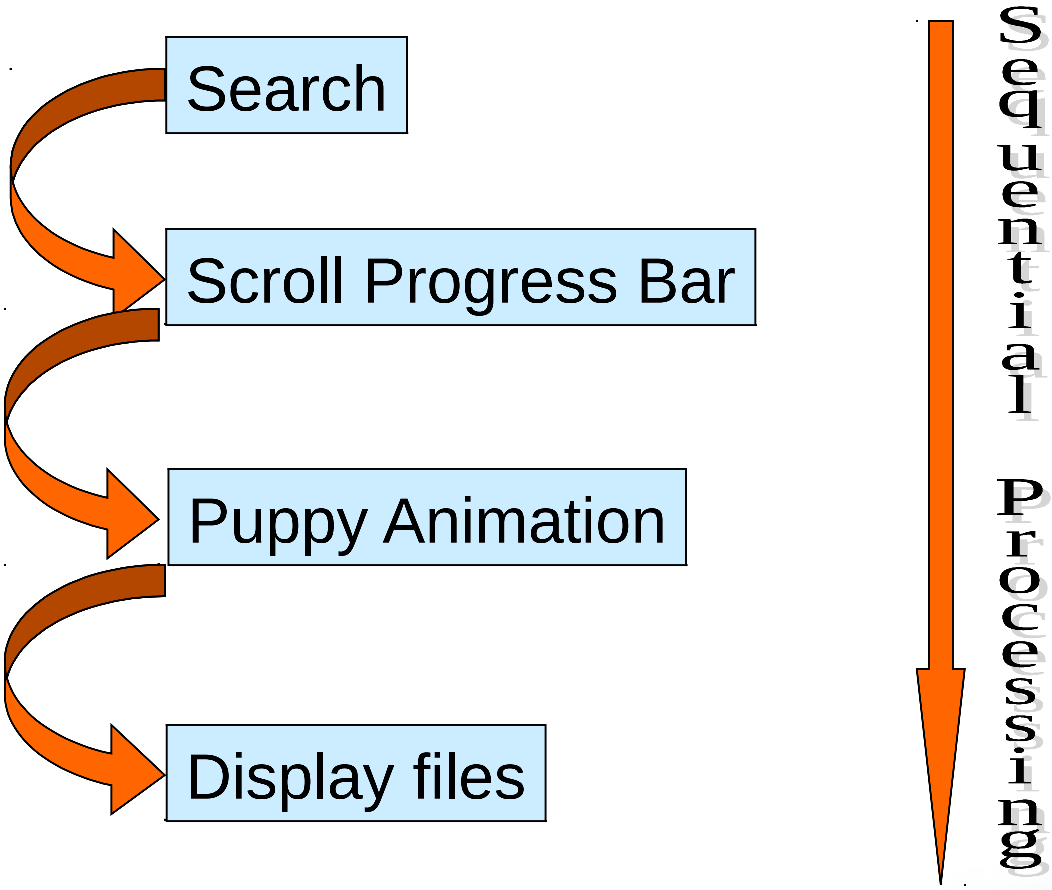
- Look at the below code snippet

```
public class WindowsSearch
{
      public void search(String fileName)
      {
                // Implementation
     }
      public void scrollProgressBar()
    {
                 // Implementation
    }
      public void animatePuppy()
    {
                 // Implementation
    }
      public void displayFiles()
    {
                 // Implementation
    }
}
```

```
class SearchDemo
{
    public static void main()
    {
        WindowsSearch ws =
         new WindowsSearch();

         ws.search(fileName);
         ws.scrollProgressBar();
         ws.animatePuppy();
         ws.displayFiles();
      }
}
```

What is the problem ?



Search

Scroll Progress Bar

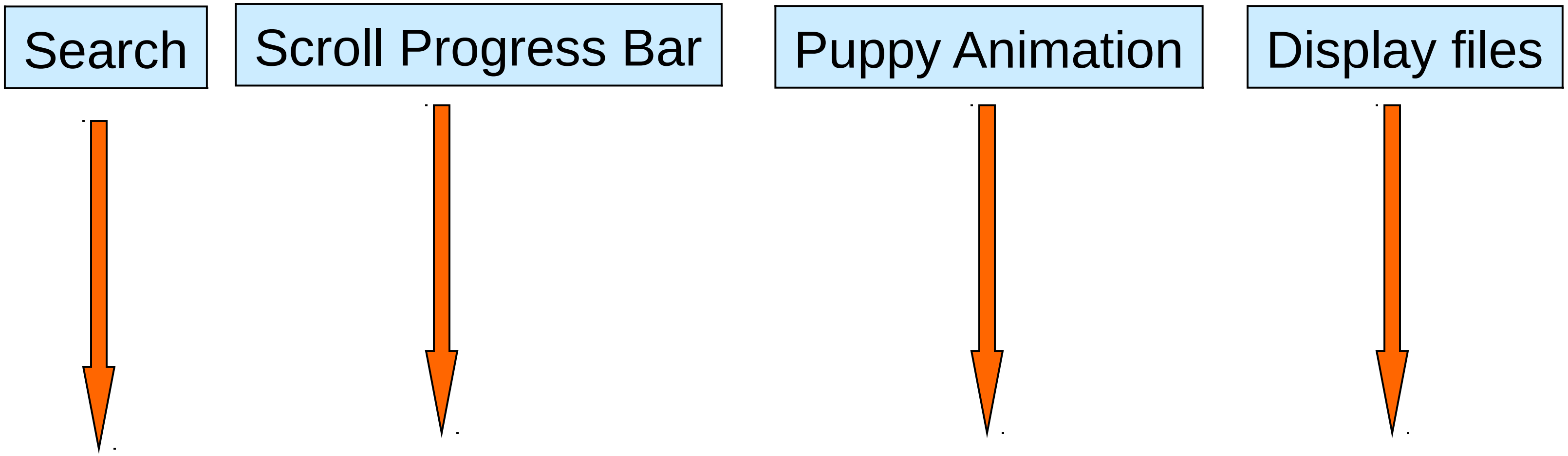
Puppy Animation

Display files

Sequential Processing

Need for Multiple Threads

- What we desire to achieve is parallel processing


Search

Scroll Progress Bar

Puppy Animation

Display files

- Every task is processed by one ‘thread’
- All four threads execute ‘almost’ at the same time

Why use Threads ?

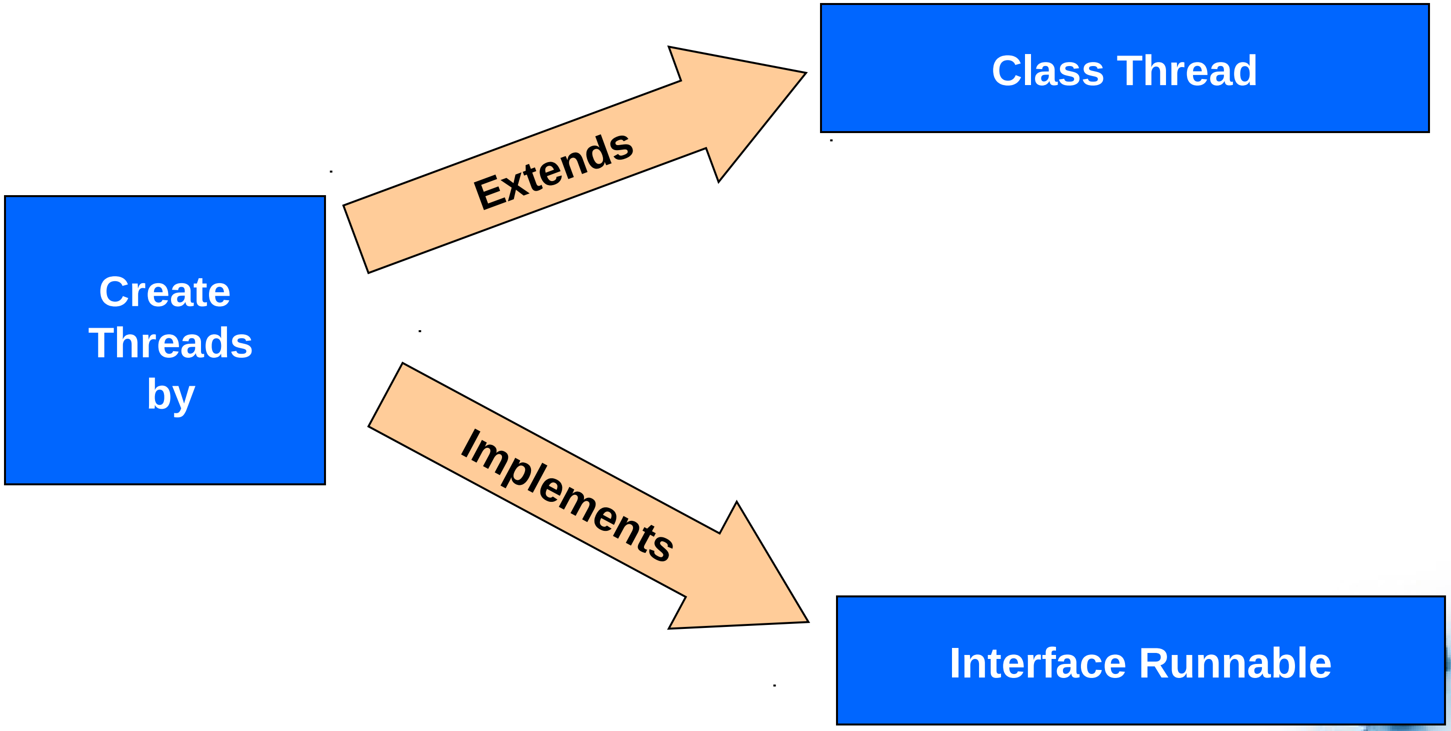
- Threads can be used in numerous scenarios
  - To improve the responsiveness of application.
  - To separate out data processing and input/output operations.
    - For non blocking input/output operations.
  - To handle asynchronous events (event handling such as a mouse click).
  - To perform repetitive or timed tasks (animations).
  - Management of multiple service request with unpredictable arrival.

How is Threading done?

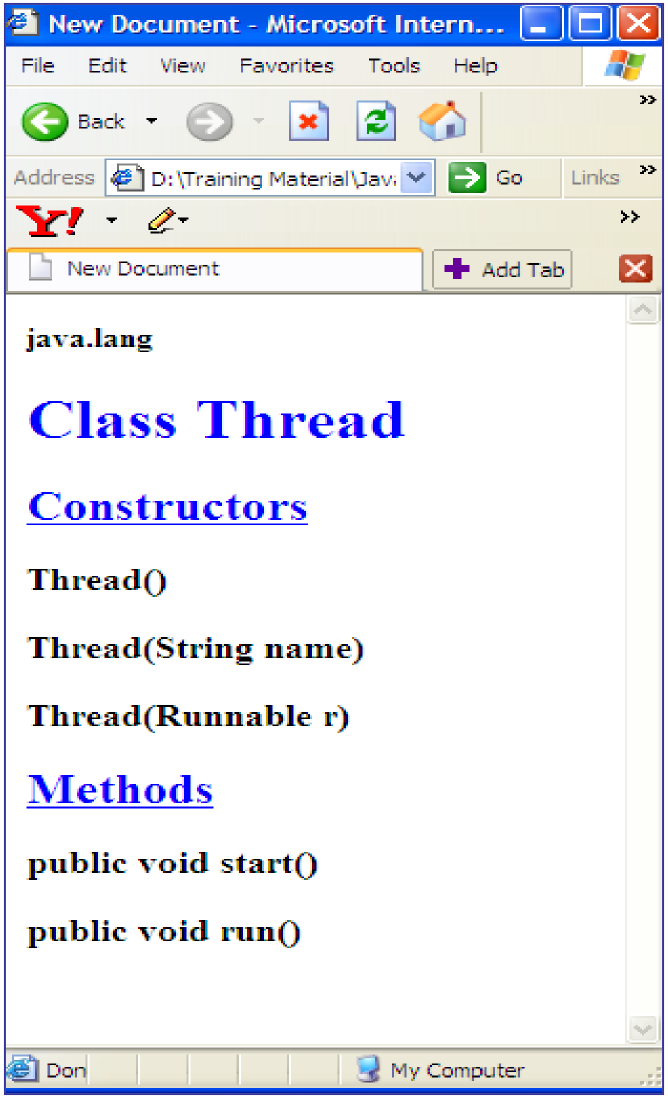
- java.lang package offers
  - a **Thread** class
  - a **Runnable** interface

How to create threads?

- There are two ways in which we can create threads in Java
  - By extending the Thread class
  - By implementing the Runnable Interface



Class Thread


New Document - Microsoft Intern...
File Edit View Favorites Tools Help
Back
Address D:\Training Material\Jav:
Go
Links
New Document
Add Tab
Don
My Computer

java.lang

Class Thread

Constructors

Thread()

Thread(String name)

Thread(Runnable r)

Methods

public void start()

public void run()

Creating a Thread

- Steps Involved
  - Define a class that **extends Thread**.
  - **Override** the **run()** method
  - Instantiate the class
  - Spawn the thread by making a **call** to **start()** method
  - **start()** method automatically calls **run()** and triggers execution of the thread.

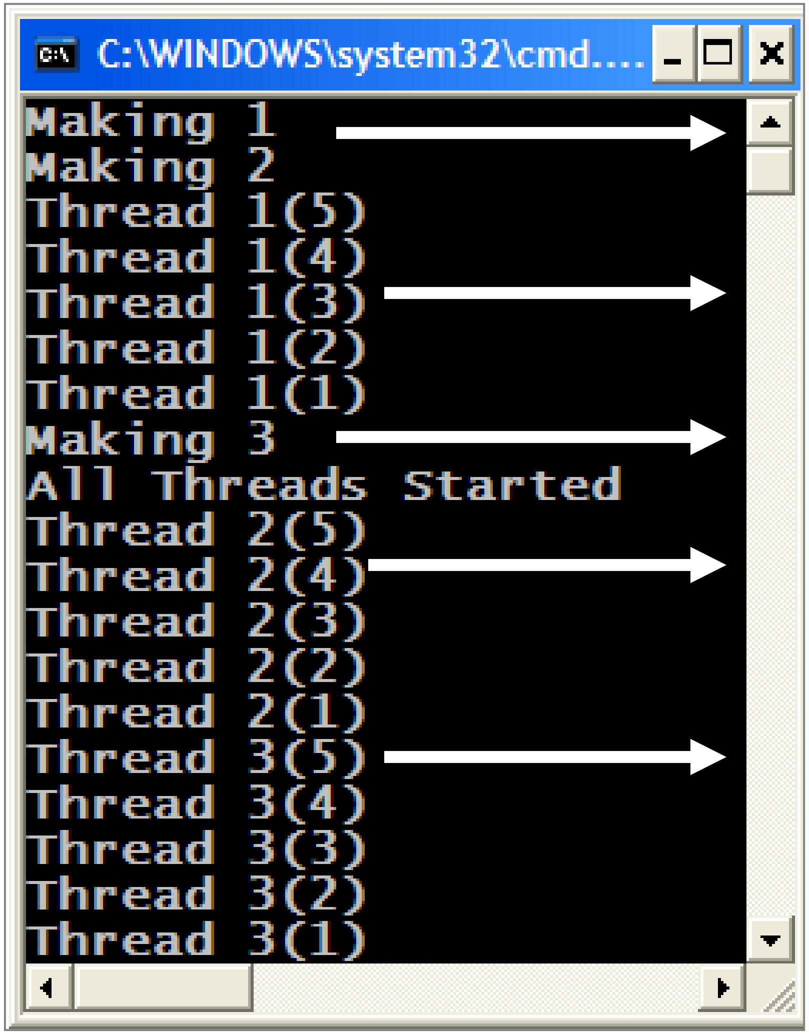
Example

```
class SimpleThread extends Thread {
    private int countDown = 5;
    private static int trdCount = 0;
    private int trdNum = ++trdCount;
    SimpleThread() {
        System.out.println("Making         thread " +
    trdNum);
    }
    public void run() {
        while(true) {
            System.out.println("Thread" +
            trdNum + "(" + countDown+")");
              if(--countDown == 0) return;
        }
    }
}
```

```
class ThreadDemo
{
    public static void
main(String[] s)
    {
        for(int i=0;i<=5;i++)
            new SimpleThread().
                        start();
        System.out.println("All
              Threads Started");
    }
}
```

Sample Listing :
ThreadDemo.java

Executing SimpleThread



```
Making 1
Making 2
Thread 1(5)
Thread 1(4)
Thread 1(3)
Thread 1(2)
Thread 1(1)
Making 3
All Threads Started
Thread 2(5)
Thread 2(4)
Thread 2(3)
Thread 2(2)
Thread 2(1)
Thread 3(5)
Thread 3(4)
Thread 3(3)
Thread 3(2)
Thread 3(1)
```

Main thread executing…

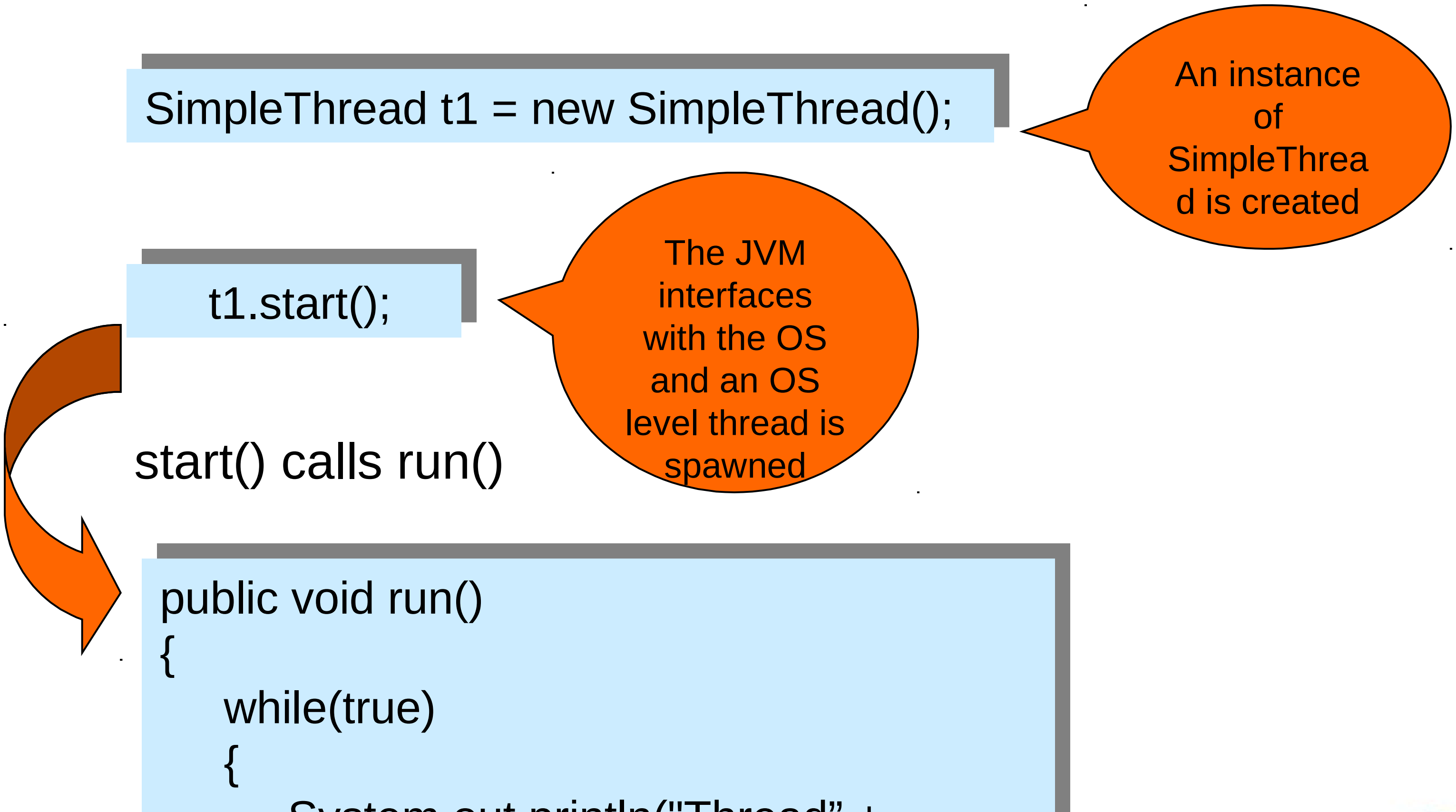
Thread 1 executing…

Main thread executing…

Thread 2 executing…

Thread 3 executing…

How does it work ?



```
SimpleThread t1 = new SimpleThread();
```

An instance of SimpleThread is created

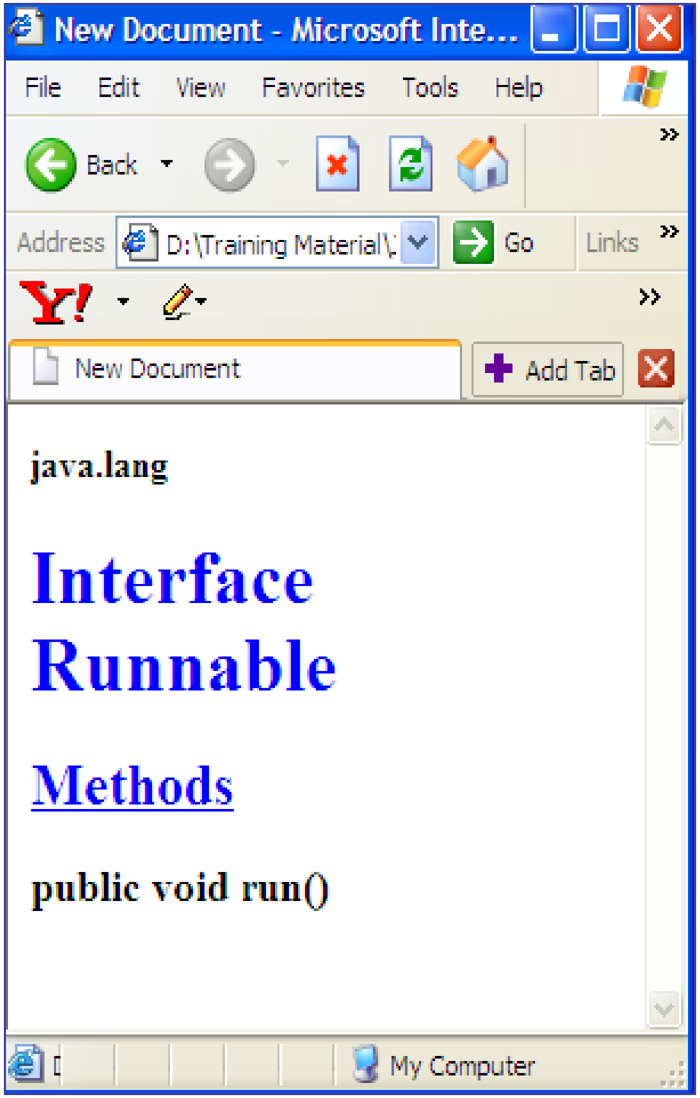
```
t1.start();
```

The JVM interfaces with the OS and an OS level thread is spawned

start() calls run()

```
public void run()
{
     while(true)
     {
          System.out.println("Thread” +
               trdNum + "(“ + countDown+")");
           if(--countDown == 0) return;
      }
}
```

Interface Runnable


New Document - Microsoft Inte...
File Edit View Favorites Tools Help
Back
Address D:\Training Material\
Go
Links
Y!
New Document
Add Tab
My Computer

java.lang

Interface Runnable

Methods

public void run()

NOTE:

The object of the class that implements **Runnable** interface becomes a **runnable object.**

Creating a Thread

- Steps Involved
  - Define a class that **implements Runnable interface**.
  - Provide implementation for the **run()** method
  - Instantiate the class
    - The object is now a Runnable object
  - **Create** a Thread instance and **assign** the Runnable object to the thread.
  - Spawn the thread by making a **call to start()** method
  - **start()** method automatically calls **run()** and triggers execution.

Example

```
class NewThread implements Runnable
{
      int start,stop;

      NewThread(int start,int stop)
      {
                this.start=start;
                this.stop=stop;
      }
      public void run()
      {
            for(int i=stop;i>start;i--)
                System.out.println(Thread.currentThread() + " : " + i);

            System.out.println("Exiting " + Thread.currentThread());
      }
}
```

Example

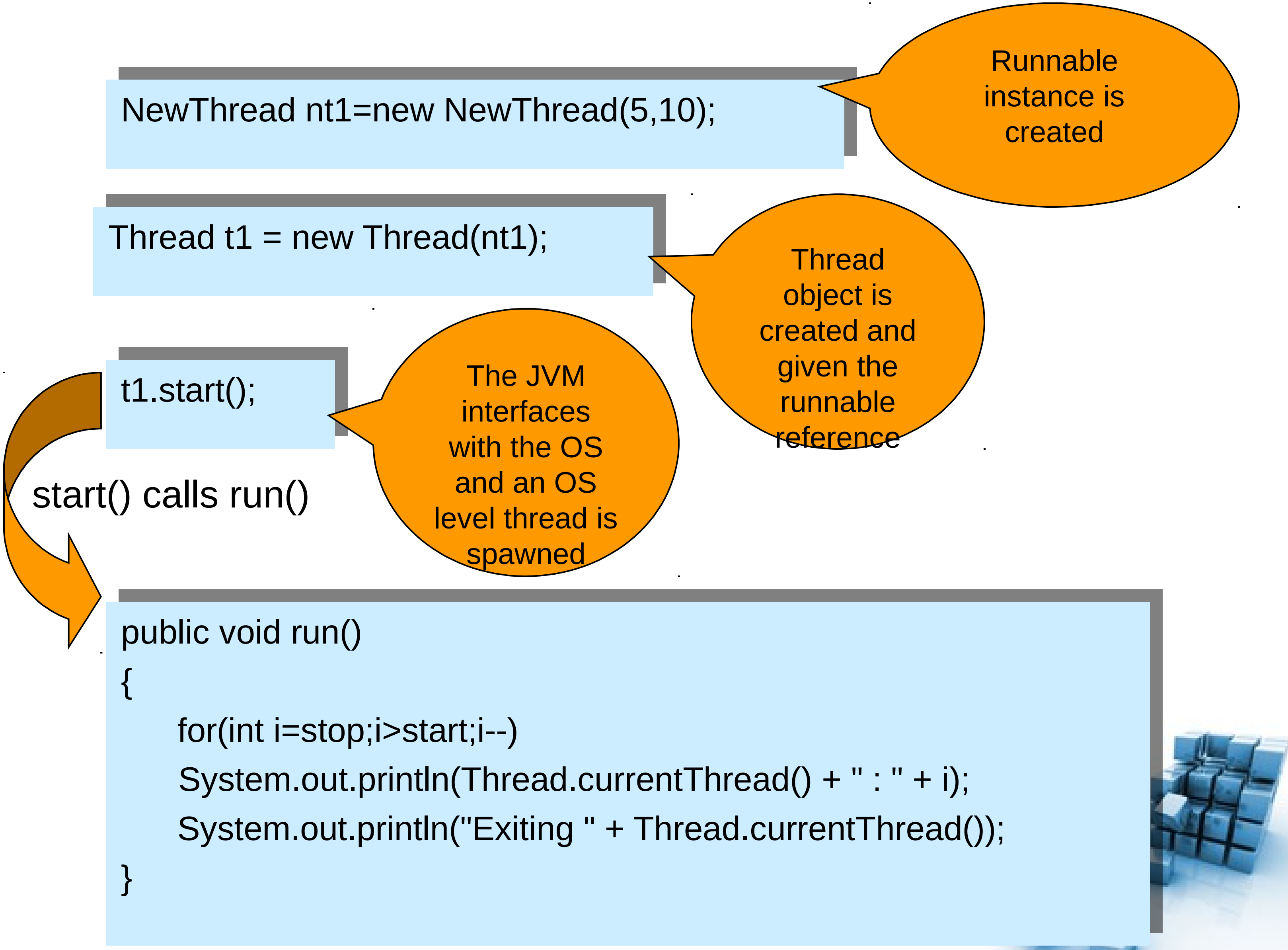
```
class RunnableDemo
{
     public static void main(String args[])
     {
              NewThread nt1=new NewThread(5,10);
              NewThread nt2=new NewThread(15,18);
              Thread t1 = new Thread(nt1);
              Thread t2 = new Thread(nt2);
              System.out.println(“Starting Thread 1 “);
              t1.start();
              System.out.println(“Starting Thread 2 “);
              t2.start();
              System.out.println("Main thread exiting");
     }
}
```

Runnable instance is created

Thread object is created and given the runnable reference

Sample Listing: **RunnableDemo.java**

How does it work ?



```
NewThread nt1=new NewThread(5,10);
```

Runnable instance is created

```
Thread t1 = new Thread(nt1);
```

Thread object is created and given the runnable reference

```
t1.start();
```

The JVM interfaces with the OS and an OS level thread is spawned

start() calls run()

```
public void run()
{
      for(int i=stop;i>start;i--)
      System.out.println(Thread.currentThread() + " : " + i);
      System.out.println("Exiting " + Thread.currentThread());
}
```

Which to choose?

- If you extend the **Thread** Class, that means that subclass cannot extend any other Class, but if you implement **Runnable** interface then you can do this.

- And the class implementing the **Runnable** interface can avoid the full overhead of **Thread** class which can be excessive.

Thread Priorities

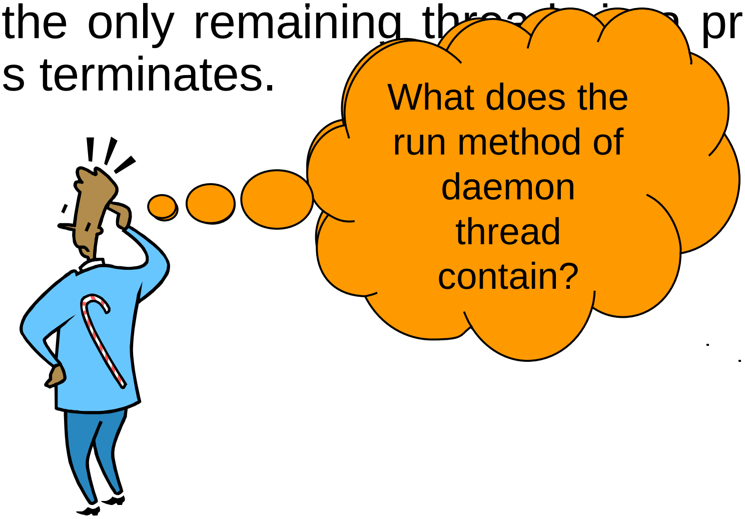
- A thread’s priority is used to decide when to switch from one running thread to the next, which is called *context switching.*
- All threads inherit their priority from the thread that created it.

- Thread priorities are between 1 and 10.
  - Ten is the highest priority (MAX_PRIORITY)
  - One is the lowest (MIN_PRIORITY)
  - Five is the default priority (NORM_PRIORITY)

- Threads can be assigned a priority using the setPriority() method of the Thread class.
  - **void setPriority(int newPriority)**

Daemon Threads

- Daemon threads are service providers for other threads running in the same process as the daemon thread.
    - Examples of daemon threads within the JVM.
        - Garbage collector thread,
        - finalizer thread
- The run() method for a daemon thread is typically an infinite loop that waits for a service request.
- Daemon threads keep executing until there is atleast one active non daemon thread.
    - When the only remaining threads in a process are daemon threads, the process terminates.



Daemon Threads

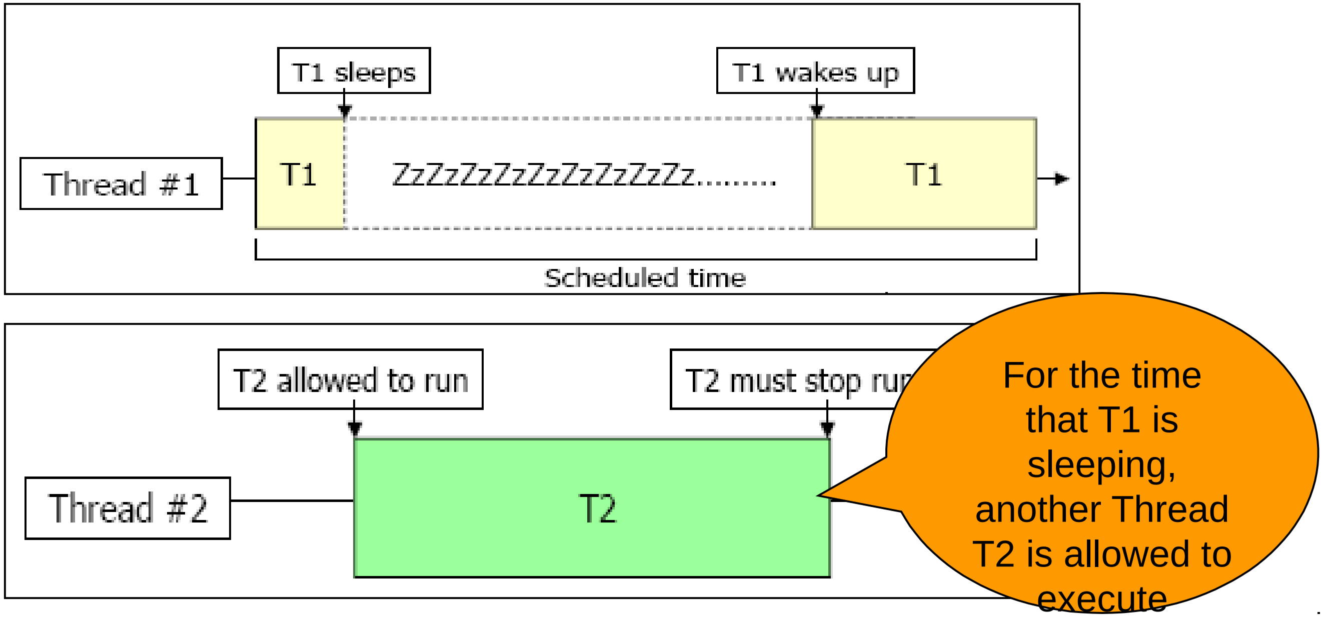
- To specify a thread as daemon thread
  - setDaemon(true) .
    - This method must be called before invoking the start method on the thread.

- Every thread acquires its ‘daemon’ property from its parent thread.
  - Threads created by daemon threads are all daemon by default.
  - Threads created by non daemon threads are non daemon by default.

Sample Listing: **DaemonDemo.java**

sleep() method

- public static void sleep(long millis)
  - Causes the thread to cease execution for the specified number of milliseconds
  - Throws InterruptedException if interrupted by another thread



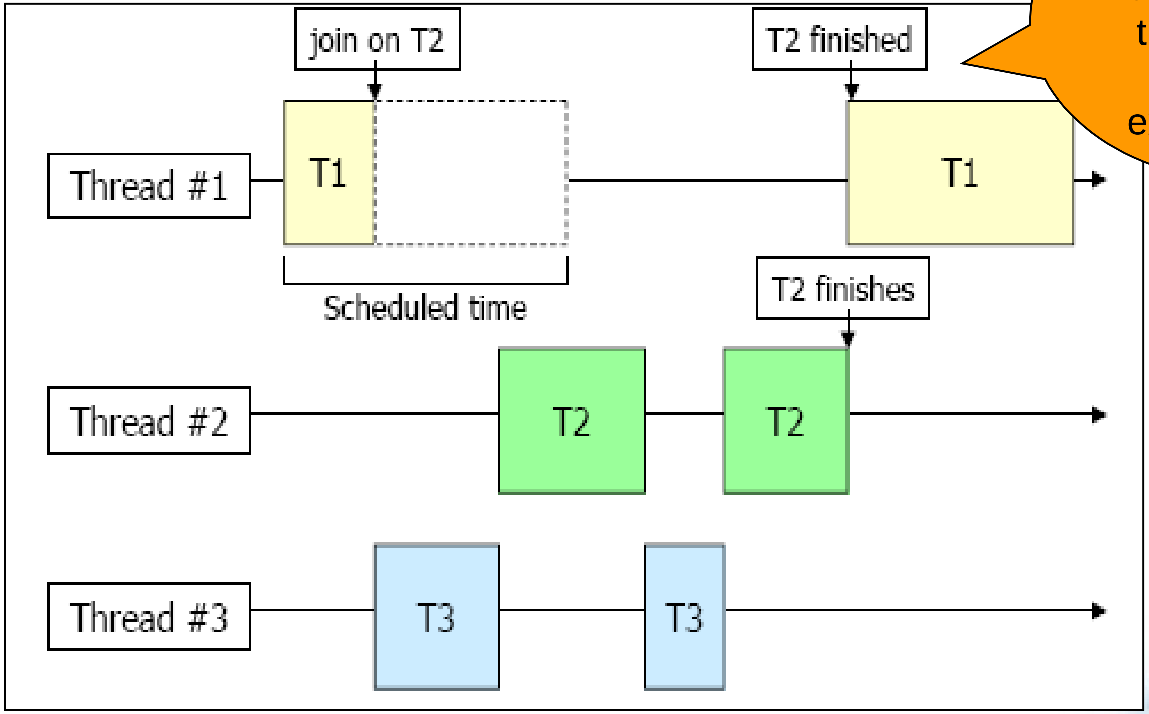
For the time that T1 is sleeping, another Thread T2 is allowed to execute

Sample Listing : **SleepDemo.java**

join() method

- public void join()
  - Causes the currently executing thread to wait for another thread to finish

SampleListing : **NoWaitDemo.java**



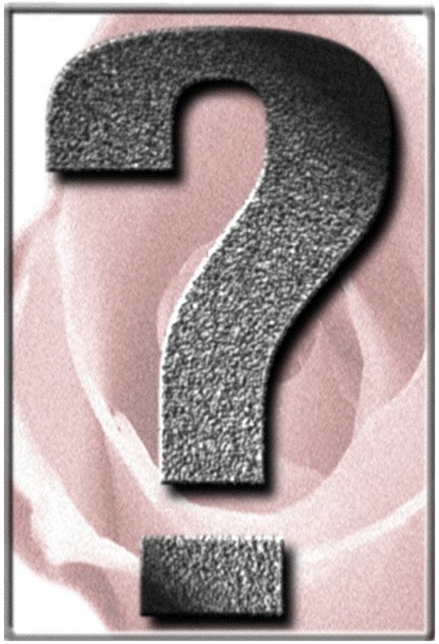
Thread1 will not execute till the time Thread2 completes executing fully

SampleListing : **JoinDemo.java**

Other methods

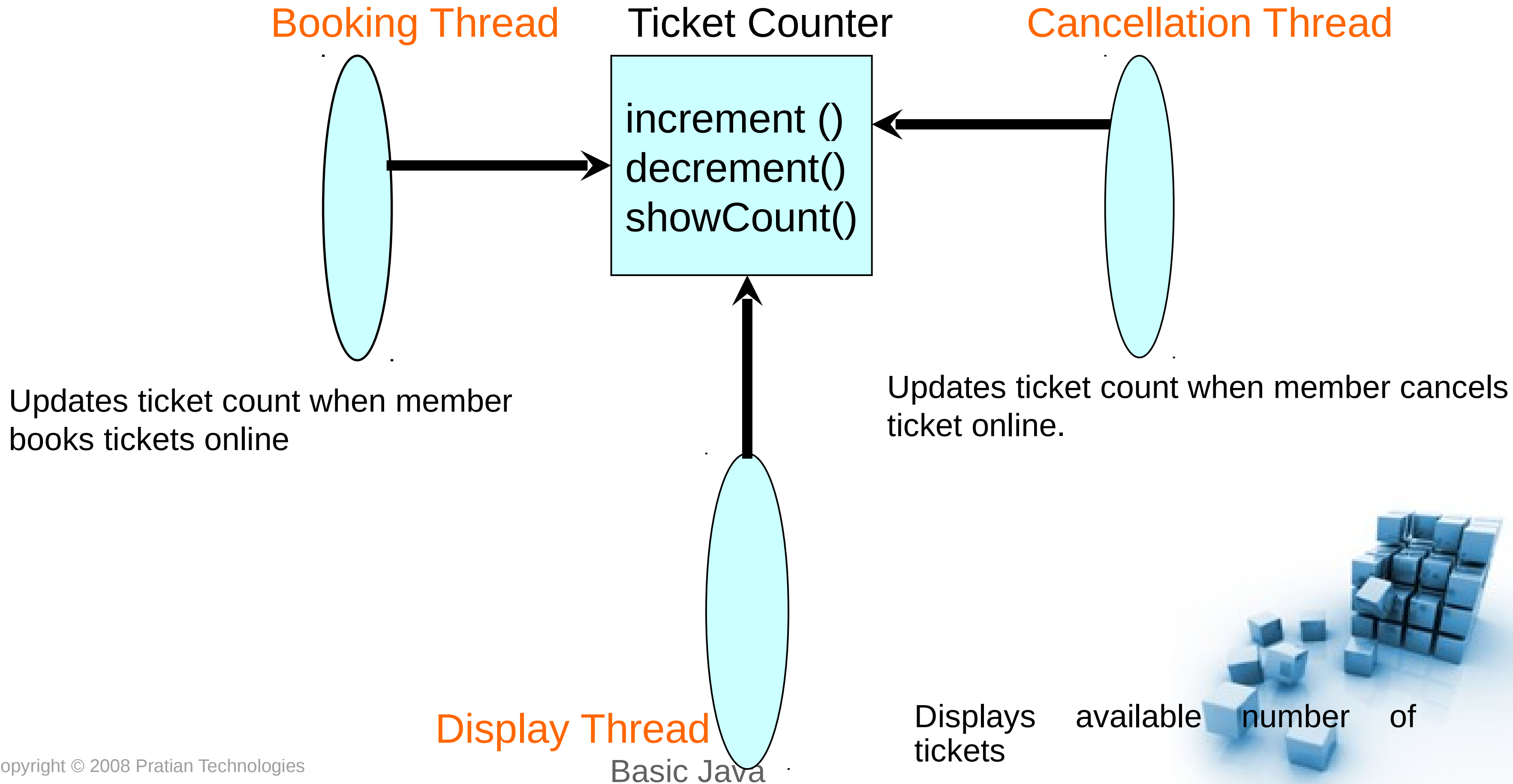
- **boolean isAlive()**
  - Determines if the thread is still running.
- **int getPriority()**
  - Returns the thread’s priority.
- **String getName()**
  - Returns the thread’s name.
- **setName(String name)**
  - Changes the name of this thread to the specified string.
- **static Thread currentThread()**
  - Returns a reference to the currently executing thread object
- **toString()**
  - Returns a string representation of this thread, including the thread's name, priority, and thread group.

Question time



Concurrent Access

- Sharing data among threads could cause inconsistencies, when multiple threads access the same object at the same time.
- Consider the below example



Booking Thread

Ticket Counter

Cancellation Thread

increment ()
decrement()
showCount()

Updates ticket count when member books tickets online

Updates ticket count when member cancels ticket online.

Display Thread

Displays available number of tickets

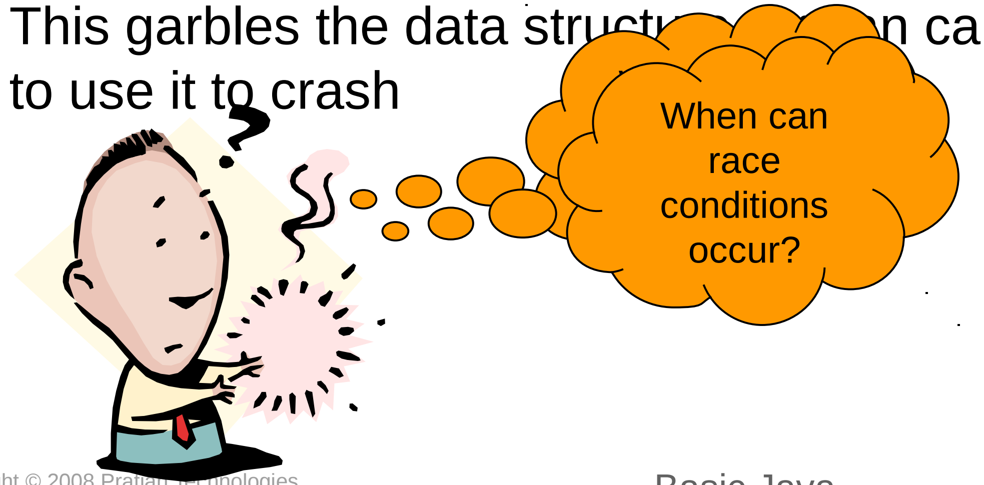
Race Conditions

- A race condition is a programming fault which produces unpredictable program state and behavior due to un-synchronized concurrent executions.

- Race Conditions can occur when two or more threads ‘race’ to update the same data structure at the same time.

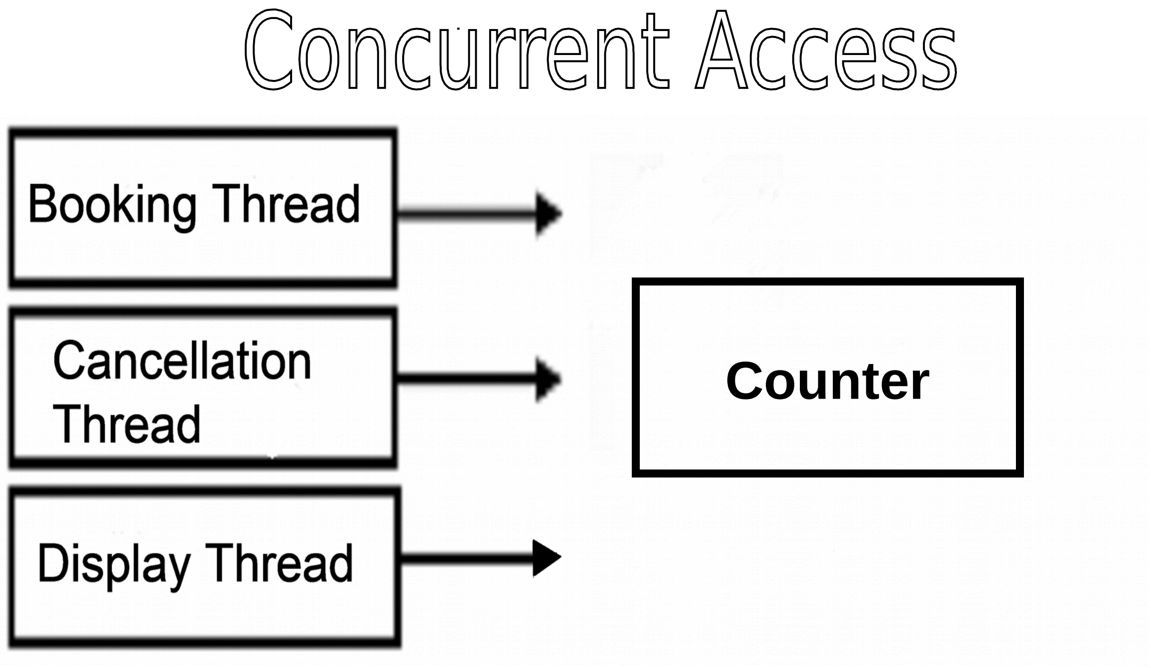
- The result can be partly what one thread wrote and partly what the other thread wrote.

- This garbles the data structu n cause the next thread that tries to use it to crash



Race Conditions

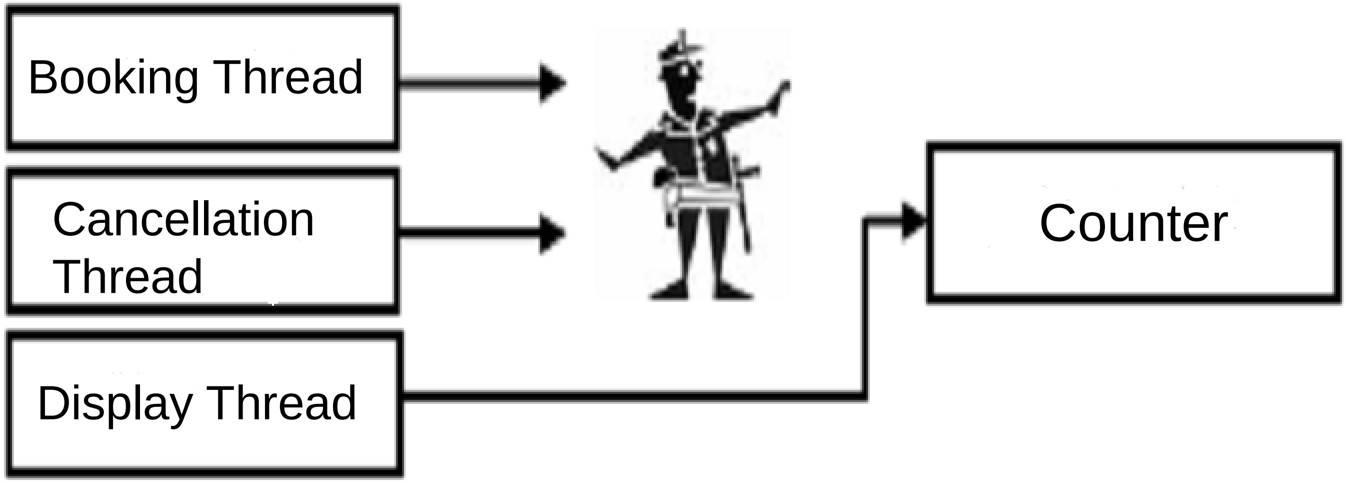
- What is the primary cause for such a race condition ?



- What is the possible work around ?

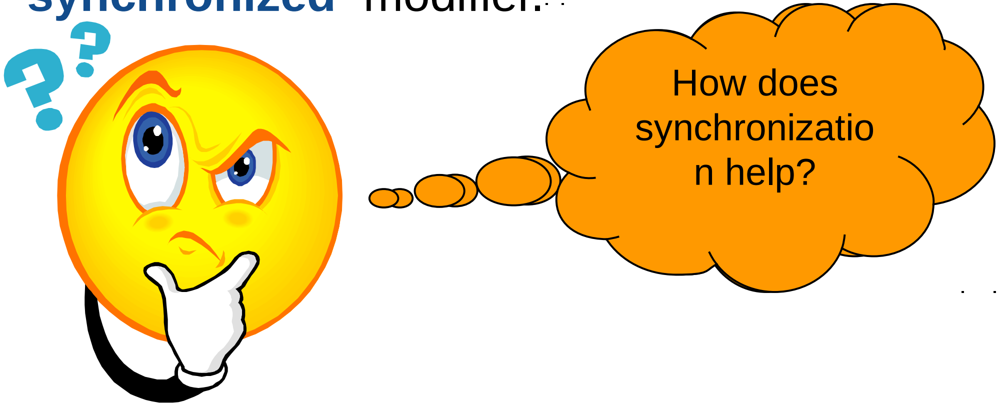
Synchronizing Access

- Synchronization makes access to the object restricted to only one thread at a time.
- The Java platform associates a lock with every object that has synchronized code.



Synchronization

- The code segments within a program that access the same object from separate, concurrent threads are called **critical sections.**
- **Synchronization** avoids race conditions by ensuring mutual exclusion to critical sections.
- Every class that has synchronized code is considered to be a **monitor**.
- A monitor operates by ensuring that at most one thread can execute the synchronized code within the object at any one time.
- A critical section can be a method or a few statements & is identified with the **'synchronized'** modifier.

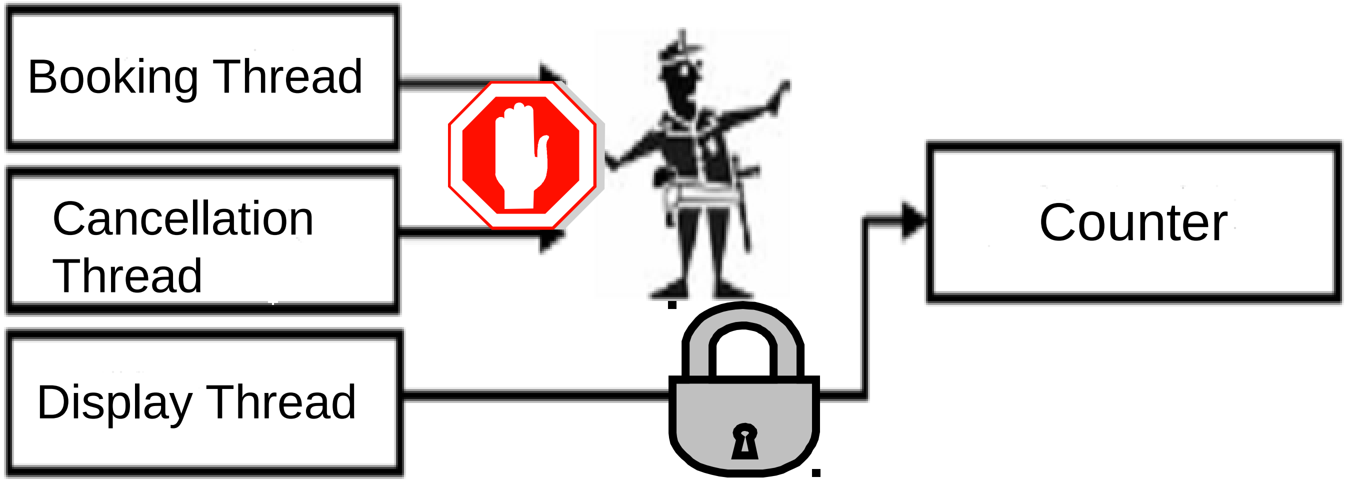


Synchronized Methods

- To make a method synchronized, the ***synchronized*** keyword is added to its declaration**.**

```
class Counter
{
    private int count = 0;
    public synchronized void increment()
    {
          count++;
    }
    public synchronized void decrement()
    {
          count--;
     }
    public synchronized int showCount()
    {
          return count;
    }
}
```

Functioning of Synchronized Methods



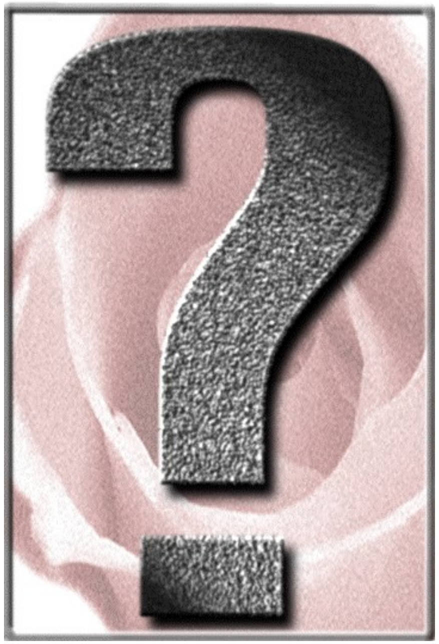
Sample Listing: SynchronizedDemo.java

Synchronized Blocks

- Unlike synchronized methods, synchronized block must specify the object on which we wish to hold the lock.

```
class Counter
{
    private int count = 0;
    public void increment()
    {
          // some code
          synchronized(this)
          {
                 count++;
          }
          // some more code
    }
}
```

Question time



- DO WE NEED THE FOLL?

Concurrent Access

```
class Counter
{
    private int count = 0;
    public void increment()
    {
          count++;
    }
    public void decrement()
    {
          count--;
     }
    public int showCount()
    {
          return count;
    }
}
```

Concurrent Access

```
class BookingThread
                  extends Thread
{
   private Counter count;
   BookingThread(Counter count)
   {
      this.count = count;
   }
   public void run()
   {
        // some logic
        if(success)
           count.decrement();
   }
}
```

```
class CancellationThread
                  extends Thread
{
   private Counter count;
   CancellationThread
                  (Counter count)
   {
      this.count = count;
   }
   public void run()
   {
       // some logic
       count.increment();
   }
}
```

Concurrent Access

```
class DisplayThread extends Thread
{
    private Counter count;
   DisplayThread(Counter count)
    {
        this.count = count;
    }
    public void run()
    {
          // some logic
          System.out.println
                          (count.showCount());
    }
}
```

```
class BookingAppDemo
{
    public static void main
              (String[] args)
    {
         Counter c = new Counter();
         BookingThread t1 = new
                       BookingThread(c);
         CancellationThread t2 = new
                    CancellationThread(c);
         DisplayThread t3 = new
                         DisplayThread();
          t1.start();
          t2.start();
          t3.start();
    }
}
```

- Scenario 1
  - BookingThread is in the process of making checks and decrementing count, just then DisplayThread accesses Counter to view count.
    - DisplayThread is not guaranteed to get the same count if BookingThread successfully processes and decrements count, leading to inconsistent data.

- Scenario 2
  - BookingThread is trying to decrement count, just then CancellationThread is concurrently invoked
    - Both threads try to make changes to Ticket count

This would result in a race condition

Synchronized Blocks

- We could lock on a portion of a method using **synchronized blocks.**
- Rather than declaring the entire method to be synchronized, a few lines of critical code can be synchronized.
- This can increase concurrency and improve performance.
- Synchronized blocks place locks for shorter periods than synchronized methods.