Core Java

Multithreading

Ameya Joshi

Email : ameya.joshi@vinsys.com

Contact : +91 9850676160

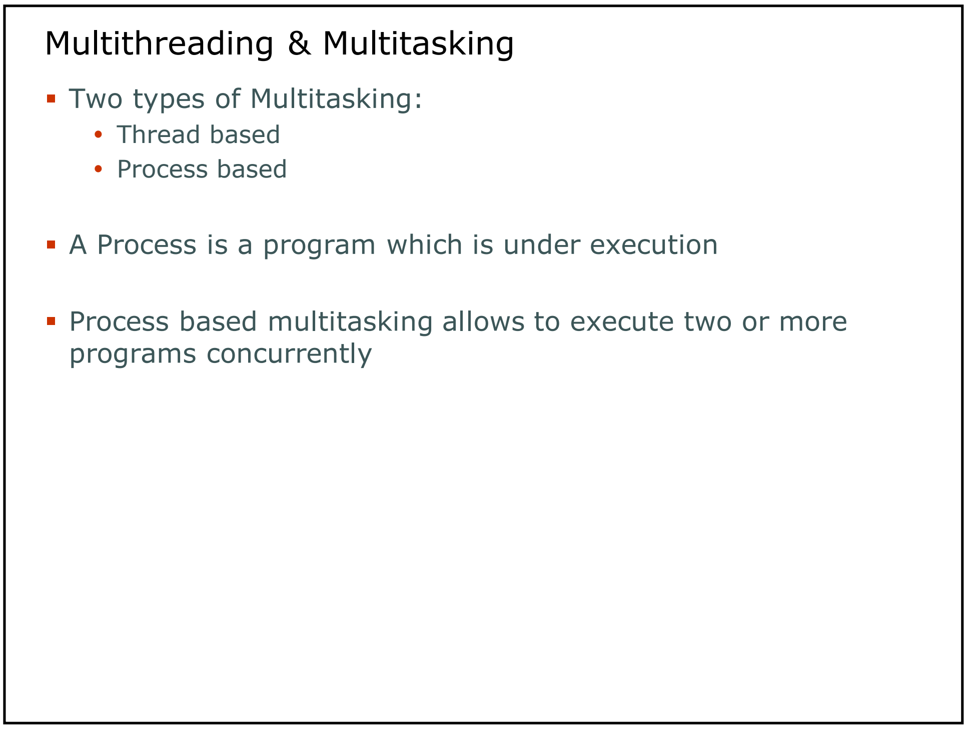
1

Introduction to Multithreading

Why Multithreading?

- Take advantage of multiprocessor systems
- Perform asynchronous or background processing
- Make the UI more responsive in case you are using some AWT or swing event toolkits.

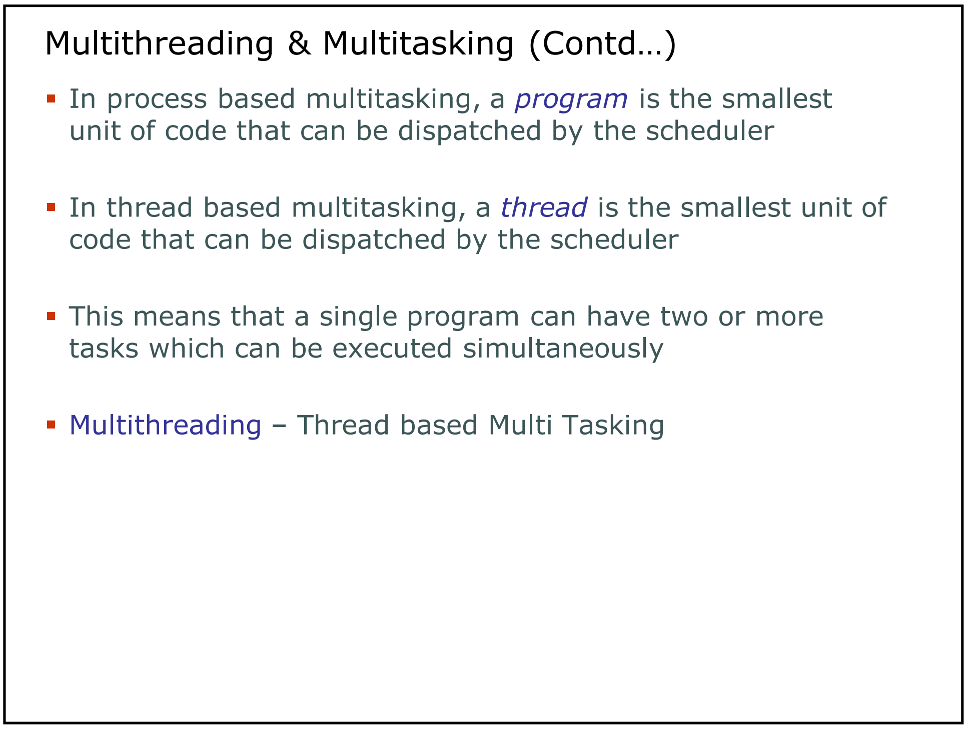
2



Multithreading & Multitasking

- Two types of Multitasking:
  - Thread based
  - Process based
- A Process is a program which is under execution
- Process based multitasking allows to execute two or more programs concurrently

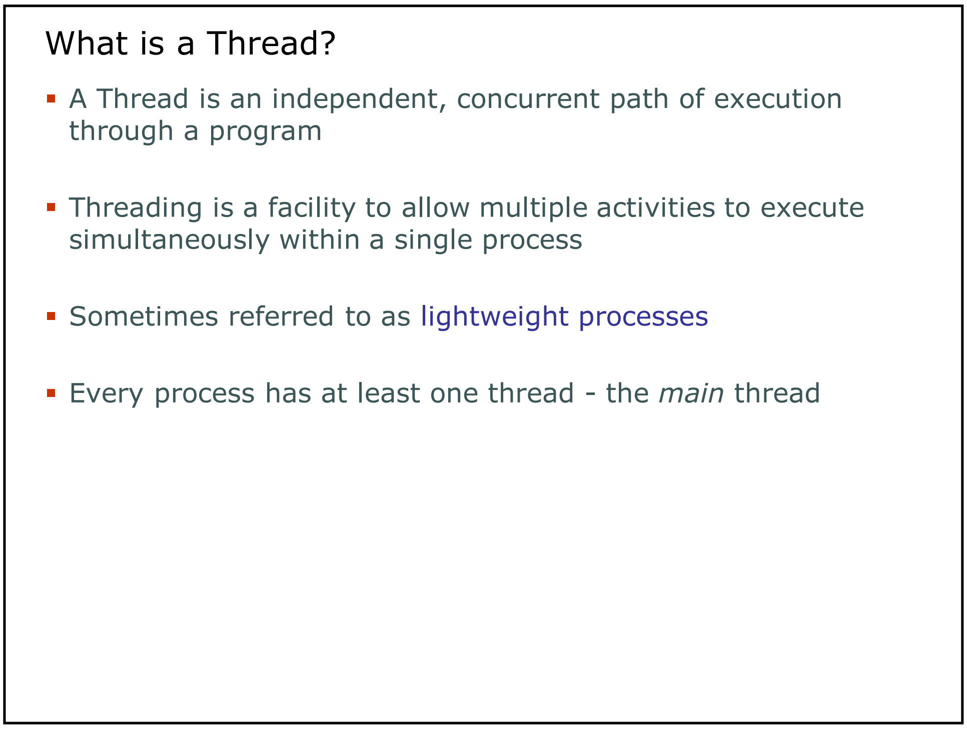
3



Multithreading & Multitasking (Contd…)

- In process based multitasking, a *program* is the smallest unit of code that can be dispatched by the scheduler
- In thread based multitasking, a *thread* is the smallest unit of code that can be dispatched by the scheduler
- This means that a single program can have two or more tasks which can be executed simultaneously
- Multithreading – Thread based Multi Tasking

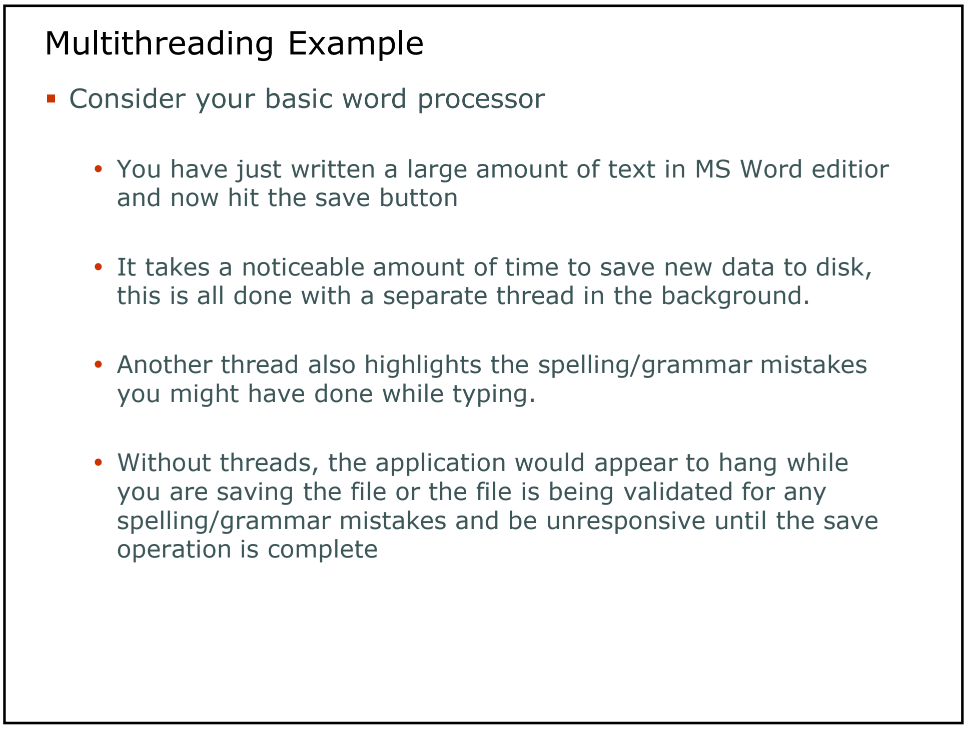
4



What is a Thread?

- A Thread is an independent, concurrent path of execution through a program
- Threading is a facility to allow multiple activities to execute simultaneously within a single process
- Sometimes referred to as lightweight processes
- Every process has at least one thread - the *main* thread

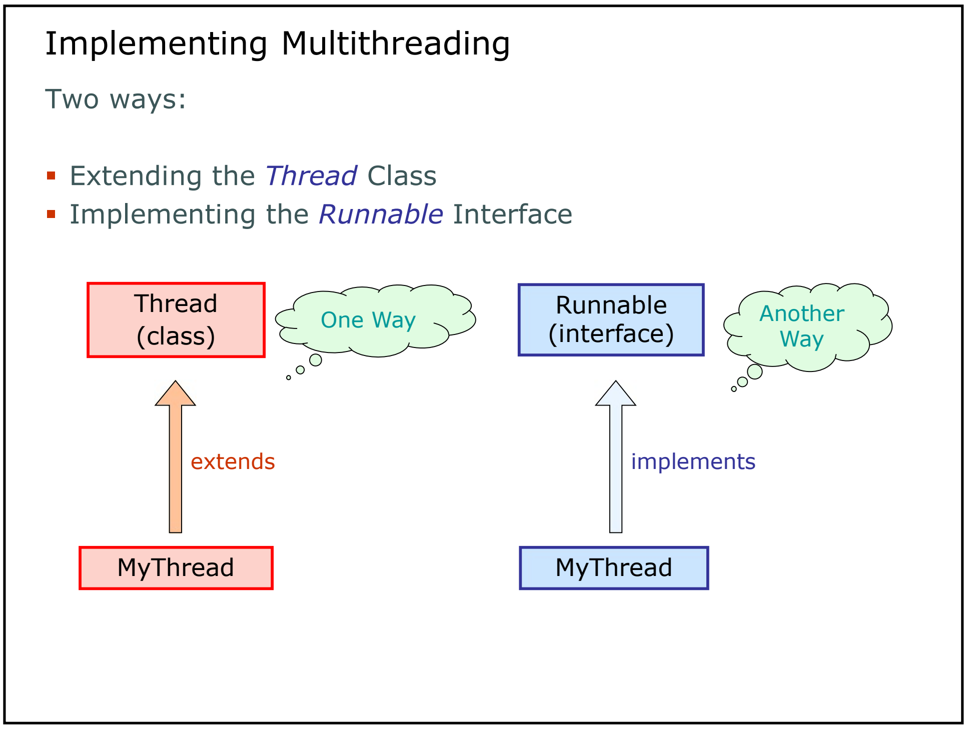
5



Multithreading Example

- Consider your basic word processor
  - You have just written a large amount of text in MS Word editior and now hit the save button
  - It takes a noticeable amount of time to save new data to disk, this is all done with a separate thread in the background.
  - Another thread also highlights the spelling/grammar mistakes you might have done while typing.
  - Without threads, the application would appear to hang while you are saving the file or the file is being validated for any spelling/grammar mistakes and be unresponsive until the save operation is complete

6



Implementing Multithreading

Two ways:

- Extending the *Thread* Class
- Implementing the *Runnable* Interface

Thread (class)

One Way

extends

MyThread

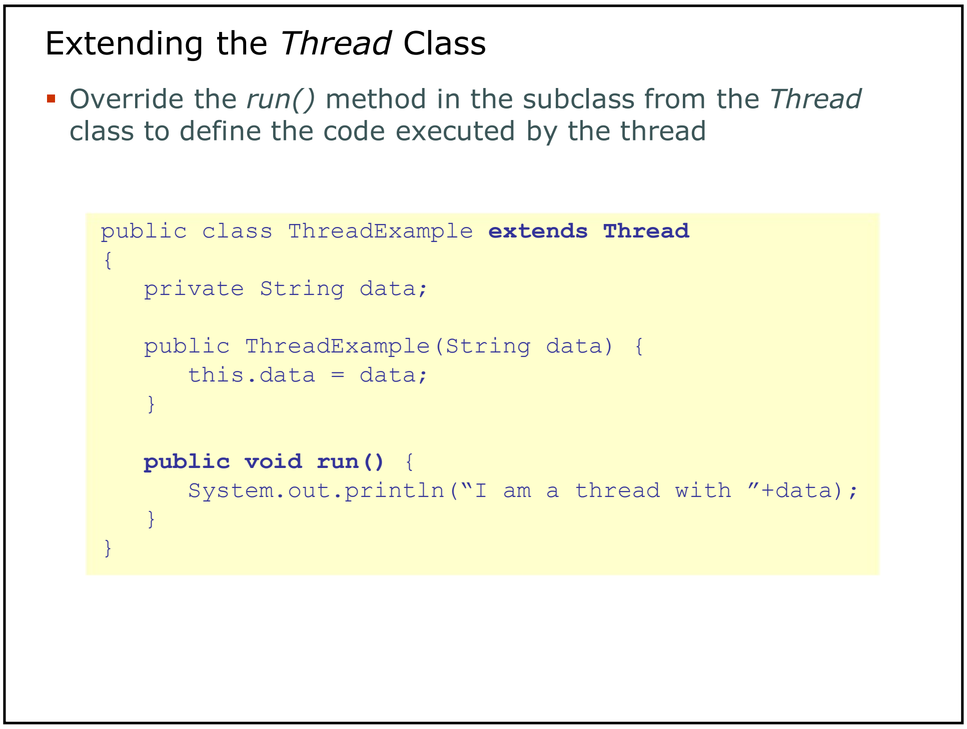
Runnable (interface)

Another Way

implements

MyThread

7



Extending the *Thread* Class

- Override the *run()* method in the subclass from the *Thread* class to define the code executed by the thread

```
public class ThreadExample extends Thread
{
   private String data;

   public ThreadExample(String data) {
      this.data = data;
   }

   public void run() {
      System.out.println(“I am a thread with ”+data);
   }
}
```

8

Running Threads

- Create an instance of this subclass (ThreadExample)
- Invoke the *start()* method on the instance of the class to make the thread eligible for running

```
public class ThreadExampleMain
{
   public static void main(String[] args) {
      Thread myThread = new ThreadExample("my data");
      myThread.start();
   }
}
```

Using the *Runnable* Interface

- Why this approach is required?
- Implement the *Runnable* interface
- Override the *run()* method to define the code executed by thread

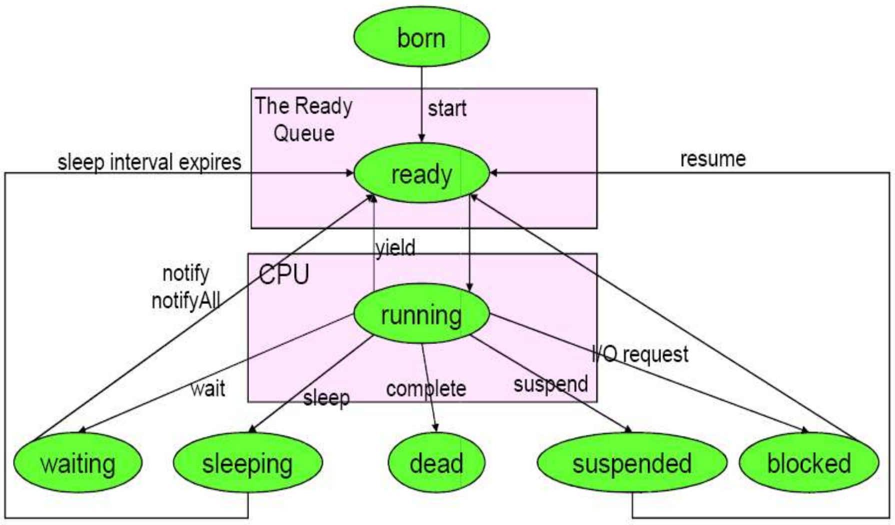
```
public class RunnableExample extends SomeClass
        implements Runnable
{
   private String data;
   public RunnableExample(String data) {
      this.data = data;
   }
   public void run() {
      System.out.println("I am a thread: "+data);
   }
}
```

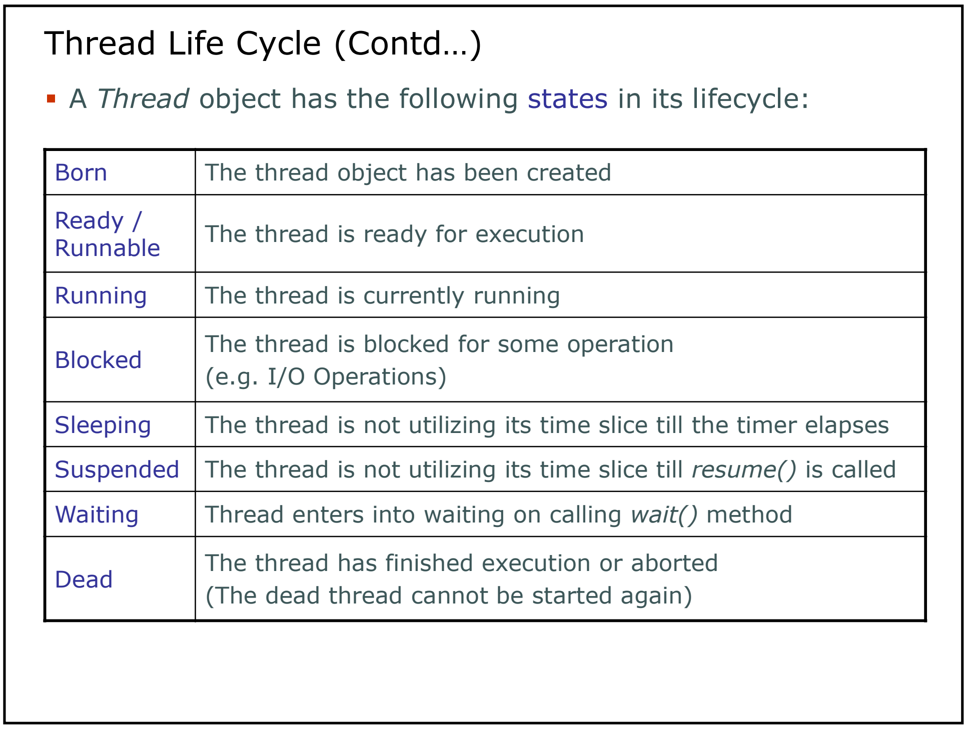
Using the *Runnable* Interface (Contd…)

- Create an object of *Thread* class
- Invoke the *start()* method on the instance of the *Thread* class

```
public class RunnableExampleMain
{
   public static void main(String[] args) {
      RunnableExample myRunnableObject = new
             RunnableExample(“my data”);
      Thread myThread = new Thread(myRunnableObject);
      myThread.start();
   }
}
```

Thread Life Cycle

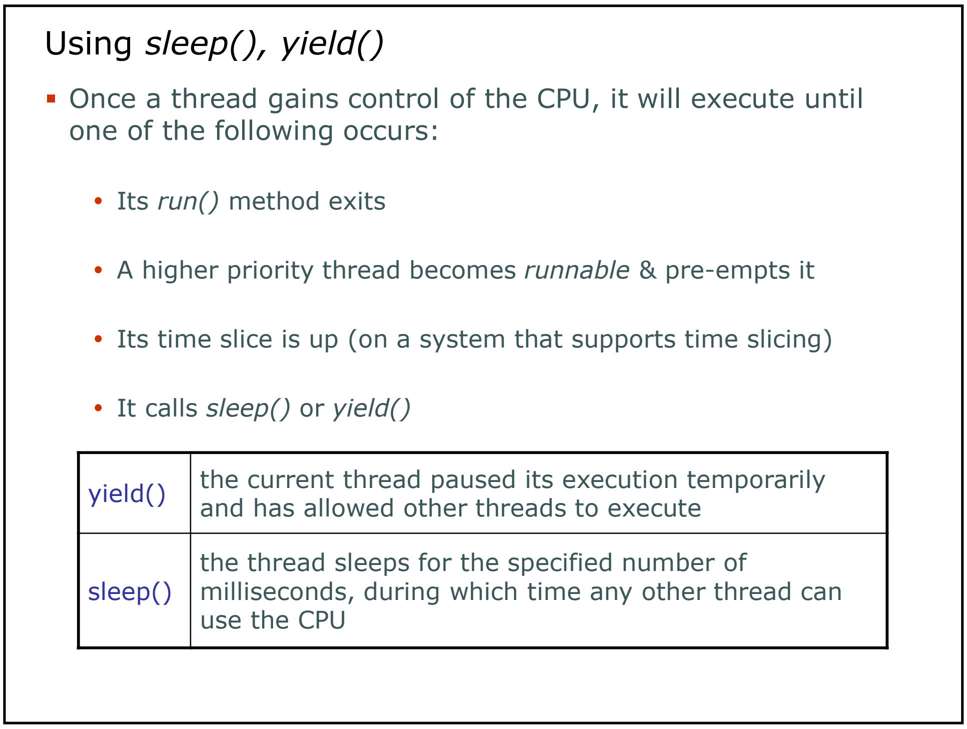




Thread Life Cycle (Contd…)

- A *Thread* object has the following states in its lifecycle:

Born	The thread object has been created
Ready / Runnable	The thread is ready for execution
Running	The thread is currently running
Blocked	The thread is blocked for some operation (e.g. I/O Operations)
Sleeping	The thread is not utilizing its time slice till the timer elapses
Suspended	The thread is not utilizing its time slice till *resume()* is called
Waiting	Thread enters into waiting on calling *wait()* method
Dead	The thread has finished execution or aborted (The dead thread cannot be started again)



Using *sleep(), yield()*

- Once a thread gains control of the CPU, it will execute until one of the following occurs:
  - Its *run()* method exits
  - A higher priority thread becomes *runnable* & pre-empts it
  - Its time slice is up (on a system that supports time slicing)
  - It calls *sleep()* or *yield()*

yield()	the current thread paused its execution temporarily and has allowed other threads to execute
sleep()	the thread sleeps for the specified number of milliseconds, during which time any other thread can use the CPU

Using *join()*

- A call to the *join* method on a specific thread causes the current thread to block until that specific thread is completed

```
public class ThreadExampleMain
{
   public static void main(String[] args) {
      Thread myThread = new ThreadExample(“my data”);
      myThread.start();
      System.out.println(“I am the main thread”);

      myThread.join();
      System.out.println(“waiting for myThread”);
   }
}
```

15

InterruptedException

- Thrown when a thread is waiting, sleeping, or otherwise occupied, and the thread is interrupted, either before or during the activity.

16

Thread Priorities

- The JVM chooses which thread to run according to a “fixed priority algorithm”
- Every thread has a priority between the range of *Thread.MIN_PRIORITY(1)* and *Thread.MAX_PRIORITY(10)*
- By default a thread is instantiated with the same priority as that of the thread that created it
- Thread priority can be changed using the *setPriority()* method of the Thread class

Thread Priorities (Contd…)

- Thread priority can be obtained using the *getPriority()* method of the Thread class
- Threads with higher priorities are scheduled before threads with lower priorities.
- Threads with higher priorities will get more CPU time than the threads with lower priorities.
- The algorithm is *preemptive*, so if a lower priority thread is running, and a higher priority thread becomes runnable, the high priority thread will pre-empt the lower priority thread

Synchronization

- Sometimes, multiple threads may be accessing the same resources concurrently
  - Reading and / or writing the same file
  - Modifying the same object / variable
- Such a resource can be termed as a shared resource.
- Synchronization controls thread execution order
- Synchronization eliminates data races

19

Synchronization (Cotd..)

When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time.

This can be achieved by process known as synchronization.
Key to synchronization is the concept of monitor .

Monitor :
A monitor is an object that is used as a mutually exclusive lock, or mutex.

Only one thread can ***own*** a monitor at a given time. When a thread acquires a lock, it is said to have ***entered*** the monitor.

All other threads attempting to enter the locked monitor will be suspended until the first thread ***exits*** the monitor

20

Synchronization (Cotd..)

synchronized keyword :

In Java the code can be synchronized in either of the two ways. Both use the ***synchronized*** keyword.

1. Writing synchronized methods	2.Writing synchronized blocks

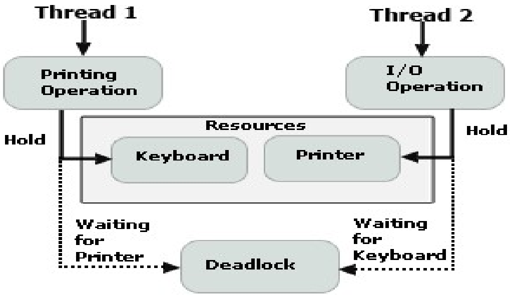
synchronized void methodName()
{
//body of the method
}
	synchronized(object){
	//statements to be
	//synchronized.
	}

Inter-Thread Communication

- When multiple threads are using a shared resource, The resources are synchronized.
- This might lead to undesired or inconsistent behaviour of threads.
- Consider a typical Producer Consumer analogy, where a Producer might just go on producing without the items being consumed. Or a consumer just tries to consume the items that might not have been produced. Either case is undesired.
- Inter-Thread communication avoids such scenarios.
- This can be achieved using wait(), notify(), notifyAll()

Thread Deadlock

- If a thread is waiting for an object lock held by the second thread
- The second thread is waiting for an object lock held by the first one
- Example: 2 threads having printing & I/O operations respectively at a time
  - Thread1 needs a printer which is held by Thread2
  - Thread2 needs the keyboard which is held by Thread1



Thank You