Network programming (IT423+IT432)

Spring 2017

Dr. Islam Taj-Eddin

IT Dept., FCI, Assiut Univ.

Threads

Threads

- A thread is a single stream of execution within a process.
- A process is a program executing in its own address space.
- You have been using threads all along.
- The main control or execution of all the programs up until now were controlled by a single thread.
- What we want to look at is mutlithreading or having multiple threads executing within the same program.

Threads

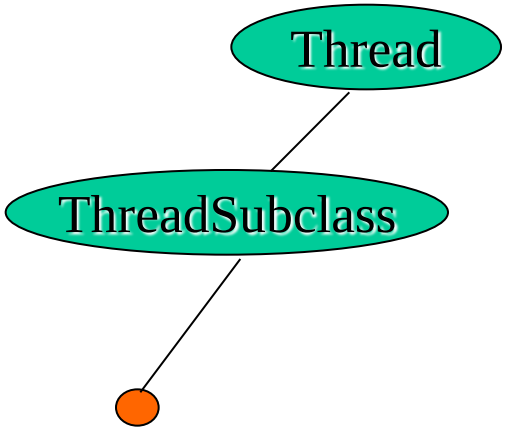
- You might be more familiar with the term multitasking.

- Multitasking:
  - having more than one program working at what seems to be at the same time.
  - The OS assigns the CPU to the different programs in a manner to give the impression of concurrency.
  - There are two types of multitasking:
    preemptive
    cooperative

- Multithreading:
  - extends the idea of multitasking by allowing individual programs to have what appears to be multiple tasks.
  - Each task within the program is called a thread.

Threads

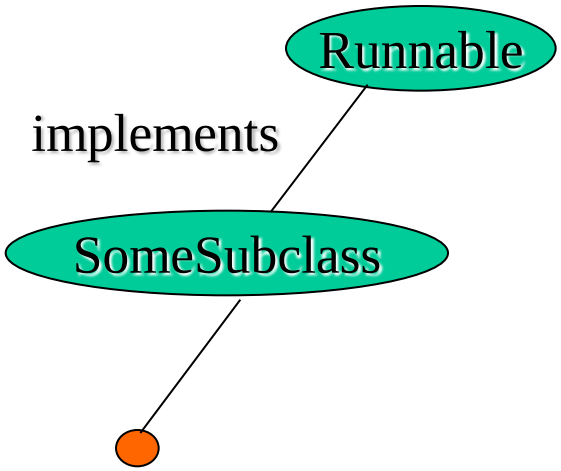
- Java has multithreading built into it.
- Java provides a Thread class for handling threads.
- There are two ways to Thread objects
  - creating objects from subclasses of the Java Thread class
  - implementing the Runnable interface for an object




```
class ThreadX extends  Thread {
        public void run() {
        //logic for the thread
        }
}

ThreadX tx = new ThreadX();
tx.start();
```

Threads




```
class RunnableY implements  Runnable {
        public void run() {
        //logic for the thread
        }
}

RunnableY ry = new RunnableY();
Thread ty = new Thread(ry);
ty.start();
```

Thread Class

- The Thread class is part of the java.lang package.

- Using an object of this class, the corresponding thread can be stopped, paused, and resumed.

- There are many constructors and methods for this class, we will look at a few of them:

  - Thread( ) - no argument
  - Thread(String n) - creates a new Thread with the name n.
  - Thread(Runnable target) - creates a new Thread object.
  - Thread(Threadgroup group, Runnable target) - This creates a new Thread object in the specified Threadgroup.

Methods in Thread Class

static methods:

- activeCount();
- currentThread();
- sleep();
- yield();

instance methods:

- getPriority();
- setPriority();
- start();
- stop();
- run();
- isAtive();
- suspend();
- resume();
- join();

Static Methods of Thread Class

- static int activeCount() - returns the number of currently active threads.
  - num_ threads = Thread. activeCount();
- static Thread currentThread() - returns the object corresponding to the currently executing thread (self reference)
  - Thread myself = Thread. currentThread();
- static void sleep(long millis) - throws InterruptedException, this causes the current thread to sleep for the specified amount of time. Other threads will run at this time.

Static Methods of Thread Class

- You can also specify the number of nanoseconds as well
  - static void sleep(long millis, int nano);

- static void yield() - causes the thread to yield the processor to any other waiting threads - Java does not guarantee preemption, you should use this to ensure fairness.

Instance Methods of Thread Class

- These methods control the thread represented by a Thread object:

- int getPriority() - returns the threads priority - a value between Thread. MIN_ PRIORITY and Thread. MAX_ PRIORITY

- void setPriority(int newpri)
  - this sets the threads priority
  - high priority threads will preempt lower ones when they become ready to run.
  - Thread myself = Thread. currentThread();
    myself. setPriority(Thread. MAX_ PRIORITY);
  - A ThreadGroup, may restrict the maximum priority of all its member threads, therefore the setPriority method may not succeed.

Instance Methods of Thread Class

- void start() - throws IllegalThreadStateException, actually starts the thread, the thread starts and enters the run() method
- void stop() - throws SecurityException, stops the thread
- void run() - this method is called when the thread is started, this is what the thread will execute while it is alive.
- boolean isAlive() - returns a value indicating whether the thread is currently alive, i.e. Started more recently and has not yet been died.

Instance Methods of Thread Class

- void suspend() - suspends the threads execution
- void resume() - resumes the execution of the thread
- void join() – causes the caller to wait until the thread dies

Creating Threads

- Creating a thread by subclassing the Thread class
- This method will allow only five thread to be started in an object.

```
public class SimpleThread extends Thread {
  private int countDown = 3;
  private static int threadCount = 0;
  private int threadNumber = ++threadCount;
  public SimpleThread( ) {
     System.out.println(“Making ” + threadNumber++);
  }
```

Creating Threads

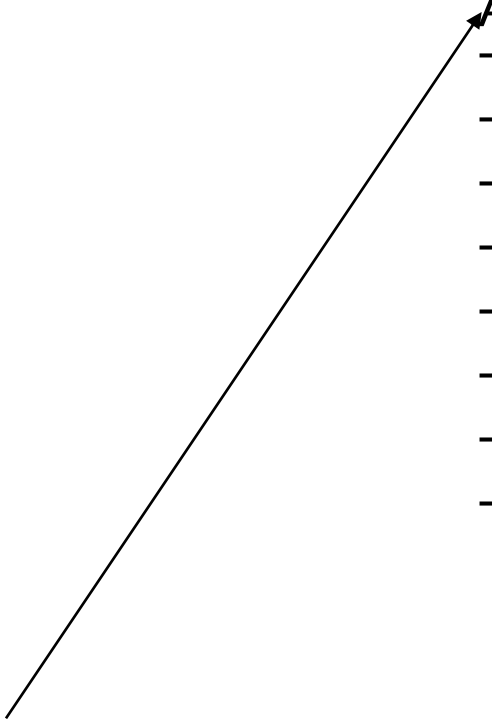
```
    public void run( ) {
        while(true) {
            System.out.println(“Thread ” + threadNumber +
            “(“ + countDown + “)”);
            if (--countDown == 0) return;
        }
    }

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

Creating Threads

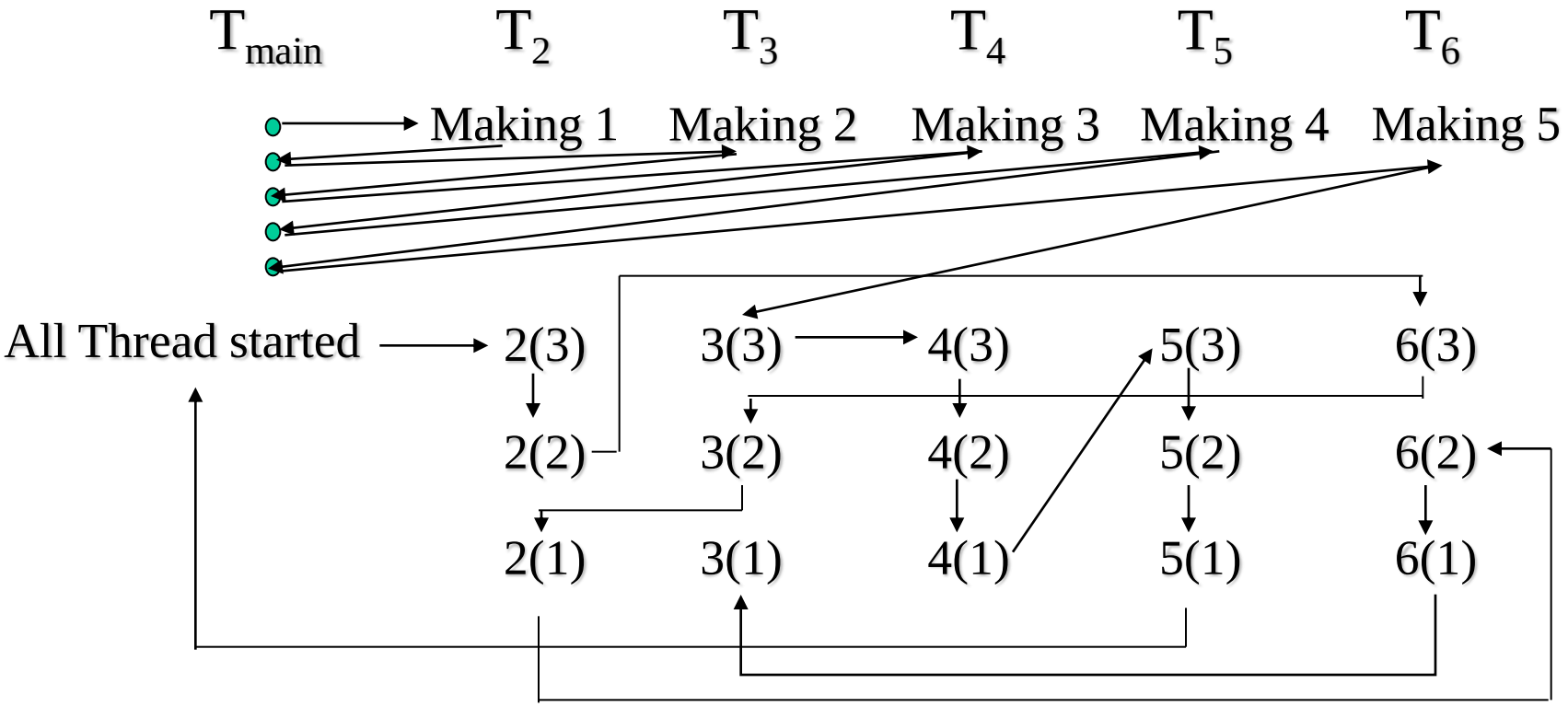
- One possible output of ‘SimpleThread’:

```
Making 1
Making 2
Making 3
Making 4
Making 5
Thread 3(3)
Thread 4(3)
Thread 4(2)
Thread 4(1)
Thread 5(3)
Thread 5(2)
Thread 5(1)
All Threads Started
Thread 2(3)
Thread 2(2)
Thread 6(3)
Thread 3(2)
Thread 2(1)
Thread 6(2)
Thread 6(1)
Thread 3(1)
```



Creating Threads

- One possible output of ‘SimpleThread’:



Synchronization in Threads

- Synchronization is a mechanism to control the execution of different threads so that:
  - when multiple threads access a shared variable, proper execution can be assured.

- Java has the synchronized keyword - this can be used to identify a segment of code or method that should be accessible to just a single thread at a time.

- Before entering a synchronization region, a thread should obtain the semaphore associated with that region - it is already taken, then the thread blocks (waits) until the semaphore is released.

Synchronization in Threads

```
class Account {
    private int balance = 0;
    synchronized void deposit(int amount)}
        balance += amount;
}}

class Customer extends Thread {
    Account account;
    Customer(Account account) {
        this.account = account;
}

    public  void run() {
        try { for (int i = 0; i < 10000; i++)
                {account.deposit(10);}
        }
```

Synchronization in Threads

```
        catch (Exception e) {
            e.printStackTrace();
        }
}}

public class BankDemo {
    private final static int NUMCUSTOMER = 10;
    public static void main(String args[]) {
        //Create account
        Account account = new Account();
        //Create and start customer threads
        Customer customer[] = new Customer[NUMCUSTOMER];
        for (int i = 0; i < NUMCUSTOMER; i++) {
          customer[i] = new Customer(account);
          customer[i].start();
        }
```

Synchronization in Threads

```
        //Wait for customer threads to complete
        for (int i = 0; i < NUMCUSTOMER; i++) {
                try {
                        customer[i].join();
                }
                catch (InterruptedException e) {
                        e.printStackTrace();
                }
        }
        //Display account balance
        System.out.println(account.getBalance());
    }
}
```

Synchronization in Threads

- In Java, any object with one or more synchronized methods is a monitor.
- When threads call a synchronized method, only one thread is let in at a time, the others wait in a queue.
- In producer- consumer type applications, consumer threads might find that there is not enough elements to consume
- It is the job of the monitor to ensure that the threads that are waiting for the producer are notified once the elements are produced.

Thread Communication

- A thread can temporarily release a lock so other threads can have an opportunity to execute a synchronized method.

- It is because the Object class defined three methods that allow threads to communicate with each other.
  - void wait() - causes the thread to wait until notified - this method can only be called within a synchronized method.
    - void wait(long msec) throws InterruptedException
    - void wait(long msec, int nsec) throws InterruptedException
  - void notify() - notifies a randomly selected thread waiting for a lock on this object - can only be called within a synchronized method.
  - void notifyall() - notifies all threads waiting for a lock on this object - can only be called within a synchronized method.

Thread Communication

```
class Producer extends Thread {
    Queue queue;
    Producer (Queue queue) {
        this.queue = queue;
    }

    public void run {
        int i = 0;
        while(true) {
          queue.add(i++);
        }
    }
}
```

Thread Communication

```
class Comsumer extends Thread {
    String str;
    Queue queue;
    Consumer (String str, Queue queue) {
        this.str = str;
        this.queue = queue;
    }

    public void run {
        while(true) {
          System.out.println(str + “: ” + queue.remove(););
        }
    }
}
```

Thread Communication

```
class queue {
    private final static int SIZE = 10;
    int array[] = new int[SIZE];
    int r = 0;
    int w = 0;
    int count = 0;
    synchronized void add(int i) {
        //wait while the queue is full
        while (count == SIZE) {
          try {
               wait()
          }
          catch (InterruptedException ie) {
               ie.printStackTrace();
               System.exit(0);
        }}
```

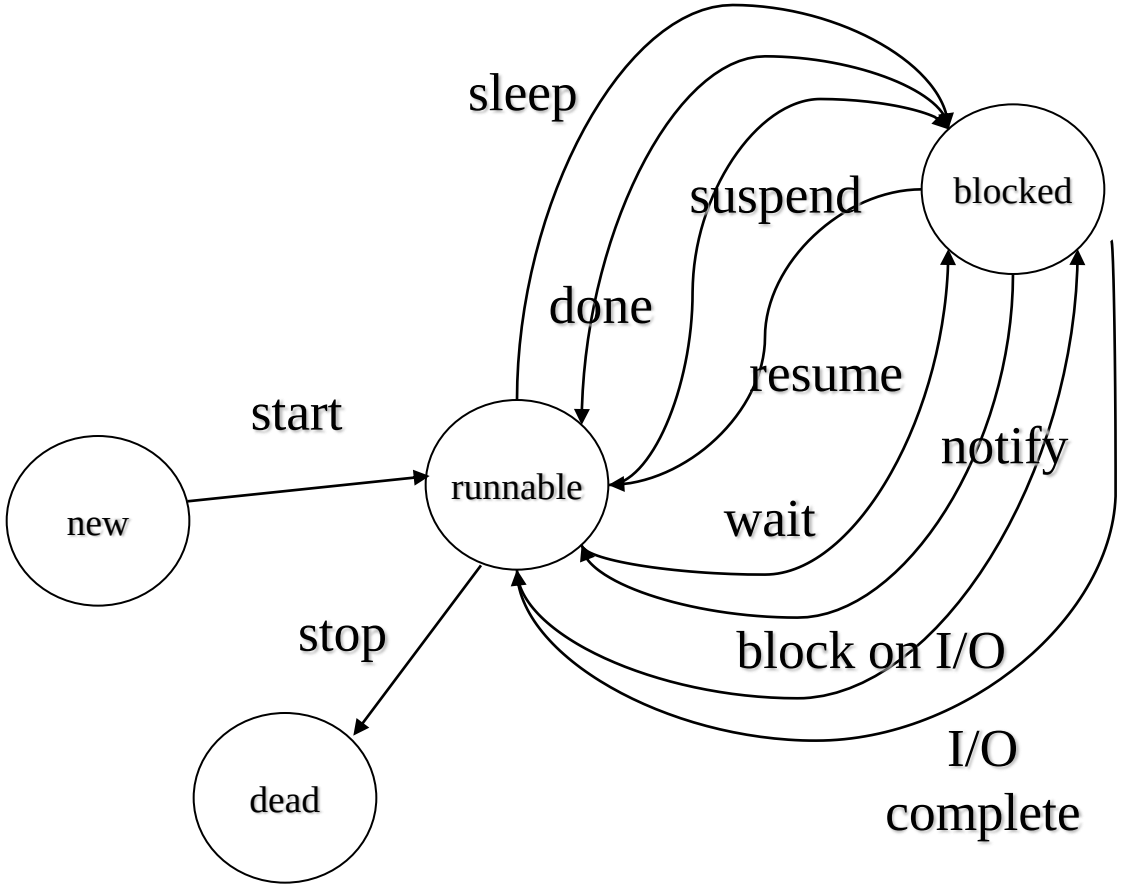
Thread Communication

```
//Add data to array and adjust write pointer
array[w++] =  i;
if (w >= SIZE)
      w = 0;
 //Increment count
++count;
//Notify waiting threads
notifyAll();
}
synchronized int remove() {
      //wait while the queue is empty
      while (count == 0) {
        try { wait();}
        catch (InterruptedException ie) {
            ie.printStackTrace();
            System.exit(0);}}
```

Thread Communication

```
//read data from array and adjust read pointer
int element = array[r++];
if (r >= SIZE)
      r = 0;
 //Decrement count
--count;
//Notify waiting threads
notifyAll(); return element;
}}
public ProducerConsumer {
      public static void main(String args[]) {
      Queue queue = new Queue();
      new Producer(queue).start();
      new Consumer(“ConsumerA”, queue).start();
      new Consumer(“ConsumerB”, queue).start();
      new Consumer(“ConsumerC”, queue).start();}}
```

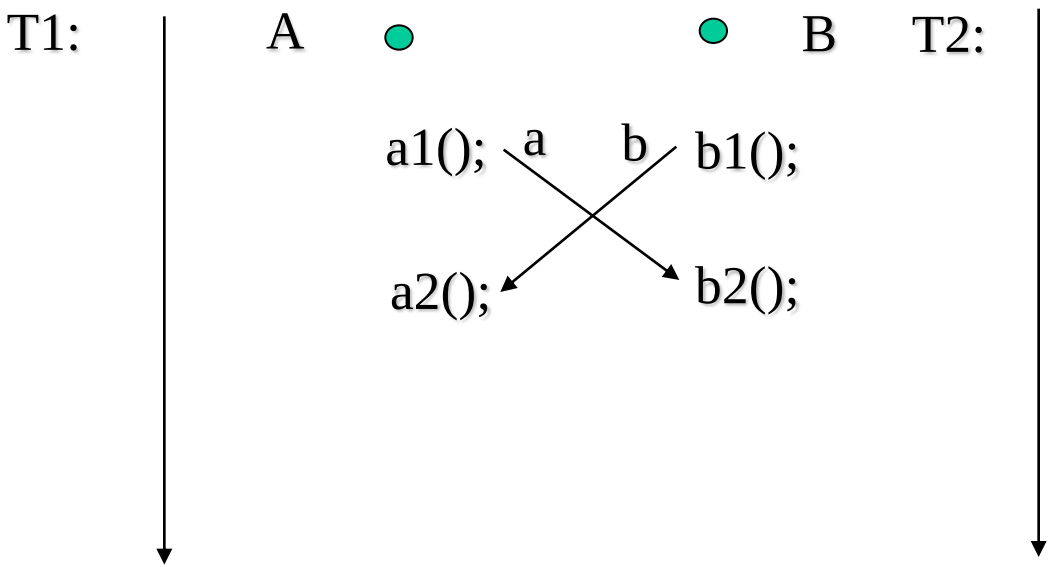
Thread Properties



Deadlock

- Deadlock is an error that can be encountered in multithreads.
- It occurs when two or more threads wait indefinitely for each other to relinquish locks.
- Assume that thread-1 holds a lock on object-1 and waits for a lock on object-2.
- Thread-2 holds a lock on object-2 and waits for a lock on object-2.
- Neither of these threads may proceed.
- Each waits forever for the other to relinquish the lock it needs.

Deadlock



Deadlock

```
class A {
    B b;
    synchronized void a1() {
        System.out.println(“Starting a1”);
        b.b2();
    }

    synchronized void a2() {
        System.out.println(“Starting a2”);
    }
}
```

Deadlock

```
class B {
    A a;
    synchronized void b1() {
        System.out.println(“Starting b1”);
        a.a2();
    }

    synchronized void b2() {
        System.out.println(“Starting b2”);
    }
}
```

Deadlock

```
class Thread1 extends Thread {
    A a;
    Thread1(A a) {
        this.a = a;
    }

    public void run() {
        for (int i = 0; i < 100000; i++)
          a.a1();
    }
}
```

Deadlock

```
class Thread2 extends Thread {
    B b;
    Thread2(B b) {
        this.b = b;
    }

    public void run() {
        for (int i = 0; i < 100000; i++)
          b.b1();
    }
}
```

Deadlock

```
public class DeadlockDemo {
    public static void main(String args[]) {
        //Create objects
        A a = new A();
        B b = new B();
        a.b = b;
        b.a = a;
        //Create threads
        Thread1 t1 = new Thread1(a);
        Thread2 t2 = new Thread2(b);
        t1.start(); t2.start();}
        //wait for threads to complete
        try {t1.join(); t2.join();}
        catch (Exception e) { e.printStackTrace(); }
        System.out.println(“Done!”);
}}
```

Deadlock

The following is sample output from this application:

```
Starting a1
Starting b2
Starting a1
Starting b2
Starting a1
Starting b2
Starting a1
Starting b2
Starting a1
Starting b1
```