Multithreading

Threads

- The ability to do multiple things at once within the same application
  - Finer granularity of concurrency
- Lightweight
  - Easy to create and destroy
- Shared address space
  - Can share memory variables directly
  - May require more complex synchronization logic because of shared address space

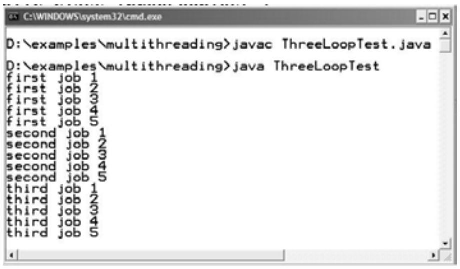
---

Three Loops
Sequential Execution

Example Code: ThreeLoopTest

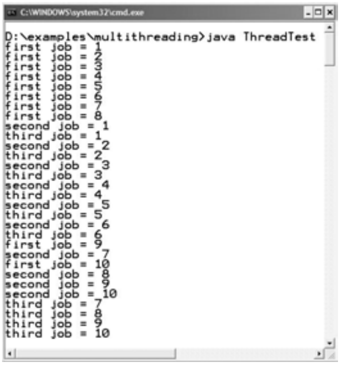
```
public class ThreeLoopTest {
  public static void main (String args[ ]){

     // first loop
     for (int i=1; i<= 5; i++)
         System.out.println(“first ” +i);

     // second loop
     for (int j=1; j<= 5; j++)
        System.out.println(“second ” + j);

    // third loop
    for (int k=1; k<= 5; k++)
        System.out.println(“third ” + k);

  }
}
```

Compile & Execute: ThreeLoopTest



```
D:\examples\multithreading>javac ThreeLoopTest.java

D:\examples\multithreading>java ThreeLoopTest
first job 1
first job 2
first job 3
first job 4
first job 5
second job 1
second job 2
second job 3
second job 4
second job 5
third job 1
third job 2
third job 3
third job 4
third job 5
```

Multi-Threaded Output



```
D:\examples\multithreading>java ThreadTest
first job = 1
first job = 2
first job = 3
first job = 4
first job = 5
first job = 6
first job = 7
first job = 8
second job = 1
third job = 1
second job = 2
third job = 2
second job = 3
third job = 3
second job = 4
third job = 4
second job = 5
third job = 5
second job = 6
third job = 6
first job = 9
second job = 7
first job = 10
second job = 8
second job = 9
second job = 10
third job = 7
third job = 8
third job = 9
third job = 10
```

Java Threads

- Java includes built-in support for threading!
  - Other languages have threads bolted-on to an existing structure

- VM transparently maps threads in Java to OS threads
  - Allows threads in Java to take advantage of hardware and operating system level advancements
  - Keeps track of threads and schedules them to get CPU time
  - Scheduling may be pre-emptive or cooperative

Creating Threads in Java

- Two approaches
  - Using Interface
    - Implement the runnable interface in a class
    - Provide an implementation for the run() method
    - Instantiate Thread object by passing runnable object in constructor
    - Start thread

  - Using Inheritance
    - Subclass java.lang.Thread
    - Override the run() method
    - Instantiate Subclass Thread Object
    - Start thread

Thread Creation Steps : using Interface

- **Step 1 - Implement the Runnable Interface**

```
class  Worker implements Runnable
```

- **Step 2 - Provide an Implementation of run method**

```
public void run ( ){
        // write thread behavior
        // code that will execute by thread
}
```

- **Step 3 - Instantiate Thread object by passing runnable object in constructor**

```
Worker w = new Worker(“first”);
Thread t = new Thread (w);
```

- **Step 4 – Start thread**
  t.start()

Three Loops
Multi-Threaded Execution

Example Code: using Interface

```
public class Worker implements Runnable {
    private String job ;
    public Worker (String j ){
       job = j;
    }
    public void run ( ) {
      for(int i=1; i<= 10; i++)
        System.out.println(job + " = "  + i);
    }
}
```

Example Code: using Interface

```
public class ThreadTest{
    public static void main (String args[ ]){

       Worker first     = new Worker (“first job”);
       Worker second = new Worker (“second job”);
       Worker third    = new Worker (“third job”)

       Thread t1 = new Thread (first );
       Thread t2 = new Thread (second);
       Thread t3 = new Thread (third);

       t1.start();
       t2.start();
       t3.start();
    }
}
```

Thread Priorities

- Every Thread has a priority
- Threads with higher priority are executed in preference to threads with lower priority
- A thread's default priority is same as of the creating thread

---

Thread Priorities

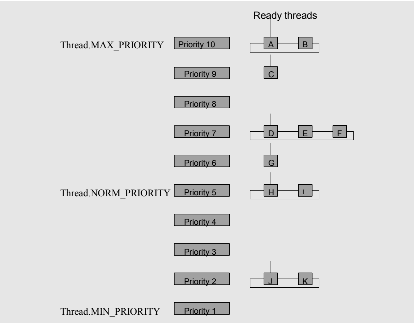
- You can change thread priority by using any of the 3 predefined constants
  - `Thread.MAX_PRIORITY` (typically 10)
  - `Thread.NORM_PRIORITY` (typically 5)
  - `Thread.MIN_PRIORITY` (typically 1)
- OR any integer value between 1 to 10 can be used as thread priority.

Useful Thread Methods

- setPriority(int priority)
  - Changes the priority of this thread
  - Throws IllegalArgumentException if the priority is not in the range MIN_PRIORITY to MAX_PRIORITY
  - For Example,

```
Thread t = new Thread(RunnableObject);
t.SetPriority(Thread.MAX_PRIORITY);
t.setPriority(7);
```

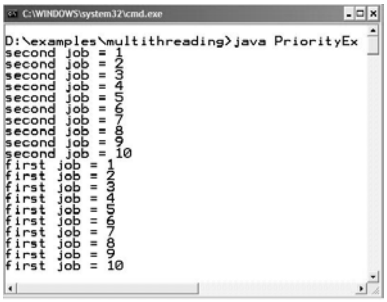
Thread priority scheduling example



Code Example: PriorityEx.java

```
public class PriorityEx{
   public static void main (String args[ ]){
      Worker first     = new Worker (“first job”);
      Worker second = new Worker (“second job”);

      Thread t1 = new Thread (first );
      Thread t2 = new Thread (second);

       t1.setPriority(Thread.MIN_PRIORITY);
       t2.setPriority(Thread. MAX_PRIORITY);

       t1.start();
       t2.start();
   }
}
```

Output: PriorityEx.java



```
D:\examples\multithreading>java PriorityEx
second job = 1
second job = 2
second job = 3
second job = 4
second job = 5
second job = 6
second job = 7
second job = 8
second job = 9
second job = 10
first job = 1
first job = 2
first job = 3
first job = 4
first job = 5
first job = 6
first job = 7
first job = 8
first job = 9
first job = 10
```

Thread Priorities

- Problems
  - A Java thread priority may map differently to the thread priorities of the underlying OS
    - Solaris has 2^{32}–1 priority levels;
    - Windows NT has only 7 user priority levels
  - Starvation can occur for lower-priority threads if the higher-priority threads never terminate, sleep, or wait for I/O

Lab Work - Reading Two Files Simultaneously

Useful Thread Methods (cont.)

- sleep (int time)
  - Causes the currently executing thread to wait for the time (milliseconds) specified
  - Waiting is efficient (non-busy)
  - Threads come out of the sleep when the specified time interval expires or when interrupted by some other thread
  - Thread coming out of sleep may go to the running or ready state depending upon the availability of the processor.

Useful Thread Methods (cont.)

- sleep (int time)
  - High priority threads should execute sleep method after some time to give low priority threads a chance to run otherwise starvation may occur
  - Sleep can be used for delay purpose
    - i.e., anyone can call Thread.sleep
    - Note that sleep throws InterruptedException. Need try/catch

---

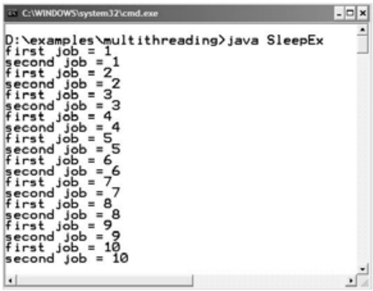
Code Example: Modify Worker.java

```
public class Worker implements Runnable {
  ………….
  public void run ( ) {
    for(int i=1; i<= 10; i++) {
      try {
         Thread.sleep(100);
      }catch (Exception ex){
         System.out.println(ex);
      }
      System.out.println(job + " = "  + i);
    } // end for
  } // end run

}// end Worker
```

Code Example: SleepEx.java

```
public class SleepEx{
   public static void main (String args[ ]){

      Worker first     = new Worker (“first job”);
      Worker second = new Worker (“second job”);

      Thread t1 = new Thread (first );
      Thread t2 = new Thread (second);

       t1.start();
       t2.start();
   }
}
```

Output: SleepEx.java



```
D:\examples\multithreading>java SleepEx
first job = 1
second job = 1
first job = 2
second job = 2
first job = 3
second job = 3
first job = 4
second job = 4
first job = 5
second job = 5
first job = 6
second job = 6
first job = 7
second job = 7
first job = 8
second job = 8
first job = 9
second job = 9
first job = 10
second job = 10
```

Useful Thread Methods (cont.)

- yield ( )
  - Allows any other threads of the same priority to execute (moves itself to the end of the priority queue)
  - If all waiting threads have a lower priority, then the yielding thread resumes execution on the CPU
  - Generally used in cooperative scheduling schemes

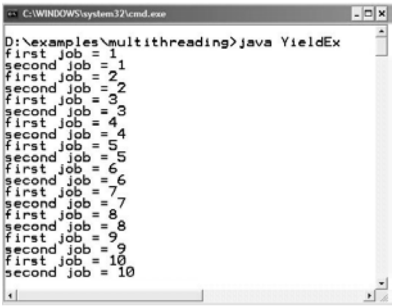
Code Example: Modify Worker.java

```
public class Worker implements Runnable {

  ………….

  public void run ( ) {

    for(int i=1; i<= 10; i++) {
      Thread.yield( );

      System.out.println(job + " = "  + i);

    } // end for
  } // end run

}// end Worker
```

Code Example: YieldEx.java

```
public class YieldEx{
   public static void main (String args[ ]){

      Worker first      = new Worker (“first job”);
      Worker second = new Worker (“second job”);

      Thread t1 = new Thread (first );
      Thread t2 = new Thread (second);

       t1.start();
       t2.start();
   }
}
```

Output: YieldEx.java

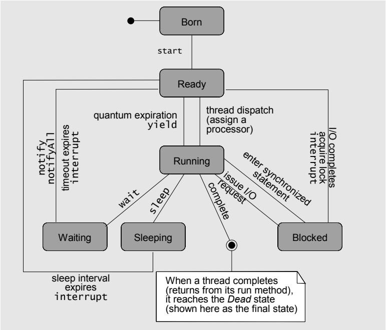


```
D:\examples\multithreading>java YieldEx
first job = 1
second job = 1
first job = 2
second job = 2
first job = 3
second job = 3
first job = 4
second job = 4
first job = 5
second job = 5
first job = 6
second job = 6
first job = 7
second job = 7
first job = 8
second job = 8
first job = 9
second job = 9
first job = 10
second job = 10
```

Thread States: Life Cycle of a Thread

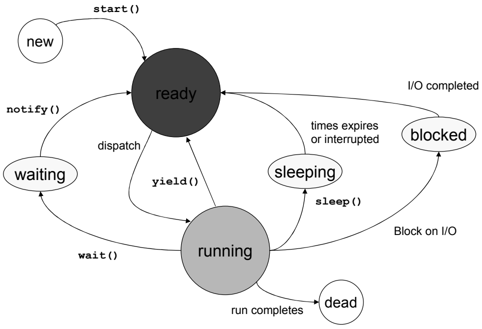
- Thread states
  - Born state
    - Thread was just created
  - Ready state
    - Thread’s `start` method invoked
    - Thread can now execute
  - Running state
    - Thread is assigned a processor and running
  - Dead state
    - Thread has completed or exited
    - Eventually disposed of by system

Thread life-cycle statechart diagram



When a thread completes (returns from its run method), it reaches the *Dead* state (shown here as the final state)

Thread Lifecycle



Joining

- Used when a thread wants to wait for another thread to complete its run()
  - Sent the thread2.join() message
    - Causes the current running thread to block efficiently until thread2 finishes its run() method
    - Must catch InterruptedException

Code Example: Modify Worker.java

```
public class Worker implements Runnable {

  ………….

  public void run ( ) {

    for(int i=1; i<= 10; i++)

      System.out.println(job + " = "  + i);

  }

}// end Worker
```

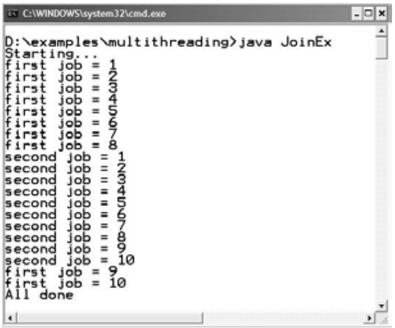
Code Example: JoinEx.java

```
public class JoinEx{
   public static void main (String args[ ]){

      Worker first     = new Worker (“first job”);
      Worker second = new Worker (“second job”);

      Thread t1 = new Thread (first );
      Thread t2 = new Thread (second);

       System.out.println("Starting...");
       t1.start();
       t2.start();

       …….
```

Code Example: JoinEx.java (cont.)

```
   // The current running thread (main() blocks until both workers have finished
    try {
       t1.join();
       t2.join();
    }
   catch (Exception ex) {
       System.out.println(ex);
   }

   System.out.println("All done ");

  }
}
```

Output: JoinEx.java



```
D:\examples\multithreading>java JoinEx
Starting...
first job = 1
first job = 2
first job = 3
first job = 4
first job = 5
first job = 6
first job = 7
first job = 8
second job = 1
second job = 2
second job = 3
second job = 4
second job = 5
second job = 6
second job = 7
second job = 8
second job = 9
second job = 10
first job = 9
first job = 10
All done
```

Synchronization

Thread's Problems

- Multiple threads can share variables among themselves.
- This sharing of variables/memory can cause synchronization problems which you must have studied in your OS course.
- The area where shared memory locations are modified are know as a critical section and only one thread should able to enter the critical section at any given point in time.

Threading

- Two Threading Challenges
  - Mutual Exclusion
    - Keeping the threads from interfering with each other
    - Worry about memory shared by multiple threads
  - Cooperation
    - Get threads to cooperate
      - Typically centers on handing information from one thread to the other, or signaling one thread that the other thread has finished doing something
    - Done using join/wait/notify

Critical Section

- A section of code that may cause problems if two or more threads are executing it at the same time
  - Typically as a result of shared memory that both thread may be reading or writing
- Race Condition
  - When two or more threads enter a critical section, they are supposed to be in a race condition because the result often depends upon the order of execution
    - Both threads want to execute the code at the same time, but if they do then bad things will happen

---

Race Condition Example

```
class Pair {
    private int a, b;

    public Pair() {
      a = 0;
      b = 0;
    }
    // Returns the sum of a and b. (reader)
    public int sum() {
      return(a+b);
    }
    // Increments both a and b. (writer)
    public void inc() {
      a++;
      b++;
    }
}
```

Reader/Writer Conflict

- Case
  - thread1 runs inc(), while thread2 runs sum()
    - thread2 could get an incorrect value if inc() is half way done
    - This happens because the lines of sum() and inc() interleave
- Note
  - Even a++ and b++ are *not* atomic statements
    - Therefore, interleaving can happen at a scale finer than a single statement!
    - a++ is really three steps: read a, increment a, write a
  - Java guarantees 4-byte reads and writes will be atomic
  - This is only a problem if the two threads are touching the same object and therefore the same piece of memory!

Reader/Writer Conflict

- Case
  - thread1 runs inc() while thread2 runs inc() on the same object
    - The two inc()'s can interleave in order to leave the object in an inconsistent state
- Again
  - a++ is not atomic and can interleave with another a++ to produce the wrong result
  - This is true in most languages

Heisenbugs

- Random Interleave – hard to observe
  - Race conditions depend on having two or more threads "interleaving" their execution in just the right way to exhibit the bug
    - Happens rarely and randomly, but it happens
  - Interleaves are random
    - Depending on system load and number of processors
    - More likely to observe issue on multi-processor systems
- Tracking down concurrency bugs can be hard
  - Reproducing a concurrency bug reliable is itself often hard
  - Need to study the patterns and use theory in order to pre-emptively address the issue

Java Locks

- Java includes built-in support for dealing with concurrency issues
  - Includes keywords in order to mark critical sections
  - Includes object locks in order to limit access to a single thread when necessary
- Java designed to encourage use of threading and concurrency
  - Provides the tools needed in order to minimize concurrency pitfalls

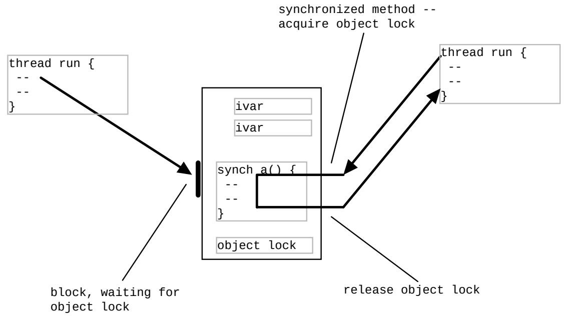
Object Lock and Synchronized keyword

- Every Java Object has as lock associated with it
- A "synchronized" keyword respects the lock of the receiver object
  - For a thread to execute a synchronized method against a receiver, it must first obtain the lock of the receiver
  - The lock is released when the method exits
  - If the lock is held by another thread, the calling thread blocks (efficiently) till the other thread exits and the lock is available
  - Multiple threads therefore take turns on who can execute against the receiver

Receiver Lock

- The lock is in the receiver object
  - Provides mutual exclusion mechanism for multiple threads sending messages to **that object**
  - Other objects have their own lock
- If a method is not synchronized
  - The thread will not acquire the lock before executing the method

Sychronized Method Picture



synchronized method --
acquire object lock

thread run {
 --
 --
}

ivar

ivar

synch a() {
 --
 --
}

object lock

thread run {
 --
 --
}

block, waiting for
object lock

release object lock

Synchronized Method Example

```
/*
 A simple class that demonstrates using the 'synchronized'
 keyword so that multiple threads may send it messages.
 The class stores two ints, a and b; sum() returns
 their sum, and inc() increments both numbers.

 <p>
 The sum() and incr() methods are "critical sections" --
 they compute the wrong thing if run by multiple threads
 at the same time. The sum() and inc() methods are declared
 "synchronized" -- they respect the lock in the receiver object.
*/
class Pair {
	private int a, b;

	public Pair() {
		a = 0;
		b = 0;
	}
```

Synchronized Method Example

```
	// Returns the sum of a and b. (reader)
	// Should always return an even number.
	public synchronized int sum() {
		return(a+b);
	}
	// Increments both a and b. (writer)
	public synchronized void inc() {
		a++;
		b++;
	}
}
```

Synchronized Method Example

```
/*
 A simple worker subclass of Thread.
 In its run(), sends 1000 inc() messages
 to its Pair object.
*/
class PairWorker extends Thread {
	public final int COUNT = 1000;
	private Pair pair;
	// Ctor takes a pointer to the pair we use
	public PairWorker(Pair pair) {
		this.pair = pair;
	}
	// Send many inc() messages to our pair
	public void run() {
		for (int i=0; i<COUNT; i++) {
			pair.inc();
		}
	}
```

Synchronized Method Example

```
	/*
	 Test main -- Create a Pair and 3 workers.
	 Start the 3 workers -- they do their run() --
	 and wait for the workers to finish.
	*/
	public static void main(String args[]) {
		Pair pair = new Pair();
		PairWorker w1 = new PairWorker(pair);
		PairWorker w2 = new PairWorker(pair);
		PairWorker w3 = new PairWorker(pair);
		w1.start();
		w2.start();
		w3.start();
		// the 3 workers are running
		// all sending messages to the same object
```

Synchronized Method Example

```
		// we block until the workers complete
		try {
			w1.join();
			w2.join();
			w3.join();
		}
		catch (InterruptedException ignored) {}

		System.out.println("Final sum:" + pair.sum()); // should be 6000
		/*
		 If sum()/inc() were not synchronized, the result would
		 be 6000 in some cases, and other times random values
		 like 5979 due to the writer/writer conflicts of multiple
		 threads trying to execute inc() on an object at the same time.
		*/
	}
}
```

Producer/Consumer Relationship

Producer/Consumer Relationship

- Consider a Producer/Consumer relationship in which a producer thread deposits a sequence of numbers into a slot of shared memory

- The consumer thread reads this data from the shared memory and prints that data.

- Problems
    - If the threads are not synchronized, data can be lost if the producer places new data into the shared slot before the consumer consumes the previous data

    - Data can be doubled if the consumer consumes the data again before the producer produces the next item.

Example Code

Producer/Consumer Relationship

Useful Object Methods

- wait( )
    - Causes the current running thread to enters a waiting state for the particular object on which wait( ) was called

- notify( ) / notifyAll( )
    - One thread in the waiting state for a particular object becomes ready on a call to notify( ) issued by another thread associated with that object.
    - If a thread calls notifyAll( ), then all threads waiting for the object are placed in the ready state.

- Every call to wait( ) must have a corresponding call to notify( ) or call notifyAll( ) as a safeguard.

Multithreaded Server

Lab Exercise

An idiom explained even more!

- Remember:
    - public static void main(String[] args)

- Well…
    - When you run a Java program, the VM creates a new thread and then sends the main(String[] args) message to the class to be run!
    - Therefore, there is ALWAYS at least one running thread in existence!
        - We can create more threads which can run concurrently