Session 7

Multithreading

Outline

- Multithreaded programming in Java
- Multithreading: advantages and issues
- The Thread class, thread groups
- The Runnable interface
- Thread synchronization
- Inter-Thread communication

Process vs. Thread

- **Process**
  - A *process* is a program executing within its own address space. In case of multitasking, for example, running Microsoft Word along with Internet Explorer, Windows Explorer, Inbox, CD Player, etc.
  - The applications are all processes executing within the Windows environment.
  - Can think applications, or stand-alone programs; each process in a system is given its own room to execute.

- **Thread**
  - A *thread* is a sequence of code executing within the context of a process.
  - You can think of each thread as running in a separate context: e.g., Word, there are many threads running in the background automatically, i.e., checking the spelling, saving automatically changes to the document.
  - The significance here is that threads are always associated with a particular process.

Multitasking vs. Multithreading

Multitasking

– allows several processes to run concurrently on the computer.

Multithreading

– allows parts of the same program to run concurrently on the computer.

- *Multithreading* extends the idea of *multitasking* by taking it one level lower.
- Programs that can run more than one thread at once are said to be *multithreaded*.

Advantages of Multithreading

- Threads share the same address space
- Context switching between threads is usually less expensive than between processes
- Cost of communication between threads is relatively low

Overview of Java Threads

- A thread is an independent sequential path of execution within a program.
- Many threads can run in concurrently within a program.
- At runtime, threads in a program exist in a common memory space and can, therefore, share both data and code, that is they are ***lightweight*** compared to processed.
- Every Thread in Java is created and controlled by a unique object of the **java.lang.Thread** class.

Overview of Java Threads (cont.)

The runtime environment distinguishes:

- **user thread**
  - Created to execute user application.

- **daemon thread**
  - Exists only to serve user threads.

As long as a user thread is alive, the JVM does not terminate the application.

Overview of Java Threads (cont.)

- When a standalone application is run, a user thread is automatically created to execute the main method. This thread is called ***main thread***.
- If no other user threads are spawned, the program terminates when the main() method finishes executing.
- All other threads, called ***child threads***, are spawned from the main thread.
- At that time, the main method can finish, but the program will keep running until all the user threads have finished.

Overview of Java Threads (cont.)

- When a GUI application is started, a special thread is automatically created to monitor the user-GUI interaction.

- This user thread keeps the program running , allowing interaction between the user and the GUI, even though the main thread might have died after the main method finishes executing.

Thread Classes

The Java programming language provides support for threads through a single interface and a handful of classes.

The Java interface and classes that include thread functionality follow:

- Thread
- Runnable
- ThreadDeath
- ThreadGroup
- Object

Notes : *All of these classes are part of the* ***java.lang*** *package.*

Thread Creation

There are two options for creating and using threads in the Java programs as follows:

- Deriving **Thread** class override its ***run method***.

- Implementing the **Runnable** interface in your class and provide an implementation for the ***run method*.**

Both of these approaches revolve around providing a ***run() method***, which is where *all the actual processing takes place*.

Deriving from the Thread Class

- If your class isn't derived from a specific class, you can easily make it threaded by deriving it from the Thread class.

Syntax:

```
public class ThreadMe extends Thread
{
    public void run()
    {
        //thread body
        // do some work
    }
}
```

Creating and Running Thread

To use this class in a real program and set the thread in motion,

```
ThreadMe me = new ThreadMe();
me.start();
```

The start method automatically calls run and gets the thread busy performing its processing.

To stop the thread,

```
me.stop();
```

To suspend the thread,

```
me.suspend();
```

To resume the thread,

```
me.resume();
```

Example using Thread class

```
class Counter extends Thread
{
    private int currentValue;
    public Counter(String threadName)
    {
        super(threadName);
        currentValue = 0;
    }
    public int getValue()
    {
        return currentValue;
    }
```

```
public void run()
{
  try {
        while (currentValue < 5)
        {
            System.out.println(getName()
            + ": " + (currentValue++));
            Thread.sleep(250);
        }
    } catch (InterruptedException e)
    {
        System.out.println(getName()
        + " interrupted.");
    }
    System.out.println("Exit from
thread: " + getName());
}}
```

Example (cont.)

```
public class Client
{
    public static void main(String[] args)
    {
        System.out.println("Method main() runs in thread " +
        Thread.currentThread().getName());
        Counter  counterA = new Counter("Counter A");
        Counter  counterB = new Counter("Counter B");
        counterA .start();
        counterB .start();
        System.out.println("Exit from main() method.");
    }
}
```

Possible Output

```
Method main() runs in thread main
Exit from main() method.
Counter A: 0
Counter B: 0
Counter A: 1
Counter B: 1
Counter A: 2
Counter B: 2
Counter A: 3
Counter B: 3
Counter A: 4
Counter B: 4
Exit from thread: Counter A
Exit from thread: Counter B
```

Implementing Runnable Interface

- If your class needs to derive from a class other than Thread, you are forced to implement the Runnable interface to make it threaded.

Syntax:

```
public class ThreadYou implements Runnable
{
    public void run()
    { // thread body
      // do some busy work
    }
}
```

Creating and Running Thread

- Creating and running a threaded class implementing the Runnable interface is a three-part process, as the following code shows:

```
ThreadYou you = new ThreadYou();
Thread t = new Thread(you);
t.start();
```

(OR)

```
ThreadYou you = new ThreadYou();
new Thread(you).start();
```

Example using **Runnable** interface

```
class Counter implements Runnable
{
    private int currentValue;
    private Thread worker;
    public Counter(String threadName)
    {
        currentValue = 0;
        worker = new Thread(this,
                                threadName);
        worker.start();
    }

    public int getValue()
    { return currentValue; }
```

```
public void run() {
  try {
        while (currentValue < 5)
        {
System.out.println(worker.getName() +
": " + (currentValue++));
            Thread.sleep(250);
        }
} catch (InterruptedException e)
{
   System.out.println(worker.getName() +
" interrupted.");
  }
  System.out.println("Exit from thread: "
+ worker.getName());
}
}
```

Example (cont.)

```
public class Client
{   public static void main(String[] args)
    { Counter counterA = new Counter("Counter A");
       try {
            int val;
            do {
                 val = counterA.getValue();
                 System.out.println("Counter value read by main thread: " + val);
                Thread.sleep(1000);
              } while (val < 5);
         } catch (InterruptedException e) {
                 System.out.println("main thread interrupted.");}
    System.out.println("Exit from main() method.");    }
}
```

Possible Output

```
Counter value read by main thread: 0
Counter A: 0
Counter A: 1
Counter A: 2
Counter A: 3
Counter value read by main thread: 4
Counter A: 4
Exit from thread: Counter A
Counter value read by main thread: 5
Exit from main() method.
```

Thread Synchronization

- Threads share the same memory space, i.e, they can share resources.
- However, there are critical situations where it is desirable that only one thread at a time has access to a shared resource.
  - E.g., crediting and debiting a shared bank account
- Java provides high-level concepts for synchronization.
  - **Lock**
    - associated with a shared resource
    - use to synchronize a shared resource
    - Implements mutual exclusion
- **In Java, all objects have a lock including arrays.**

Thread Synchronization

Object Lock Mechanism

(1) A thread must acquire the object lock before it can enter the shared resource.

(2) When a thread exits a shared resource, the runtime system ensures that the object lock is relinquished.

- There are two ways in which execution of code can be synchronized:
  - synchronized methods
  - synchronized blocks

Example : Without Synchronization

```
class StackImpl {
    private Object[] stackArray;
    private int topOfStack;
    public StackImpl(int capacity) {
        stackArray = new Object[capacity];
        topOfStack = -1;
    }

    public boolean push(Object element) {
            if (isFull()) return false;
            ++topOfStack;
        stackArray[topOfStack] = element;
        return true;
}
```

Example : Without Synchronization

```
public Object pop() {
    if (isEmpty()) return null;
    Object obj = stackArray[topOfStack];
    stackArray[topOfStack] = null;
    topOfStack--;
    return obj;
}
    public boolean isEmpty()
    { return topOfStack < 0; }

    public boolean isFull()
    { return topOfStack >= stackArray.length - 1; }
}
```

Example : Without Synchronization

```
public class Mutex {
public static void main(String[] args) {
    final StackImpl stack = new StackImpl(20);
    (new Thread("Pusher") {  public void run()
                                {for(;;) {System.out.println("Pushed: " +
                                                stack.push("2008")); } } }
                            }).start();
    (new Thread("Popper") { public void run()
                                {for(;;) {System.out.println("Popped: " +
                                                stack.pop());}    }
                            }).start();
    System.out.println("Exit from main().");
    }
}
```

Possible Output

```
Exit from main().
...
Pushed: true
Popped: 2003
Popped: 2003
Popped: null
...
Popped: null
java.lang.ArrayIndexOutOfBoundsException: -1
at StackImpl.push(Mutex.java:15)
at Mutex$1.run(Mutex.java:41)
Popped: null
Popped: null
...
```

Example : Synchronized Methods

```
class StackImpl {
    private Object[] stackArray;
    private int topOfStack;
    public StackImpl(int capacity) {
        stackArray = new Object[capacity];
        topOfStack = -1;
    }
    public synchronized boolean push(Object element) {
        if (isFull()) return false;
            ++topOfStack;
        stackArray[topOfStack] = element;
        return true;
}
```

Example : Synchronized Methods

```
public synchronized Object pop() {
    if (isEmpty()) return null;
    Object obj = stackArray[topOfStack];
    stackArray[topOfStack] = null;
    topOfStack--;
    return obj;
}
    public boolean isEmpty()
    { return topOfStack < 0; }

    public boolean isFull()
    { return topOfStack >= stackArray.length - 1; }
}
```

Example : Synchronized Methods

```
public class Mutex {
public static void main(String[] args) {
    final StackImpl stack = new StackImpl(20);
    (new Thread("Pusher") {  public void run()
                                {for(;;) {System.out.println("Pushed: " +
                                                    stack.push(“2008”)); }   }
                            }).start();
    (new Thread("Popper") { public void run()
                                {for(;;) {System.out.println("Popped: " +
                                                    stack.pop());}    }
                            }).start();
    System.out.println("Exit from main().");
    }
}
```

Possible Output

Exit from main().

Pushed: true

Pushed: true

Pushed: true

Pushed: true

Popped: 2008

Popped: 2008

Pushed: true

Pushed: true

…

Synchronized Blocks

Syntax:

Synchronized (<object reference expression>) {<code block>}

e.g.,

```
public Object pop() {
    synchronized (this)
    {     // Synchronized block on current object
          // ...
    }
}
```

Thread States

The *state* of a thread defines its current mode of operation, such as whether it is running or not.

Following is a list of the Java thread states:

- **New**
- **Runable (Ready-to-run)**
- **Running**
- **Blocked (Not-running)**
- **Dead**

Thread States : New

- A thread is in the *"new"* state when it is first created until its *start* method is called.
- New threads are already initialized and ready to get to work, but they haven't been given the cue to take off and get busy.

Thread States : Ready-To-Run

- When the *start* method is called on a new thread, the *run* method is in turn called and the thread enters the *"runable"* state.

Thread States : Blocked

Following is a list of the different possible events that can cause a thread to be temporarily halted:

- The suspend method is called.
- The sleep method is called.
- The wait method is called.
- The thread is blocking for I/O.

Following is a list of the corresponding events that can put a thread back in the "runnable" state:

- If a thread is suspended, the resume method is called.
- If a thread is sleeping, the number of specified milliseconds elapses.
- If a thread is waiting, the object owning the condition variable calls notify or notifyAll.
- If a thread is blocking for I/O, the I/O operation finishes.

Thread States : Dead

A thread can enter the "dead" state through one of two approaches, which follow:

- The run method finishes executing.
- The stop method is called.

Thread Priorities

- Threads are assigned priorities that the scheduler can use to determine how the threads will be scheduled.
- Thread priorities are defined as integers between 1 and 10.
- Ten is the highest priority. One is the lowest. The normal priority is five.
- Higher priority threads get more CPU time.
- A thread inherits the priority of its parent thread.

Two methods in Thread class

setPriority(int priority)

getPriority()

Static variables in Thread class

Thread.MIN_PRIORITY

Thread.MAX_PRIORITY

Thread.NORM_PRIORITY

Example : Using Thread Priorities

```
MyThread  frank= new MyThread ("Frank");
MyThread  mary= new MyThread (“Mary");
MyThread  chris= new MyThread (“Chris");


frank.setPriority(Thread.MIN_PRIORITY);
mary.setPriority(Thread.NORM_PRIORITY);
chris.setPriority(Thread.MAX_PRIORITY);


frank.start();
mary.start();
chris.start();
```

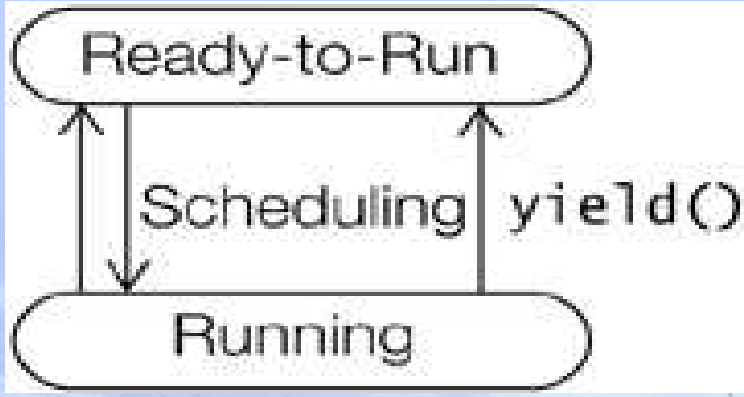
Thread Scheduler

Schedulers in JVM implementations usually employ one of the two following strategies:

- Preemptive Scheduling

- Time-Sliced or Round Robin Scheduling

Running and Yielding



void start()

- enters the Thread into runable state and waits for its turn to get CPU time. The scheduler decides for it.

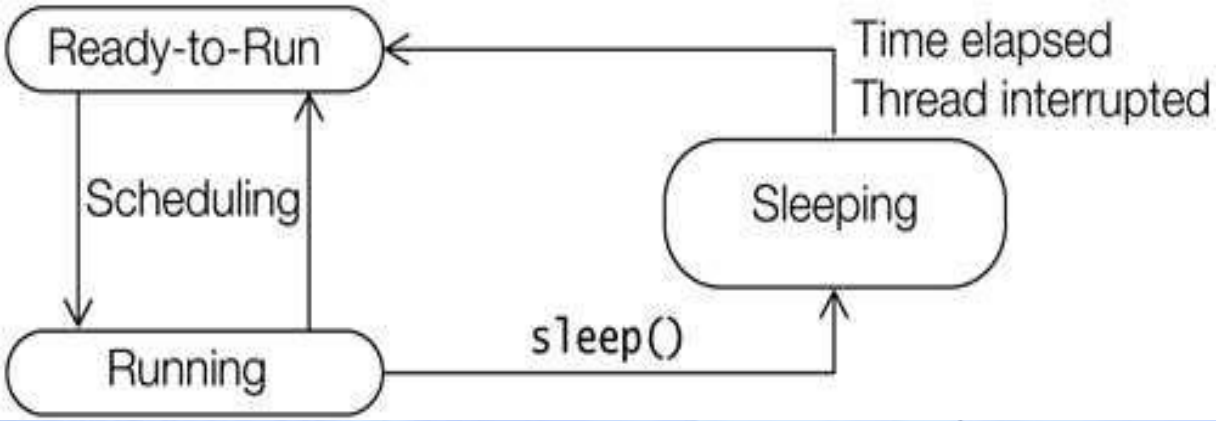
static void yield()

- will cause the current thread in the running state to transit to the ready-to-run state, thus relinquishing CPU.
- If there are no threads in ready state, this thread will continue executing.

Example

```
public void run()
{
    try {
        while (!done()) {
                doLittleBitMore();
                Thread.yield(); // Current thread yields
            }
        } catch (InterruptedException e) {
            doCleaningUp();
        }
}
```

Sleeping and Waking up

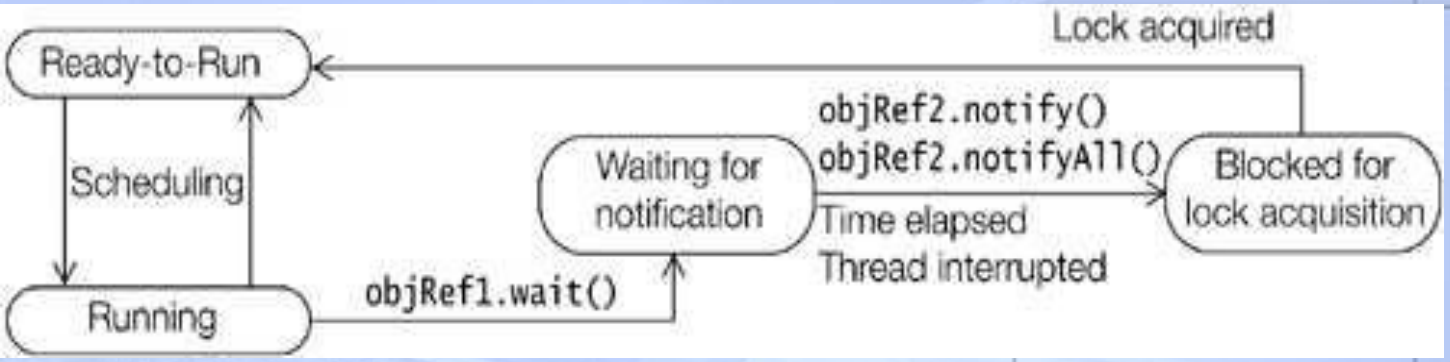


static void sleep(time)

- Will cause the currently running thread to pause its execution and transit to the sleeping state.
- Does not relinquish any lock.

Waiting and Notifying

Provides communication between threads that synchronize the same objects.



These methods can only be executed on an object whose lock the Thread holds, otherwise, the call will result in an **IllegalMonitorStateException**.

final void **wait(long timeout)** throws InterruptedException
final void **wait(long timeout, int nanos)** throws InterruptedException
final void **wait()** throws InterruptedException

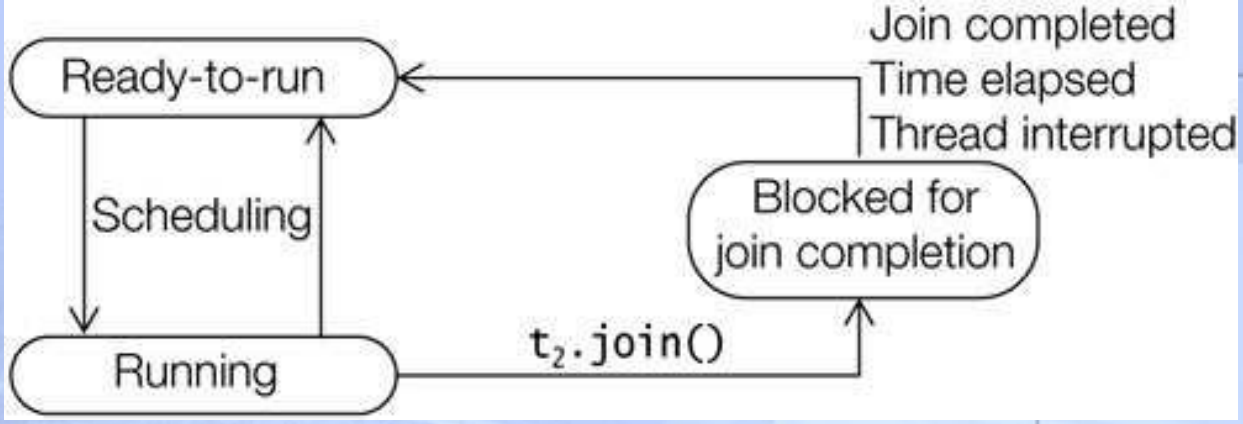
After called a method, the thread is added to the wait set of the object.

Waiting and Notifying (cont.)

final void **notify()**
final void **notifyAll()**

- notify thread(s) that are in the wait set of the object to go to the ready state.

Joining



void **join()** throws InterruptedException

- To wait for other thread to complete its execution before continuing.
- A running thread t1 invokes the method join() on a thread t2.
- The join() call has no effect if t2 has already completed.
- If Thread t2 is still alive, then thread t1 waits until
  - t2 completes
  - t1 times out
  - t1 interrupted.

Example

```
public static void main(String[] args) {
Counter counterA = new Counter("Counter A");
    Counter counterB = new Counter("Counter B");
     counterA .start();          counterB.start();
    try {
          System.out.println("Wait for the child threads to finish.");
          counterA.join();
          if (!counterA.isAlive())
                System.out.println("Counter A not alive.");
          counterB.join();
          if (!counterB.isAlive())
                System.out.println("Counter B not alive.");
          } catch (InterruptedException e) {
                System.out.println("Main Thread interrupted."); }
    System.out.println("Exit from Main Thread.");   }
```

Possible Output

Wait for the child threads to finish.
Counter A: 0
Counter B: 0
Counter A: 1
Counter B: 1
Counter A: 2
Counter B: 2
Counter A: 3
Counter B: 3
Counter A: 4
Counter B: 4
Exit from Counter A.
Counter A not alive.
Exit from Counter B.
Counter B not alive.
Exit from Main Thread.

Thread Termination

- The Thread dies when it completes its run() method, either by returning normally or by throwing an exception.
- Once in this state, the thread can not be restarted, even by calling the start() method once more on the thread object.

Assignment

Create a Bank Account which includes accountHolder, accountNumber and balanace. Add constructors and the methods deposit and withdraw.

Create another class which creates two threads, one thread will deposit to the shared bank account and other will withdraw from that account.

Write this program in synchronized manner.

Corporate Profile

Thank you!