# LECTURE 16

## JAVA CONCURRENCY

# SUBJECTS

**Introduction to Java Thread**

**Java Threads Basics**

- Sleep
- Join
- Interrupt

**Java Semaphores**

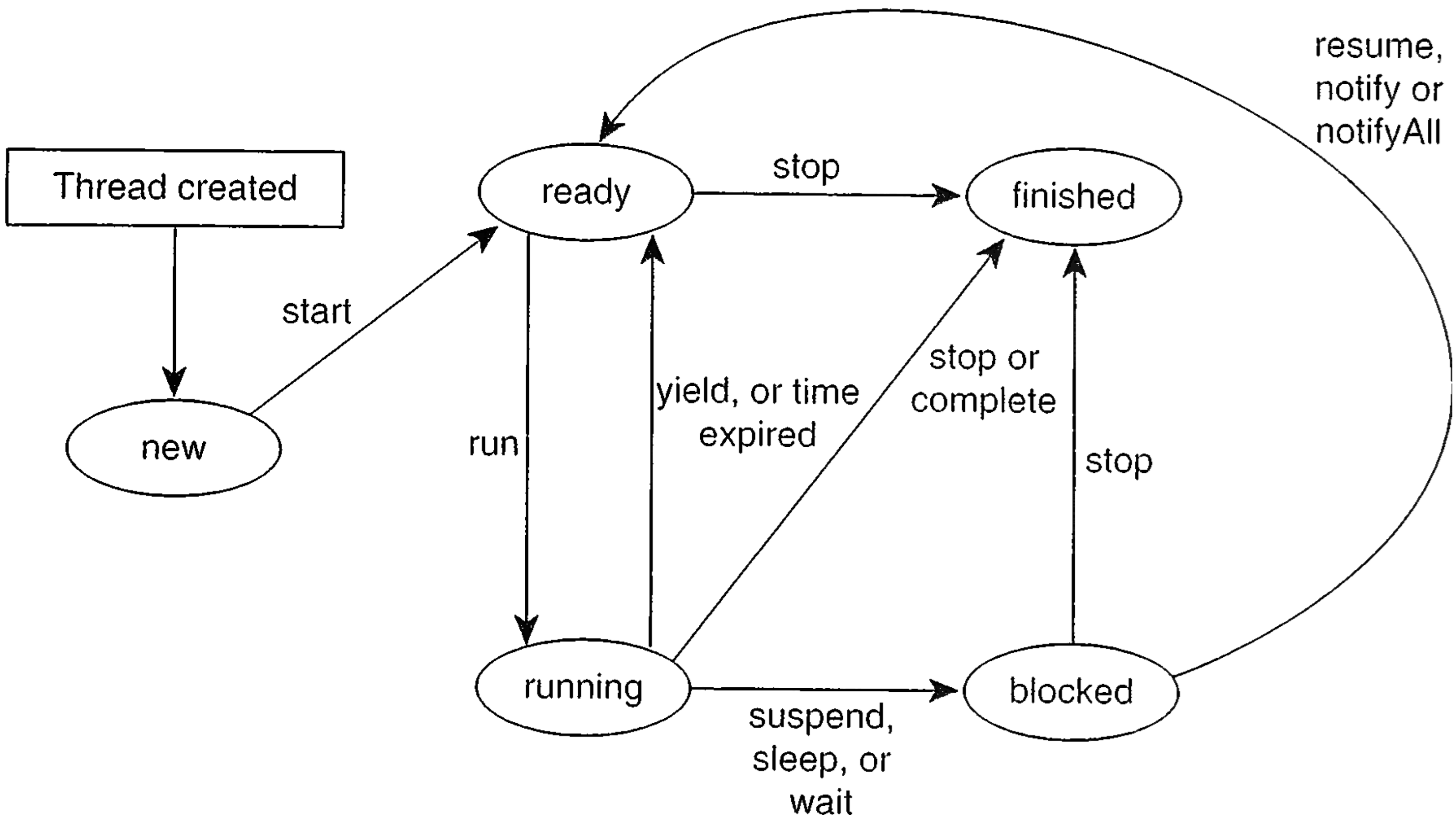
**Java Intrinsic Locks**

**Monitors Implemented with Intrinsic Locks**

**Monitors Implemented with Lock and Condition Interfaces**

**Atomic Variables**

# THREAD STATES



# CREATING THREADS BY EXTENDING THE THREAD CLASS

uOttawa
L'Université canadienne
Canada's university

```
// Custom thread class
public class CustomThread extends Thread
{
  ...
  public CustomThread(...)
  {
    ...
  }

  // Override the run method in Thread
  public void run()
  {
    // Tell system how to run custom thread
    ...
  }
  ...
}
```

```
// Client class
public class Client
{
  ...
  public someMethod()
  {
    ...
    // Create a thread
    CustomThread thread = new CustomThread(...);

    // Start a thread
    thread.start();
    ...
  }

  ...
}
```

# CREATING THREADS BY IMPLEMENTING THE RUNNABLE INTERFACE

uOttawa
L'Université canadienne
Canada's university

```
// Custom thread class
public class CustomThread
  implements Runnable
{
  ...
  public CustomThread(...)
  {
    ...
  }

  // Implement the run method in Runnable
  public void run()
  {
    // Tell system how to run custom thread
    ...
  }

  ...
}
```

```
// Client class
public class Client
{
  ...
  public someMethod()
  {
    ...
    // Create an instance of CustomThread
    CustomThread customThread
     = new CustomThread(...);

    // Create a thread
    Thread  thread = new  Thread(customThread);

    // Start a thread
    thread.start();
    ...
  }

  ...
}
```

# THREAD GROUPS

**A thread group is a set of threads**

**Some programs contain quite a few threads with similar functionality**

- We can group them together and perform operations on the entire group

**For example, we can suspend or resume all of the threads in a group at the same time.**

# USING THREAD GROUPS

**Construct a thread group:**

```
ThreadGroup g = new ThreadGroup(“thread
  group”);
```

**Place a thread in a thread group:**

```
Thread t = new Thread(g, new RunnableClass());
```

**Find out how many threads in a group are currently running:**

```
System.out.println(“the number of runnable
  threads in the group “ + g.activeCount());
```

**Find which group a thread belongs to:**

```
theGroup = t.getThreadGroup();
```

# PRIORITIES

**The priorities of threads need not all be the same**

**The default thread priority is:**

- NORM_PRIORITY(5)

**The priority is an integer number between 1 and 10, where:**

- MAX_PRIORITY(10)
- MIN_PRIORITY(1)

**You can use:**

- `setPriority(int)`: change the priority of this thread
- `getPriority()`: return this thread's priority

# THREAD SLEEPING

**You can make a thread sleep (blocked) for a number of milliseconds**

- Very popular with gaming applications (2D or 3D animation)

  `Thread.sleep(1000);`

  `Thread.sleep(1000, 1000);` (accuracy depends on system)
- Notice that these methods are static
- They thrown an `InterruptedException`

# JOINING A THREAD

**If you create several threads, each one is responsible for some computations**

**You can wait for the threads to die**

- Before putting together the results from these threads

**To do that, we use the join method defined in the Thread class**

```
try {thread.join();}

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

**Exception is thrown if another thread has interrupted the current thread**

# JOIN EXAMPLE

```
public class JoinThread {
   public static void main(String[] args) {
      Thread thread2 = new Thread(new WaitRunnable());
      Thread thread3 = new Thread(new WaitRunnable());

      thread2.start();
      try {thread2.join();} catch (InterruptedException e) {e.printStackTrace();}

      thread3.start();
      try {thread3.join(1000);} catch (InterruptedException e) {e.printStackTrace();}
   }
}
```

# INTERRUPTING THREADS

**In Java, you have no way to force a Thread to stop**

- If the Thread is not correctly implemented, it can continue its execution indefinitely (*rogue thread!*)

**But you can interrupt a Thread with the `interrupt()` method**

- If the thread is sleeping or joining another Thread, an `InterruptedException` is thrown
- In this case, the interrupted status of the thread is cleared

# INTERRUPT EXAMPLE

```
public class InterruptThread {

  public static void main(String[] args) {

    Thread thread1 = new Thread(new WaitRunnable());
    thread1.start();

    try {Thread.sleep(1000);}
    catch (InterruptedException e){e.printStackTrace();}
    thread1.interrupt();
  }
```

# INTERRUPT EXAMPLE

```
private static class WaitRunnable implements Runnable {
  @Override
  public void run() {
    System.out.println("Current time millis: " + System.currentTimeMillis());
    try {Thread.sleep(5000);}
    catch (InterruptedException e) {
      System.out.println("The thread has been interrupted");
      System.out.println("The thread is interrupted: "+Thread.currentThread().isInterrupted());
    }
    System.out.println("Current time millis: " + System.currentTimeMillis());
  }
}
```

**Sample Output:**
Current time millis : 1274017633151
**The thread has been interrupted**
**The thread is interrupted : false**
Current time millis : 1274017634151

# INTRINSIC LOCKS

**Any piece of code that can be simultaneously modified by several threads must be made Thread Safe**

**Consider the following simple piece of code:**

```
public int getNextCount(){
  return ++counter;
}
```

**An increment like this, is not an atomic action, it involves:**

- Reading the current value of `counter`
- Adding one to its current value
- Storing the result back to memory

# INTRINSIC LOCKS

If you have two threads invoking `getNextCount()`, the following sequence of events might occur (among many possible scenarios):

| | | | |
|---|---|---|---|
| 1 | *Thread 1 :* *reads counter, gets **0**, adds **1**, so **counter = 1*** | 2 | *Thread 2 :* *reads counter, gets **0**, adds **1**, so **counter = 1*** |
| 3 | *Thread 1 :* *writes **1** to the counter field and **returns 1*** | 4 | *Thread 2 :* *writes **1** to the counter field and **returns 1*** |

**Therefore, we must use a lock on access to the "counter"**

**You can add such lock to a method by simply using the keyword: `synchronized`**

```
public synchronized int getNextCount(){
    return ++counter;
}
```

**This guarantees that only one thread executes the method**

**If you have several methods with the synchronized keyword, for a given Object, only one method can be executed at a time**

- This is called an **Intrinsic Lock**

# INTRINSIC LOCKS

**Each Java object has an intrinsic lock associated with it (sometimes simply referred to as monitor)**

**In the last example, we used that lock to synch access to a method**

- But instead, we can elect to synch access to a block (or segment) of code

```
public int getNextValue() {
        synchronized (this) {return value++;}
 }
```

- Or alternatively, use the lock of another object

```
public int getNextValue() {
          synchronized (lock) {return value++;}
}
```

- The latter is useful since it allows you to use several locks for thread safety in a single class

# INTRINSIC LOCKS

**So we mentioned that each Java object has an intrinsic lock associated with it**

**What about static methods that are not associated with a particular object?**

- There is also an intrinsic lock associated with the class
- Only used for synchronized class (static) methods

# JAVA MONITORS

**Bad news: In Java, there is no keyword to directly create a monitor**

**Good news: there are several mechanisms to create monitors**

- The simplest one, uses the knowledge that we have already gathered regarding the intrinsic locks

# JAVA MONITORS

**The intrinsic locks can be effectively used for mutual exclusion (competition synchronization)**

**We need a mechanism to implement cooperation synchronization**

- In particular, we need to allow threads to suspend themselves if a condition prevents their execution in a monitor
- This is handled by the `wait()` and `notify()` methods

These two methods are so important, they have been defined in the Object class…

# "WAIT" OPERATIONS

**`wait()`**

Tells the calling thread to give up the monitor and **wait** until some other thread enters the same monitor and calls **`notify()`** **`or`** **`notifyAll()`**

**`wait(long timeout)`**

Causes the current thread to wait until another thread invokes the **`notify()`** **`or`** **`notifyAll()`** method, or the specified amount of time elapses

# "NOTIFY" OPERATIONS

**`notify()`**

Wakes up a single thread that is waiting on this object's monitor (intrinsic lock). If more than a single thread is waiting, the choice is arbitrary (*is this fair?*)

The awakened thread will not be able to proceed until the current thread relinquishes the lock.

**`notifyAll()`**

Wakes up all threads that are waiting on this object's monitor.

The awakened thread will not be able to proceed until the current thread relinquishes the lock.

The next thread to lock this monitor is also randomly chosen

# INTRINSIC LOCK BASED MONITOR EXAMPLE

```
public class BufferMonitor{
    int [] buffer = new int [5];
    int next_in = 0, next_out = 0, filled = 0;

    public synchronized void deposit (int item ) throws InterruptedException{
        while (buffer.length == filled){
            wait(); // blocks thread
        }

        buffer[next_in] = item;
        next_in = (next_in + 1) % buffer.length;
        filled++;

        notify(); // free a task that has been waiting on a condition
    }
```

# INTRINSIC LOCK BASED MONITOR EXAMPLE

```
    public synchronized int fetch() throws InterruptedException{
        while (filled == 0){
            wait(); // block thread
        }

        int item = buffer[next_out];
        next_out = (next_out + 1) % buffer.length;
        filled--;

        notify(); // free a task that has been waiting on a condition

        return item;
    }
}
```

# ANOTHER MECHANISM TO CREATE MONITORS

**You can also create a monitor using the Java `Lock` interface**

- `ReentrantLock` is the most popular implementation of `Lock`

**`ReentrantLock` defines two constructors:**

- Default constructor
- Constructor that takes a Boolean (specifying if the lock is fair)

**In a fair lock scheme, the threads will get access to the lock in the same order they requested it (FIFO)**

- Otherwise, the lock does not guarantee any particular order
- Fairness is a slightly heavy (in terms of processing), and therefore should be used only when needed

**To acquire the lock, you just have to use the method `lock`, and to release it, call `unlock`**

# “LOCK” EXAMPLE

```
public class SimpleMonitor {
   private final Lock lock = new ReentrantLock();
   public void testA() {
      lock.lock();
      try {//Some code}
      finally {lock.unlock();}
   }
   public int testB() {
      lock.lock();
      try {return 1;}
      finally {lock.unlock();}
   }
}
```

# QUESTION??

***Why do we need the try-finally construct in the previous example?***

# ANOTHER MECHANISM TO CREATE MONITORS

**What about conditions?**

- Without being able to wait on a condition, monitors are useless…
  - *Cooperation is not possible*

**There is a specific class that has been developed just to this end: `Condition` class**

- You create a **`Condition`** instance using the **`newCondition()`** method defined in the **`Lock`** interface

# BUFFER MONITOR EXAMPLE (AGAIN)

```
public class BufferMonitor {

    int [] buffer = new int [5];
    int next_in = 0, next_out = 0, filled = 0;

    private final Lock lock = new ReentrantLock(true);
    private final Condition notFull = lock.newCondition();
    private final Condition notEmpty = lock.newCondition();
```

# BUFFER MONITOR EXAMPLE (AGAIN)

```
public void deposit (int item ) throws InterruptedException{
    lock.lock(); // Lock to ensure mutually exclusive access
    try{
        while (buffer.length == filled){
            notFull.await(); // blocks thread (wait on condition)
        }
        buffer[next_in] = item;
        next_in = (next_in + 1) % buffer.length;
        filled++;
        notEmpty.signal(); // signal thread waiting on the empty condition
    }
    finally{
        lock.unlock();// Whenever you lock, you must unlock
    }
}
```

# BUFFER MONITOR EXAMPLE (AGAIN)

```
public void fetch () throws InterruptedException{
    lock.lock(); // Lock to ensure mutually exclusive access
    try{
        while (filled == 0){
            notEmpty.await(); // blocks thread (wait on condition)
        }
        int item = buffer[next_out];
        next_out = (next_out + 1) % buffer.length;
        filled--;
        notFull.signal(); // signal thread waiting on the full condition
    }
    finally{
        lock.unlock();// Whenever you lock, you must unlock
    }
    return item;
}
```

# "LOCK" VS "SYNCHRONIZED"

**Monitors implemented with `Lock` and `Condition` classes have some advantages over the *intrinsic lock* based implementation:**

1. Ability to have more than one condition variable per monitor (see previous example)
2. Ability to make the lock fair (remember, synchronized blocks or methods do not guarantee fairness)
3. Ability to check if the lock is currently being held (by calling the **`isLocked()`** method)
   - Alternatively, you can call **`tryLock()`** which acquires the lock only if it is not held by another thread
4. Ability to get the list of threads waiting on the lock (by calling the method **`getQueuedThreads()`**)

**The above list is not exhaustive…**

# "LOCK" VS "SYNCHRONIZED"

**Disadvantages of `Lock` and `Condition` :**

1. Need to add lock acquisition and release code

2. Need to add try-finally block

# JAVA SEMAPHORES

**Java defines a semaphore class:**

`java.util.concurrent.Semaphore`

**Creating a counting semaphore:**

```
Semaphore available = new Semaphore(100);
```

**Creating a binary semaphore:**

```
Semaphore available = new Semaphore(1);
```

**We will later implement our own Sempahore class**

# SEMAPHORE EXAMPLE

```
public class Example {
    private int counter= 0;

    private final Semaphore mutex = new Semaphore(1)

    public int getNextCount() throws InterruptedException {
        mutex.acquire();
        try {
            return ++counter;
        } finally {
            mutex.release();
        }
    }
}
```

# SEMAPHORE EXERCISE

**Although a Semaphore class is included in the standard Java library, nonetheless, with the knowledge you accumulated so far,**

***can you create a Counting Semaphore class using Intrinsic Locks?***

# SEMAPHORE EXERCISE

```java
public class Semaphore{
    private int count;
    public Semaphore (int count) {
        this.count = count;
    }
    public synchronized void acquire(){
        while (count <=0){
            try {
                wait();
            }
            catch (InterruptedException e){}
        }
        count--;
    }
```

```java
    public synchronized void release(){
        ++count;
        notify();
    }
}
```

# ATOMIC VARIABLES

**In case you require synch for only a variable in your class, you can use an atomic class to make it thread safe:**

- AtomicInteger
- AtomicLong
- AtomicBoolean
- AtomicReference

**These classes make use of low level hardware mechanisms to ensure synchronization**

- This results in better performance

# ATOMIC VARIABLES EXAMPLE

```
public class AtomicCounter {
    private final AtomicInteger value = new AtomicInteger(0);
    public int getValue(){
        return value.get();
    }
    public int getNextValue(){
        return value.incrementAndGet();
    }
    public int getPreviousValue(){
        return value.decrementAndGet();
    }
}
```

**Other possible operations:**
`getAndIncrement(), getAndAdd(int x), addAndGet(int x)`…

# THANK YOU!

## QUESTIONS?