Java Programming, Comprehensive

Lecture 8

Bineet Sharma

Agenda: Working With Java Threads

- Introducing threads
- Thread synchronization

Working With Java Threads

Objectives

Applied

- Use the Thread class or the Runnable interface to create a thread.
- Use the methods of the Thread class to control when the processor executes a thread.
- Use the interrupt method and the InterruptedException class to create a thread that can be interrupted.
- Use the synchronized keyword to create synchronous threads.
- Use the wait and notifyAll methods of the Object class to coordinate the execution of two interdependent threads.

Knowledge

- Explain the basic difference between a program that runs in a single thread and a program that runs under multiple threads.

Working With Java Threads

Objectives (cont.)

- Name three common reasons for using threads in a Java application.
- List the three Java API classes or interfaces that have methods related to threading.
- Explain the advantage of creating a thread by extending the Runnable interface rather than by inheriting the Thread class.
- List the five states of a thread, and describe the status of a thread in each state.
- Explain how the sleep method works.
- Explain why methods that can be executed concurrently by multiple threads need to be synchronized.
- Describe the producer/consumer pattern used for concurrency control.

Working With Java Threads

What is a thread?

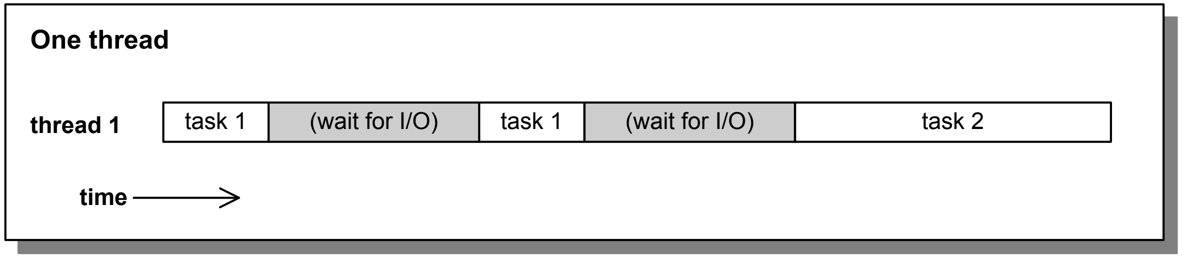
- A *thread* is a single sequential flow of control within a program
- A thread often completes one specific task
- By default, java applications use a single thread, called *main thread* and continues until main exits
- Programs can benefit by using two or more threads to allow different parts of the program to execute simultaneously
- The threads don't actually execute simultaneously in a single CPU computer. Instead, a part of the Java virtual machine called the *thread scheduler* alternately lets the portions of each thread executes, giving illusion of all of the tasks running at the same time

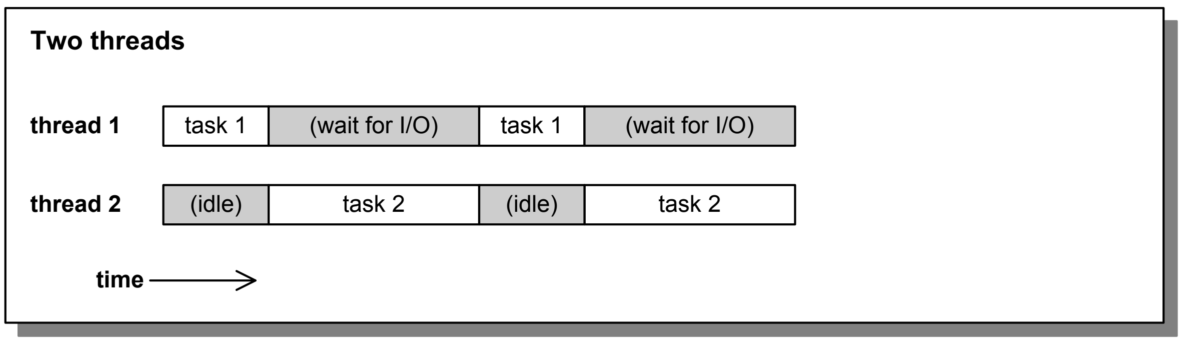
Working With Java Threads

Typical uses for threads

- To improve the performance of applications with extensive I/O operations
- To improve the responsiveness of GUI applications
- To allow two or more users to run server-based applications simultaneously

Working With Java Threads



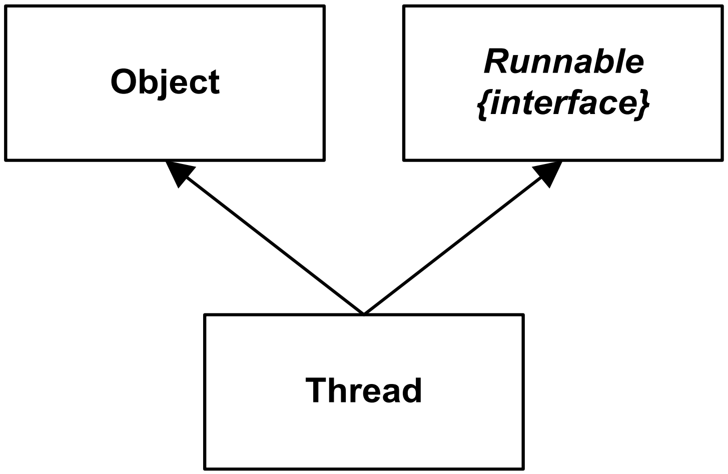


Typical I/O extensive application

- I/O is 1000sX slower than CPU operations.
- Most of the time, I/O application will wait on I/O to complete
- A second independent operations could run in parallel

Working With Java Threads

Classes and interfaces used to create threads



Working With Java Threads

Important class taking part in threads programming

Thread – Class inherits Object and implements Runnable

Runnable – Interface to be implemented by those classes whose objects are going to be executed by thread. Must define a run() method

Object – Class, which has methods that are used for threading

Working With Java Threads

Key methods of the Thread class, Runnable interface, and Object class

Method	*Class/Interface*	Description

start ***Thread*** registers this thread with the thread scheduler so it's available for execution

run **Runnable, Thread** An abstract method that's declared by the Runnable interface and implemented by the Thread class. The thread scheduler calls this method to run the thread

sleep ***Thread*** Causes the current thread to wait (sleep) for a specified period of time so the CPU can run other threads

Working With Java Threads

Key methods of the Thread class, Runnable interface, and Object class

Method	Class/Interface	Description
wait	Object	Causes the current thread to wait until another thread calls the notify or notifyAll method for the current object
notify	Object	Notifies one arbitrary thread that's waiting on this object that it can resume execution
notifyAll	Object	Notifies all the threads that are waiting on this object that they can resume execution

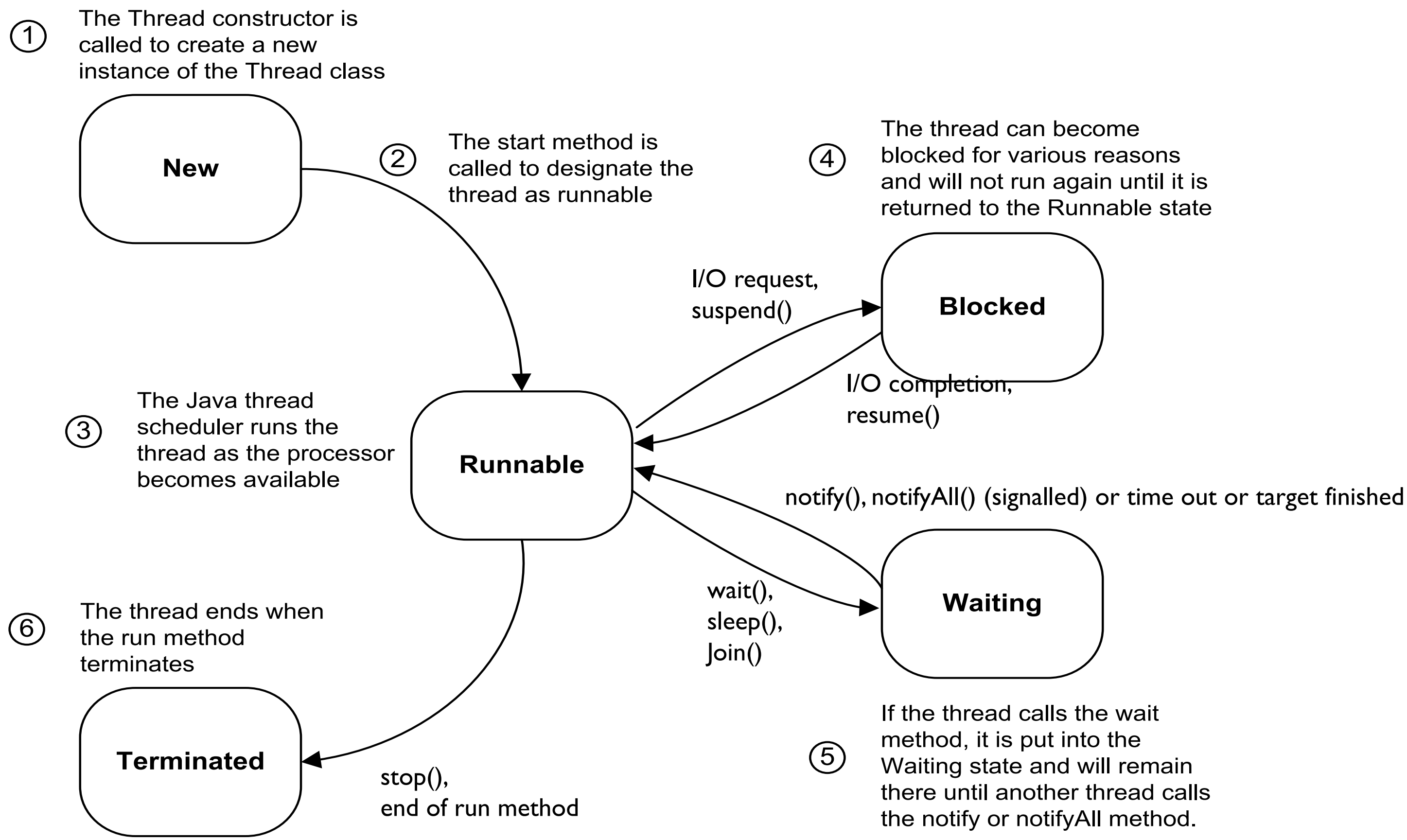
Working With Java Threads

Two ways to create a thread

- Inherit the Thread class.
- Implement the Runnable interface in your class, and then pass a reference to this Runnable object to the constructor of the Thread class.
  - This is useful if the thread needs to inherit a class other than the Thread class.

Working With Java Threads

The life cycle of a thread



Working With Java Threads

Thread states

- **New**: Created, but not yet started
- **Runnable**: Thread is available to run by the thread scheduler – it may be running or waiting in the thread queue
- **Blocked**: Temporarily has been removed from Runnable state – either by *IO/locked*. It will return to Runnable state when condition changes – e.g. I/O completes
- **Waiting**: *wait* method has been called, waits until another method calls *notify or notifyAll*.
- **Terminated**: *run* method has ended

Working With Java Threads

The Thread class

```
java.lang.Thread;
```

Common constructors of the Thread class

- **Thread()** : creates default thread object, with a default name
- **Thread(**Runnable**)** : creates thread from any object which has implemented Runnable interface
- **Thread(**String**)** : creates default thread object with specified name
- **Thread(**Runnable, String**)**: creates specified thread with a name

Working With Java Threads

Common methods of the Thread class

- **`run()`**: implements run method of Runnable interface. This is the code run.
- **`start()`**: places the thread in a runnable state
- **`getName()`**: returns the name of the thread
- **`currentThread()`** returns the currently running thread object
- **`setDaemon(`**`boolean`**`)`** makes the thread a daemon (child/subordinate) thread and terminates with parent (by default threads are independent)
- **`sleep(`**`long`**`)`** places the current thread in a blocked state for specified millisecond, throws InterrruptedException
- **`interrupt()`** interrupts a thread
- **`isInterrupted()`** returns true if the thread is interrupted
- **`yield()`** hint to thread scheduler that this thread can yield (which may or may not be respected)

Working With Java Threads

You can create threads in one of the two ways:

- By extending the Thread class
- By implementing the Runnable interface

Creating a thread from Thread class

1. Create a class that inherits the Thread class.
2. Override the run method (with no arguments) to perform the desired task.
3. Create the thread by instantiating an object from the class.
4. Call the start method of the thread object.

Working With Java Threads

Sample output for a thread by extending the Thread class

- *main* is the main thread, which starts your application
  - *Do not confuse with the main() method, we are talking main thread*
- Here, *main* start another thread *Thread-0*.
- Once *main* starts *Thread-0* it terminates (completes it tasks)
- Then *Thread-0* starts and then terminates (complete it tasks)

```
main started.
main starts Thread-0.
main finished.
Thread-0 started.
Thread-0 finished.
```

Create thread

Using Thread class

IOThread.java

Main.java

Working With Java Threads

A class named IOThread that defines a thread

```
public class IOThread extends Thread
{
    @Override
    public void run()
    {
        //if you are here, who is running?
        System.out.println(this.getName() + " started.");

        try
        {
            // Sleep for 2 seconds to simulate an IO task
            // that takes a long time
            Thread.sleep(2000);
        }
        catch(InterruptedException e) {}

        System.out.println(this.getName() + " finished.");
    }
}
```

Working With Java Threads

A Main class that starts a thread

```
public class Main
{
    public static void main(String[] args)
    {
        //if you are here, who is running?
        Thread t1 = Thread.currentThread();
        System.out.println(t1.getName() + " started.");

        // create the IO thread
        Thread t2 = new IOThread();

        // start the IO thread
        t2.start();

        System.out.println(t1.getName() + " starts " +
            t2.getName() + ".");

        System.out.println(t1.getName() + " finished.");
        //t2 is out of scope, however, it will keep running
    }
}
```

Working With Java Threads

Creating a thread using the Runnable interface

1. Create a class that implements the Runnable interface.
2. Implement the run method to perform the desired task.
3. Create the thread by supplying an instance of the Runnable class to the Thread constructor.
4. Call the start method of the thread object.

Create thread

Using

Runnable Interface

IOTask.java

Main_2.java

Working With Java Threads

An IOTask class that implements the Runnable interface

```
public class IOTask implements Runnable{
    @Override
    public void run()
    {
        Thread ct = Thread.currentThread();
        //if you are here, who is running?
        System.out.println(ct.getName() + " started.");

        try
        {
            // Sleep for 2 seconds to simulate an IO task
            // that takes a long time
            Thread.sleep(2000);
        }
        catch(InterruptedException e) {}

        System.out.println(ct.getName() + " finished.");
    }
}
```

Working With Java Threads

A Main class that starts a thread using the Runnable interface

```
public class Main_2
{
    public static void main(String[] args)
    {
        Thread t1 = Thread.currentThread();
        System.out.println(t1.getName() + " started.");

        // create the new thread
        Thread t2 = new Thread(new IOTask());
                //in last example we did
                // Thread t2 = new IOThread();

        // start the new thread
        t2.start();
        System.out.println(t1.getName() + " starts " +
                           t2.getName() + ".");

        System.out.println(t1.getName() + " finished.");
        //t2 is out of scope, however, it will keep running
    }
```

Working With Java Threads

The setPriority method of the Thread class

- **`setPriority(int)`** : hint to thread scheduler to set the priority of the thread between 1 to 10 (may not be honored)

Fields of the Thread class used to set thread priorities

- **`MAX_PRIORITY`** maximum priority (10)
- **`MIN_PRIORITY`** minimum priority (1)
- **`NORM_PRIORITY`** default priority (5).
  - By default, every thread gets the priority of the thread which created it. Thread created by main thread gets priority of 5.

The priority of thread is system dependent as the thread scheduler relies on the OS for implementing threads

Working With Java Threads

A Main class that sets the priority of a thread

```
public class Main
{
    public static void main(String[] args)
    {
        Thread t1 = new Thread(new IOTask());
        t1.setPriority(Thread.MIN_PRIORITY);
        t1.start();
        . . .
    }
}
```

Working With Java Threads

Interrupt a thread: Sample output

- How would you approach in writing a thread application which will display the following output:

```
Press the Enter key to stop counting.
Thread-0 count 1
Thread-0 count 2
Thread-0 count 3

Thread-0 interrupted.
```

Interrupt a thread

Counter.java

Main_3.java

Working With Java Threads

A Counter class that defines a task that can be interrupted

```
public class Counter implements Runnable
{
    @Override
    public void run()
    {
        Thread ct = Thread.currentThread();
        int count = 1;
        while (!ct.isInterrupted())
        {
            System.out.println(
                ct.getName() + " count " + count);
            count++;
            try
            {
                // Sleep for 1 second
                Thread.sleep(1000);
            }
```

Working With Java Threads

A Counter class that defines a task that can be interrupted (cont.)

```
            catch(InterruptedException e){
                break;
            }
        }
        System.out.println(ct.getName() + " interrupted.");
    }
}
```

If a thread is interrupted while it is sleeping, InterruptedException is thrown, in this case isInterrupted method won't indicate that the thread has been interrupted. Hence the above code

Working With Java Threads

A Main class that interrupts a thread

```
import java.util.Scanner;

public class Main_3 {
    public static void main(String[] args) {
        System.out.println(
            "Press the Enter key to stop counting.");
        Thread counter = new Thread(new Counter());

        // start the counter thread
        counter.start();
        Scanner sc = new Scanner(System.in);
        String s = "start";

        // wait for the user to press Enter
        while (!s.equals(""))
            s = sc.nextLine();

        // interrupt the counter thread
        counter.interrupt();
    }
}
```

Working With Java Threads

Synchronized threads

A multi-threaded program starts many threads to perform its task. These threads could be created from same thread class, or multiple thread classes

This poses, data synchronization issues if threads share the data.

Threads could run *asynchronously* (independent of each other – called *asynchronous threads*) or run *synchronously* (dependent with each other so that they can share resources – called *synchronous threads*)

Concurrency issue: when two threads access same resources, one can corrupt the data other was in the middle of using it, as there is no guarantee when any thread will run. This can happen because a running method can be interrupted in the middle and other thread running same method can start running.

Synchronized method

A method can be created as *synchronized* so that only one thread can run the method at a time – by *locking* the whole object. All other threads that attempts to run any synchronized method of that object is blocked until the first thread exits the method

Interestingly, a thread can run an unsynchronized method of a locked object, as the lock is only checked if thread tries to run synchronized method.

Even if the method has one line of code, you should synchronize it, if there is chance that two threads can execute same method

Working With Java Threads

The syntax for creating a synchronized method

```
public|private synchronized returnType methodName(
     [parameterList])
{
    statements
}
```

Working With Java Threads

A synchronized method that calculates future values

```
public synchronized double calculateFutureValue(
                              double monthlyPayment,
                              double yearlyInterestRate,
                              int years)
{
    int months = years * 12;
    double monthlyInterestRate =
        yearlyInterestRate/12/100;
    double futureValue = 0;
    for (int i = 1; i <= months; i++)
    {
        futureValue = (futureValue + monthlyPayment) *
                      (1 + monthlyInterestRate);
    }
    return futureValue;
}
```

A synchronized method that increments an instance variable

```
public synchronized int getInvoiceNumber()
{
    return invoiceNumber++; //even one liner matters
}
```

Working With Java Threads

- Can threads share a data without locking the objects?
  - Yes, you can if the shared data is a single field, as Java guarantees loading and storing of variables are atomic
    - Except for longs and doubles
    - Only loads and stores are atomic, not expressions like x++
- Issues:
  - However, threads can store variables in registers (or cpu cache)
  - So, a shared variable which is in the register of one thread is not seen by another thread hence changes by one another is lost
- Volatile variable
  - volatile int myInt;
    - *A filed may be declared volatile, in which case the Java memory model ensures that all threads see a consistent value for he variable*
    - It works for longs and doubles also

Working With Java Threads

Thread communication

Multithreaded applications need to communicate between different threads to let other thread know that some event has occurred

Following methods help communication between threads

- **`wait()`** : Current thread is placed in a Waiting state. Unless another thread calls the notify or notifyAll method of the current object. wait() relinquishes the lock on the object so that other blocked threads can run. It throws InterruptedException
- **`notify()`** : This returns one arbitrary thread to the Runnable state. May not be right thread you wanted to run next, so, nofifyAll may be better choice
- **`notifyAll()`**: Restores all threads that are waiting for the current object's lock to the Runnable state. Thread scheduler then picks one of the thread to run

These methods only work on synchronized methods, if called on unsynchronized methods you get a IllegalMonitorStateException

Working With Java Threads

Code that waits on a condition

```
// if there are no orders ready, wait
while (orderQueue.count() == 0)
{
    try
    {
        wait();
    }
    catch (InterruptedException e) {}
}
```

Code that satisfies the condition and notifies other threads

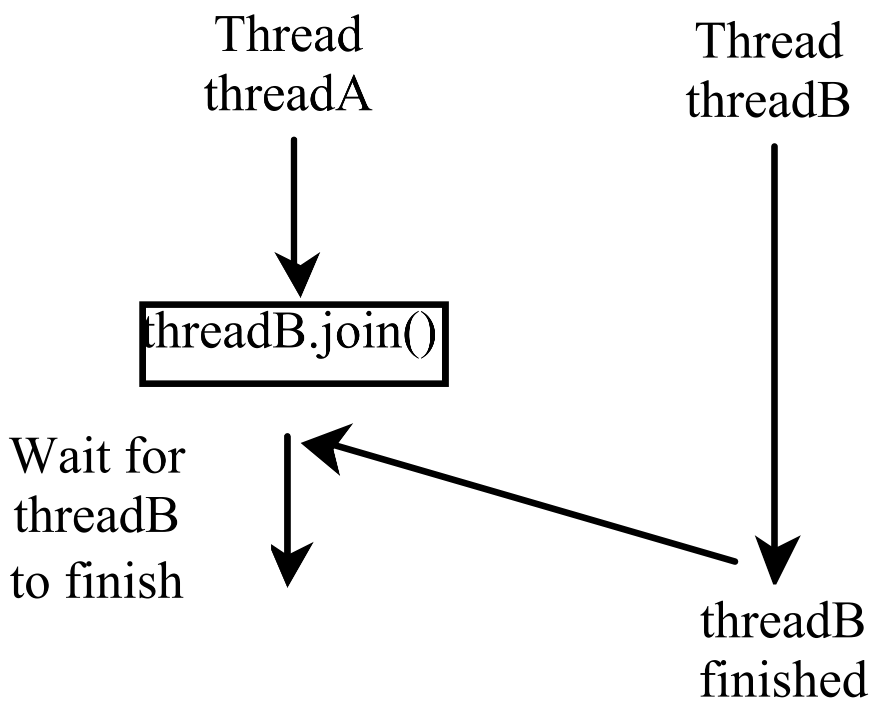
```
// add an order to the queue
orderQueue.add(order);

// notify other threads
notifyAll();
```

Working With Java Threads

Thread communication

You can use join() method to force one thread to wait for another thread to finish



Working With Java Threads

Join a thread: Sample output:

- Here, there are two threads, *JoinExample* and *PrintHello*
- Both of them print counts, however, *JoinExample* calls join() on *PrintHello* after 40
- So, *JoinExample* will wait until *PrintHello* thread terminates.

```
JoinExample Says: Join! 0
PrintHello says: Hello! 0
JoinExample Says: Join! 1
PrintHello says: Hello! 1
PrintHello says: Hello! 2
. . . .
JoinExample Says: Join! 39
PrintHello says: Hello! 42
JoinExample Says: Join! 40
PrintHello says: Hello! 43
PrintHello says: Hello! 44
PrintHello says: Hello! 45
PrintHello says: Hello! 46
PrintHello says: Hello! 47
PrintHello says: Hello! 48
```

Join a thread: Sample output

```
PrintHello says: Hello! 50
PrintHello says: Hello! 51
PrintHello says: Hello! 52
. . . .
PrintHello says: Hello! 98
PrintHello says: Hello! 99
PrintHello says: I am done!
JoinExample Says: Join! 41
JoinExample Says: Join! 42
. . . .
JoinExample Says: Join! 49
JoinExample says I am done!
```

What type of application can use this concept?

Interrupt a thread

PrintHello.java

JoinExample.java

Working With Java Threads

A thread that yields on a condition

```
class PrintHello extends Thread {
   public void run() {
      for (int i = 0; i < 100; i++) {
         System.out.println("PrintHello says:Hello!"+ i);
                  if (i%5==0) Thread.yield();
      }
      System.out.println("PrintHello says: I am done! ");
   }
}
```

Working With Java Threads

A thread that joins another thread

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

            PrintHello t1 = new PrintHello();
            t1.start();

            try {
                for (int i = 0; i < 50; i ++){
                    System.out.println("JoinExample Says:
                                        Join! " + i );
                    if (i%5==0) Thread.yield();
                    if (i == 40)
                        t1.join();
                    // wait for the thread to terminate
                }
            }
```

Working With Java Threads

A thread that joins another thread (cont.)

```
                catch (InterruptedException e) {
                    System.out.println("ERROR: Thread was
                                              interrupted");
                }

                System.out.println("JoinExample says I am
                                         done!");
            }
    }
```

Working With Java Threads

Thread states ***(further explained)*****:**

A thread state. A thread can be in one of the following states:

- NEW (after object is created)
  A thread that has not yet started is in this state.
- RUNNABLE (after start())
  A thread executing in the Java virtual machine is in this state. It can be running or waiting to be picked up depending on Scheduler. yield() or time out may put it in Ready state, to be picked up next cycle to run
- BLOCKED (waiting on a explicit lock)
  A thread that is blocked waiting for a monitor lock is in this state. A thread in the blocked state is waiting for a monitor lock to enter a synchronized block/method or reenter a synchronized block/method after callingObject.wait.

(from oracle documentation)

Working With Java Threads

Thread states *(further explained, contd.)*:

A thread state. A thread can be in one of the following states:

- WAITING ((Thread.join(), Object.wait(), LockSupport.park() with no timeout)
  A thread that is waiting indefinitely for another thread to perform a particular action is in this state. For example, a thread that has called Object.wait() on an object is waiting for another thread to call Object.notify() orObject.notifyAll() on that object. A thread that has called Thread.join() is waiting for a specified thread to terminate.
- TIMED_WAITING (waiting for timeout)

  Thread.sleep(), Object.wait() with timeout, Thread.join() with timeout, LockSuppport.parkNanos(), LockSupport.parkUntil().

  A thread that is waiting for another thread to perform an action for up to a specified waiting time is in this state.
- TERMINATED (run() completed)
  A thread that has exited is in this state.

(from oracle documentation)

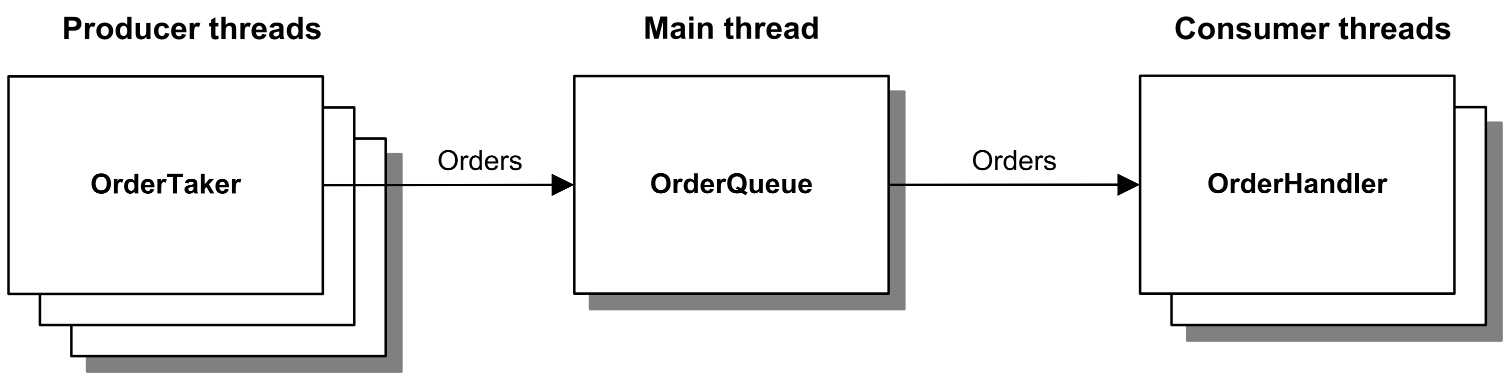
Working With Java Threads

The operation of the Order Queue application

- This application simulates a multithreaded ordering application.
- It has multiple order takers. Each order takers are running the application in a separate thread. They generate orders that are added to a queue that runs in the application in main thread.
- The orders are then handled by multiple order-handling threads, which remove orders from the queue and display them on the console
- This application is designed as *producer/consumer* design pattern. Threads that produce objects place them in a queue so the objects can later be retrieved by threads that consume them

Working With Java Threads

The operation of the Order Queue application



Working With Java Threads

Share Resources: Sample output

```
Starting the order queue.
Starting 3 order taker threads, each producing 3 orders.
Starting 2 order handler threads.

     OrderTaker threads                OrderHandler threads
============================    =============================
Order #1 created by Thread-1
Order #2 created by Thread-2
                                Order #1 processed by Thread-3
                                Order #2 processed by Thread-4
Order #3 created by Thread-0
Order #4 created by Thread-1
Order #5 created by Thread-2
Order #6 created by Thread-0
                                Order #3 processed by Thread-3
                                Order #4 processed by Thread-4
Order #7 created by Thread-1
Order #8 created by Thread-2
Order #9 created by Thread-0
                                Order #5 processed by Thread-3
                                Order #6 processed by Thread-4
                                Order #7 processed by Thread-3
                                Order #8 processed by Thread-4
                                Order #9 processed by Thread-3
```

Share Resources Among Threads

Order Queue application

Working With Java Threads

Classes used by the Order Queue application

The Order class

Constructor	Description
Order`(int number)`	Creates an Order object with the specified number.
Method	**Description**
toString()	Returns a string in the form "Order #*n*," where *n* is the order number.

The OrderQueue class

Method	Description
pushOrder`(Order order)`	Adds the specified order to the queue.
`Order` **pullOrder**`()`	Retrieves the first available order from the queue.

Working With Java Threads

Classes used by the Order Queue app (cont.)

The OrderTaker class

Constructor	Description
`OrderTaker(int orderCount, OrderQueue queue)`	Creates a new order taker that adds the specified number of orders to the queue.
Method	**Description**
`run()`	Adds the number of orders specified by the constructor to the queue specified by the constructor. A message is displayed on the console for each order added, and the thread sleeps for one second between orders.

Working With Java Threads

Classes used by the Order Queue app (cont.)

The OrderHandler class

Constructor	Description
`OrderHandler(OrderQueue queue)`	Creates a new order handler that reads orders from the specified queue.
Method	**Description**
`run()`	Retrieves orders from the queue specified by the constructor. A message is displayed on the console for each order retrieved, and the thread sleeps for two seconds between orders.

Working With Java Threads

The code for the Order class

```
public class Order
{
    private int number;

    public Order(int number)
    {
        this.number = number;
    }

    @Override
    public String toString()
    {
        return "Order #" + number;
    }
}
```

Working With Java Threads

The code for the OrderQueue class

```
import java.util.LinkedList;

public class OrderQueue
{
    private LinkedList<Order> orderQueue =
        new LinkedList<>();

    public synchronized void pushOrder(Order order)
    {
        orderQueue.addLast(order);
        // notify any waiting threads
        notifyAll();
    }
```

Working With Java Threads

The code for the OrderQueue class (cont.)

```
    public synchronized Order pullOrder()
    {
        // if no orders in queue, wait
        while (orderQueue.size() == 0)
        {
            try
            {
                wait();
            }
            catch (InterruptedException e)
            {}                          // ignore interruptions
        }
        return orderQueue.removeFirst();
    }
}
```

Working With Java Threads

The code for the OrderTaker class

```
public class OrderTaker extends Thread
{
    //shared between all OrderTaker threads
    private static int orderNumber = 1; //starts at 1

    private int count = 0;
    private int maxOrders;
    private OrderQueue orderQueue;

    public OrderTaker(int orderCount, OrderQueue orderQueue)
    {
        this.maxOrders = orderCount;      // orders to create
        this.orderQueue = orderQueue;     // order queue
    }
```

Working With Java Threads

The code for the OrderTaker class (cont.)

```
public void run(){
        Order order;
        while (count < maxOrders){
            order = new Order(getOrderNumber());
            // add order to the queue
            orderQueue.pushOrder(order);
            System.out.println(order.toString() +
                " created by " + this.getName());
            count++;   //why this is not synchronized?
            try{
                Thread.sleep(1000);  // delay one second
            }
            catch (InterruptedException e)
            {}                        // ignore interruptions
        }
    }
    private static synchronized int getOrderNumber(){
        return orderNumber++;
       //why this method is snhrhronized?
    }
}
```

Working With Java Threads

The code for the OrderHandler class

```
public class OrderHandler extends Thread {
    private OrderQueue orderQueue; //shared among handlers
    public OrderHandler(OrderQueue orderQueue){
        this.orderQueue = orderQueue;
    }
    @Override
    public void run(){
        Order order;
        while (true){ // get next available order
            order = orderQueue.pullOrder(); //if no order?
            System.out.println(
                "                              " +
                order.toString() +
                " processed by " + this.getName());
            try{
                Thread.sleep(2000);  // delay two seconds
            }
            catch (InterruptedException e){} //ignore
        }
    }
}
```

Working With Java Threads

The code for the OrderQueueApp class

```
public class OrderQueueApp
{
    public static void main(String[] args)
    {
        final int TAKER_COUNT = 3;   // OrderTaker threads
        final int ORDER_COUNT = 3;   // orders per
                                     // OrderTaker thread
        final int HANDLER_COUNT = 2; // OrderHandler threads

        // create the order queue
        OrderQueue queue = new OrderQueue();

        System.out.println("Starting the order queue.");

        System.out.println("Starting " + TAKER_COUNT +
            " order takers, " + "each producing " +
            ORDER_COUNT + " orders.");

        System.out.println("Starting " + HANDLER_COUNT +
            " order handlers.\n");
```

Working With Java Threads

The code for the OrderQueueApp class (cont.)

```
        String s = "     OrderTaker threads            "
          + "OrderHandler threads    \n"
          + "============================= "
          + "==============================";
        System.out.println(s);

        // create the Taker threads
        for (int i = 0; i < TAKER_COUNT; i++){
            OrderTaker t = new OrderTaker(
                ORDER_COUNT, queue);
            t.start();
        }

        // create the Handler threads
        for (int i = 0; i < HANDLER_COUNT; i++){
            OrderHandler h = new OrderHandler(queue);
            h.start();
        }
    }
}
//what happened to main thread, and queue?
```

Java Concurrency Framework

- Advanced Thread Concepts:
  - Issues with concurrent programming
  - Concurrency Framework

Java Concurrency Framework

- Concurrency in Java is provided by Thread
- JVM creates main thread and runs your program
- Your program can spawn multiple other threads
- Threads and processes are different, but similar in some matter
  - Process are independently running programs and have their own complete memory block
  - Threads do have program counters and call stacks, however, they share main memory, file pointers and other process state
  - That makes it easier for OS to deal with the thread, but, leaves burden on programmer to synchronize threads and all shared resources

Java Concurrency Framework

- In Java, access to shared resources are controlled through Synchronized objects and Synchronized methods
- Locking and unlocking is done automatically
- If a thread cannot acquire a lock will block
- This makes writing Thread safe (not corrupting data if multiple threads are accessing same resource) program not so trivial

Java Concurrency Framework

- Some concurrencies Issues:
  - Deadlock: If one thread locks A resource, then another thread locks a resource B, after that first thread wants to lock resource B, and then second thread wants to lock resource A
  - Race Conditions: When you cannot predict the outcome of running a multithreaded application. The output is non-deterministic, depends upon order of thread execution, speed of CPU and other resources
  - Starvation: If one thread goes un-served, may be a slow thread, or with low priority
- So, you need to write a Thread safe class that guarantees the internal state of the class being always predictable

Java Concurrency Framework

- Primitive synchronization tools are not enough as they don’t scale with program complexity
- So, instead of re-inventing the wheel Java provides Java Concurrency Framework to deal with complexity of parallel programming. It provides (higher level classes for):
  - Task Scheduling Framework (Executor): Handles invocation, scheduling and execution of tasks
  - Concurrent collections
  - Atomic variables: guarantees that variables are updated atomically
  - Locks: Similar as synchronized, however, with more granularity
- Framework provides code reusability (productivity), efficiency, reliability, and scalability

Java Concurrency Framework

- Task Scheduling Framework (Executor) handles:
  - Invocation, scheduling and execution of tasks through a set of execution policies
  - Thread pools (ThreadPoolExecutor class)
    - Makes it easier to manage creating large number of threads
    - Avoids chances of servers running out of CPU capacity (by not creating threads indefinitely due to high demand)
      - Allows fixed number of threads to be created
      - Free threads picks up a task from task queue
      - If threads are busy, tasks waits (avoids server overloading)

Java Concurrency Framework

- Concurrent Collections: Concurrency framework rewrote some of the collections optimized for parallel processing
  - Queue class is extended from Collection (implements list in LIFO or FIFO)
  - BlockingQueue interface extends queue interface, and BolockingQueue implements a thread safe operations designed for producer-consumer application
    - Defines a first-in-first-out data structure that blocks or times out when you attempt to add to a full queue, or retrieve from an empty queue.
  - ConcurrentMap extends Map
    - Defines full atomic operations. Removes or replaces a key-value pair only if the~~y~~ key is present, or add a key-value pair only if the key is absent

Java Concurrency Framework

- Synchronizer Classes: Helps you in synchronizing between threads
  - Semaphores: It is implemented as counting semaphore which holds certain number of permits
    - Threads use it as permit. If the permit is exhausted, the succeeding threads must wait
    - Threads use acquire() and release() methods to get or release permits
    - Useful to guard access to a fixed size resources. For example:
      - To limit the number of threads working in parallel to save system resources
    - They are like locks, but with the added power that they can be released by a thread other than the "owner".

Java Concurrency Framework

- Synchronizer Classes: Helps you in synchronizing between threads (contd.)
  - Mutexes: A semaphore with single permit. Similar as semaphore, however, can be released by other thread which was not holding the permit

Java Concurrency Framework

- Synchronizer Classes: Helps you in synchronizing between threads (contd.)
  - Barriers (Cyclic Barrier): Handy in making all threads reach a common destination before proceeding
    - Treads call await() once they reach the barrier
    - Timeout can be used to avoid indefinite waiting
    - The barrier can be re-used hence the name Cyclic
  - Latches: Works like a Cyclic Barrier. Good for those tasks where a set of threads needs to complete a divided tasks
    - A latch is initialized with a number. Each thread reaching the synchronization point will call countdown()
    - Once the count reaches zero all blocked threads (who called await()) are released

Java Concurrency Framework

- Synchronizer Classes: Helps you in synchronizing between threads (contd.)
  - Exchanger: Is a Cyclic Barrier with only two threads which can also exchange data at a given point

Java Concurrency Framework

- Atomic Variables: AtomicInteger, AtomicBoolean, AtomicLong
  - This provides for atomic conditional updating of single variables
  - Remember that myInt++ of a regular integer is not atomic (meaning between increment, other threads could corrupt the value if used in parallel programming)

Java Concurrency Framework

A Counter class can be made safe like this:

```
class Synchronized Counter {
      private int c = 0;

      public Synchronized void increment() {
            c++;
      }

      public Synchronized void decrement() {
            c--;
      }

      public Synchronized int value() {
            return c;
      }
}
(from Oracle tutorial)
```

Java Concurrency Framework

This is acceptable solution as Counter class is small

- But for complex class, it might be steep price. Avoid the liveness impact of unnecessary synchronization by using AtomicInteger

```
class AtomicCounter {
      private AtomicInteger c = new AtomicInteger(0);

      public void increment() {
            c.incrementAndGet();
      }

      public void decrement() {
            c.decrementAndGet();
      }

      public int value() {
            return c.get();
      }
}
(from Oracle tutorial)
```

Java Concurrency Framework

- Locks: Generalizes the lock and unlock built-in behavior
  - If the synchronization requirements are very complex, explicit locks may be better. Examples:
    - ReentrantLock implements the Lock interface
      - Reentrant means a thread can obtain the lock more than once without deadlocking
    - ReentrantReadWriteLock defines locks that may be shared among readers but are exclusive to writers
    - LockSupport provides lower level blocking and unblocking support, which allows developers to implement their own lock classes

Next Lecture

- Event driven programming

What Will You Do This Week & Next?

- Go through lecture#8 material and demo codes
  - Chapter #22 from Murach's book, and chapter #28 from Herbert Schildt's book
- Complete test your understanding quiz if any
  - Test your understanding quizzes are not graded
- Complete HW#8
  - I will give you model answer after the due date
- Read ahead your lecture#9 notes
  - Chapters #15, and #16 from Murach's book

Summary: Working With Java Threads

- Introducing threads
- Thread synchronization