Advanced Java

Multithreading

Objective

At the end of this session, you will be able to:

- Identify need of multi threading
- Write simple programs using Multithreading
- Write multi threaded programs with Thread Synchronization
- Identify & Prevent Deadlocks in multi threaded programs

Agenda

- Introduction to Multithreading
- Implementing Multithreading
- Thread Life Cycle
- Using *sleep(), yield(), join()*
- Thread priorities
- Thread Synchronization
- Using *wait()* & *notify()*
- Deadlocks

Introduction to Multithreading

Why Multithreading?

- Make the UI more responsive
- Take advantage of multiprocessor systems
- Simplify modeling in simulation application
- Perform asynchronous or background processing

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

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

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

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

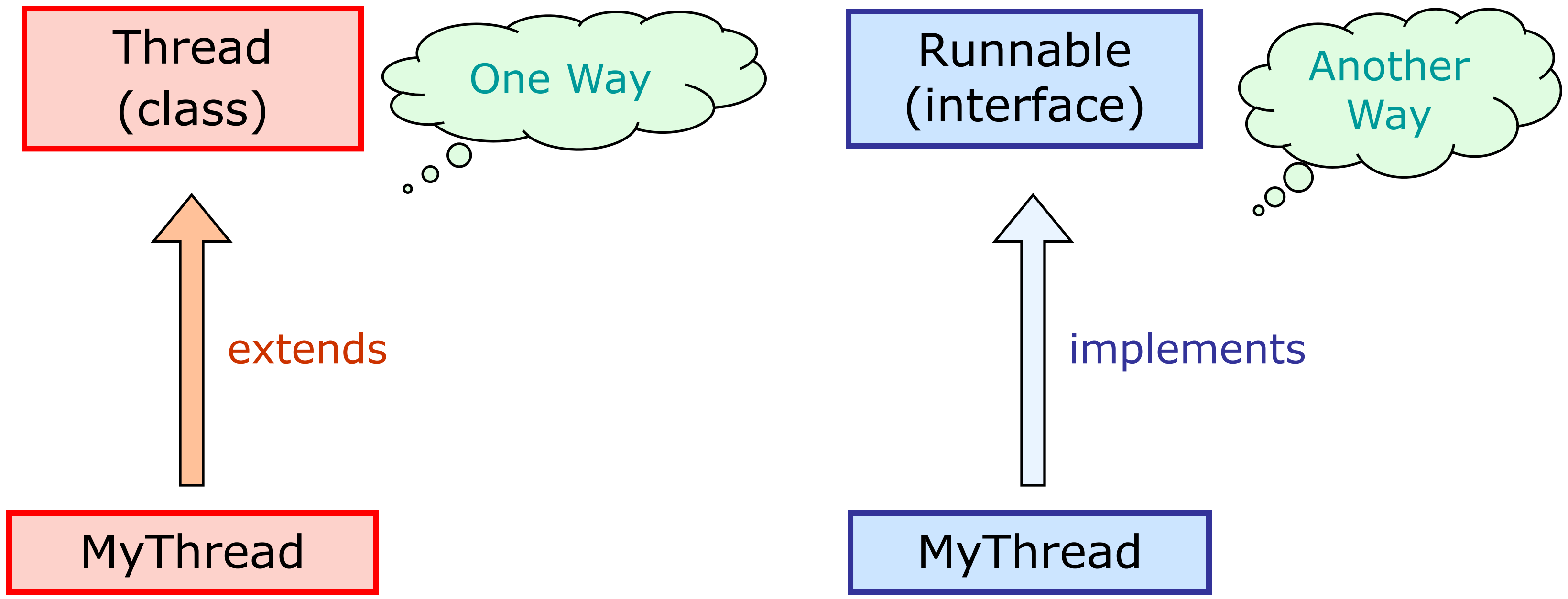
  It takes a noticeable amount of time to save new data to disk, this is all done with a separate thread in the background

  Without threads, the application would appear to hang while you are saving the file and be unresponsive until the save operation is complete

Implementing Multithreading

Two ways:

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



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);
   }
}
```

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();
   }
}
```

RunThread.java

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();
  }
}
```

RunnableThread.java

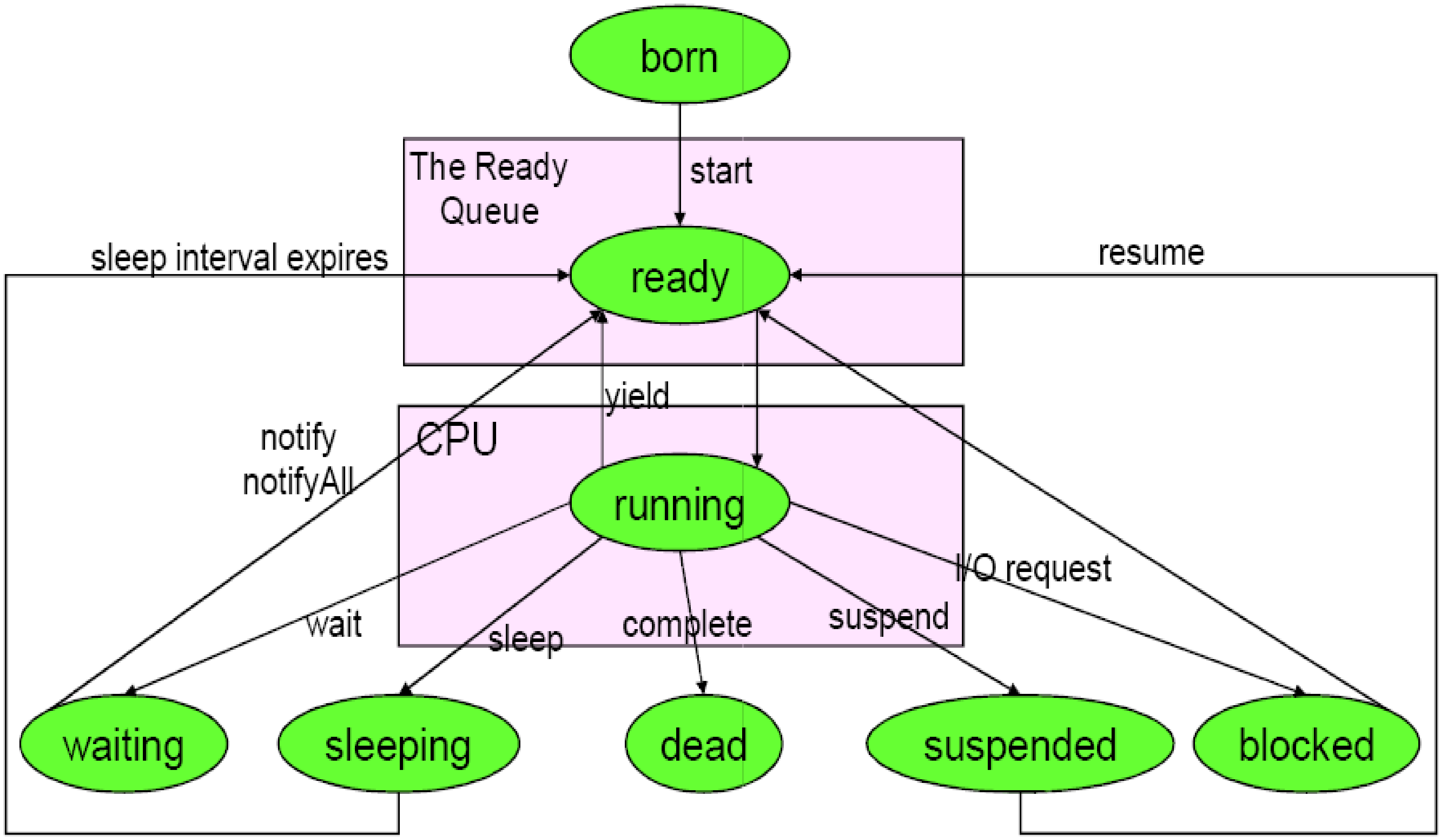
Give this a Try…

1. Whenever we type a letter in MS-Word & simultaneously listen a song in Winamp, is it an example of Multithreading?

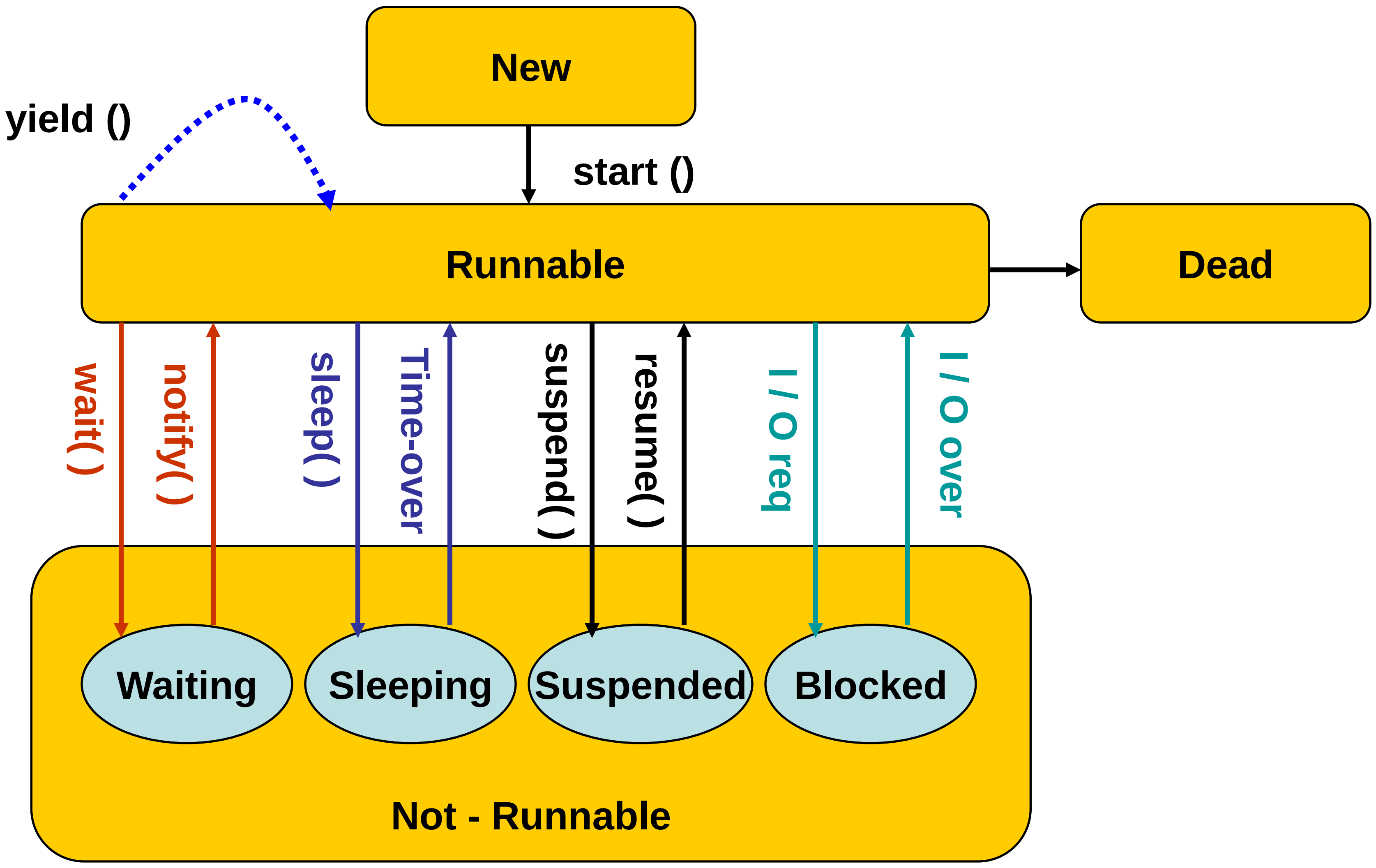
2. What is the signature of the *run()* method?

3. Which is the interface that should be implemented for Multithreading?

Thread Life Cycle



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 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”);
   }
}
```

Other *Thread* Operations

- *interrupt()* method

  Interrupts the thread on which it is invoked

- *interrupted()* method

  Returns a boolean value indicating the interrupted status of the current thread & resets the same to not interrupted

- A combination of these two methods is useful for requesting a thread to yield, sleep or terminate

Give this a Try

1. After executing run() method completely, thread enters in which state?
2. If thread us running, and sleep() method is invoked, it enters in which state?
3. If thread is running, & there is some I/O operations in the program, thread enters in which state?

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 run to completion before Threads with lower priorities get a chance of CPU time
- 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

ThreadPriority.java

Synchronization

- Sometimes, multiple threads may be accessing the same resources concurrently
  - Reading and / or writing the same file
  - Modifying the same object / variable
- Synchronization controls thread execution order
- Synchronization eliminates data races
- Java has built in primitives to facilitate this coordination

Synchronization (Contd…)

- Producer / Consumer Example:

  There are 2 threads, a *producer* & a *consumer,* both accessing the same object of type *CubbyHole*

  CubbyHole is a simple object that holds a single value as its contents

  The *producer* thread randomly generates values and stores them in the *CubbyHole* object

  The *consumer* then retrieves these values as they are generated by the producer

Synchronization: Producer

```
public class Producer extends Thread
{
   private CubbyHole cubbyhole;
   public Producer(CubbyHole c) {
      cubbyhole = c;
         }
   public void run() {
      for (int i = 0; i < 10; i++) {
         cubbyhole.put(i);
         try {
            sleep((int)(Math.random() * 100));
         } catch (InterruptedException e) { }
      }
   }
}
```

Synchronization: Consumer

```
public class Consumer extends Thread
{
   private CubbyHole cubbyhole;
      public Consumer(CubbyHole c) {
      cubbyhole = c;
      }
   public void run() {
      int value = 0;
      for (int i = 0; i < 10; i++) {
         value = cubbyhole.get();
      }
   }
}
```

Synchronization: Problems

- When the *producer* produces a value, it stores it in the *CubbyHole,* and then the *consumer* is only supposed to retrieve it, once and only once

- Depending on how the threads are scheduled:
  - The *producer* could produce two values before the consumer is able to retrieve one
  - The *consumer* could consume the same value twice before the producer has produced the next value

- If the *producer* & *consumer* access the *cubbyhole* at the same time, they could produce an inconsistent state or miss a produced value

Synchronization: Solution

```
public class CubbyHole
{
   private int contents;
   private boolean available = false;

   public synchronized int get()
   { ... }

   public synchronized void put(int value)
   { ... }
}
```

- When a thread executes a synchronized method, it locks the object of the method
- No synchronized methods can be called by another thread on that object until it is unlocked

Synchronization: Wait / Notify

- *Synchronized* methods stop the *producer* & *consumer* from modifying the *CubbyHole* at the same time

```
public synchronized int get() {
   while (available == false) {
      try {
         //wait for Producer to put value
         wait();
      } catch (InterruptedException e) { }
   }
   available = false;
   //notify Producer that value has been retrieved
   notifyAll();
   return contents;
}
```

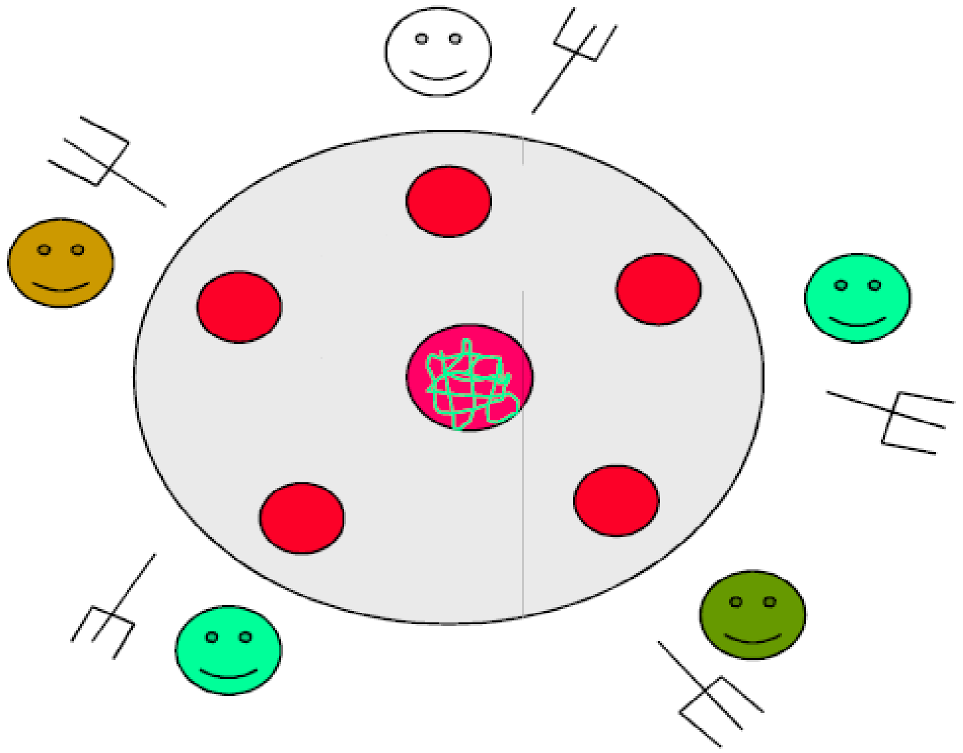
Give this a Try…

1. Method can be declared as synchronized. TRUE/FALSE
2. Block can be declared as synchronized. TRUE/FALSE
3. Complete Class can be declared as synchronized. State True / False

Thread Deadlock

The Dining Philosopher Problem

- All philosophers are holding a single fork & are waiting for each other



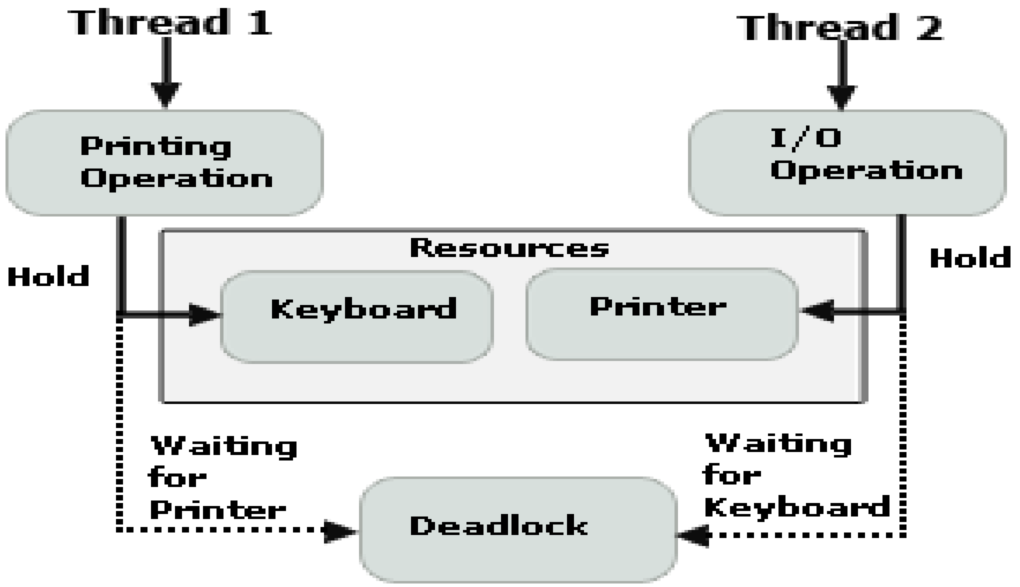
Java multithreading has a provision to solve this problem

Thread Deadlock (Contd…)

- 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



Thread Deadlock (Contd…)

- A Deadlock can be prevented, but if it has occurred, the program is dead

DeadLockDemo.java

Give this a Try…

1. Whenever deadlock occurs in the program, what happens?

Summary

In this session, we have covered:

- Introduction to Multithreading
- Implementing Multithreading
- Thread Life Cycle
- Using *sleep(), yield(), join()*
- Thread Priorities
- Thread Synchronization
- Using *wait()* & *notify()*
- Deadlocks

Thank You