

ID2212 Network Programming with Java
Lecture 5

Multithreading with Java

Hooman Peiro Sajjad
Vladimir Vlassov
KTH/ICT/SCS
HT 2016

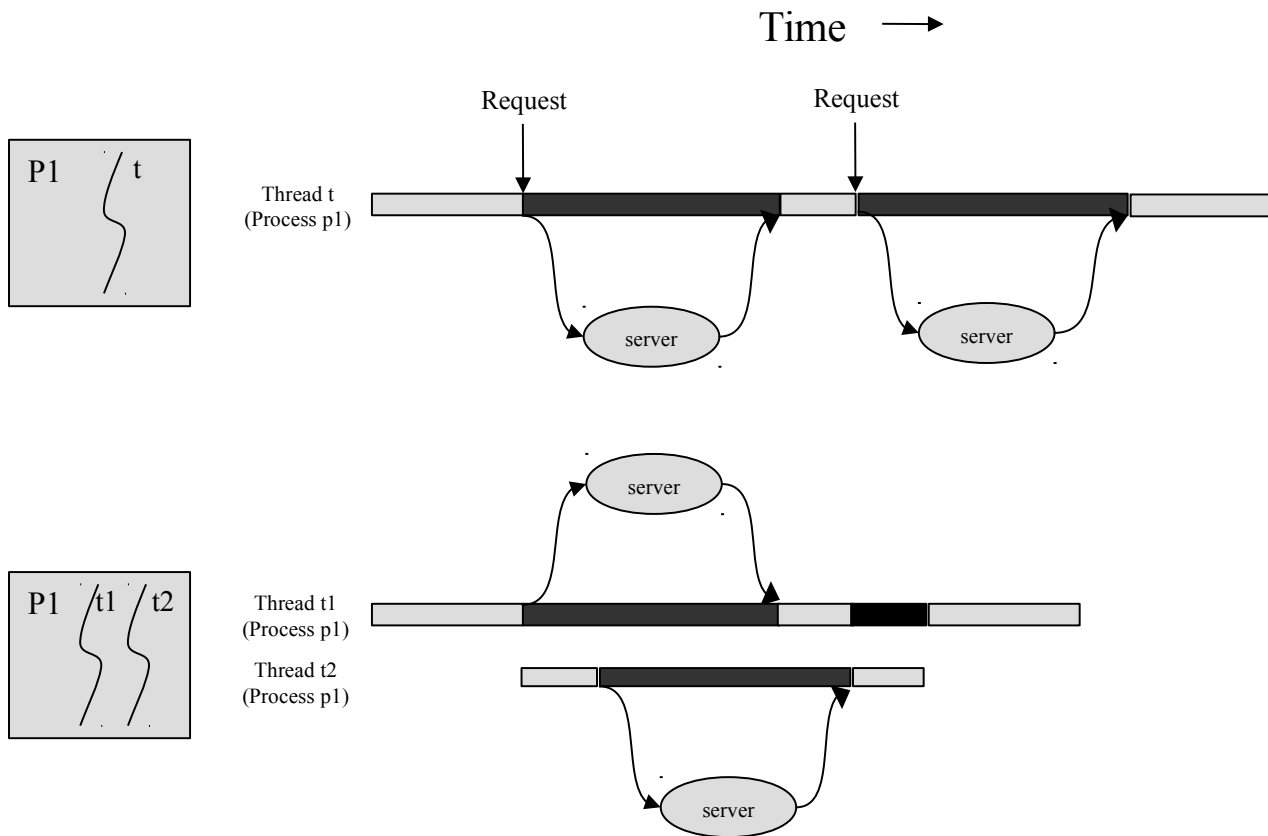
Outline

- Introduction to threads
- Multithreading with Java
 - **Thread** class and **Runnable** interface
 - Two ways of creating threads
- Java thread synchronization
 - **synchronized** methods and blocks
 - Shared objects as monitors; Bounded buffer (consumer-producer) example
- Concurrent utilities
 - Locks and Conditions;
 - The Executor framework; Example of using a thread pool
 - Synchronizers; Atomic variables; Concurrent collections
- Further reading:
<http://docs.oracle.com/javase/tutorial/essential/concurrency/>

Process and Thread (1/2)

- **Process:** A unit of activity characterized by a sequence of instructions, a current state and an associated set of system resources
- **Thread:** A path of execution in a process characterized by an execution state, execution stack and local variables

Process and Thread (2/2)



General Benefits of Using Threads

- **Creating** threads takes **less time**
- **Terminating** threads takes **less time**
- **Switching** between threads takes **less time**
- Threads in a process can **communicate** to each other **without interference of kernel**

Multithreading in a Distributed Application

- Multiple threads in a client-server application (request-response interaction)
- Client side
 - Multithreading to hide **communication latency**
 - Responsive user interface
 - Typically two threads: one to interact with the user via (G)UI; another to interact with the server e.g. over TCP socket connection
 - The (G)UI thread communicates user's control actions and input to the communication thread through e.g. **shared buffers** or by method invocations on the communication thread object, e.g. to stop it
 - Communication thread passes server responses to the (G)UI thread by GUI call-backs (method invocations) or/and through shared buffers.
 - Access to shared buffers should be **synchronized**

Multithreading in a Distributed Application

(cont'd)

- Server side
 - Multithreading for **scalability** and, as a consequence, for better performance (**higher throughput**)
 - One thread (the main thread) listens on the server port for client connection requests and assigns (creates) a thread for each client connected
 - Each client is served in its own thread on the server
 - The listening thread should provide client information (e.g. at least the connected socket) to the servicing thread
 - Typically servicing threads are independant, but might access shared resources, e.g. database, and therefore might need to be synchronized.

Multithreading in Java

- *A Java thread* is a light-weight process represented by an object of the **Thread** (sub)class that includes **start** and **run** methods
 - Stack and PC (Program Counter) register
 - Accesses all variables in its scope
- Each thread has a method **void run()**
 - Executes when the thread starts
 - Thread vanishes when it returns
 - You must implement this method
- Classes for multithreading:
 - **public class Thread**
 - **public class ThreadGroup**
 - **public interface Runnable**

First Way to Program and Create a Java Thread

1. Extend the **Thread** class

- Override the **run** method and define other methods if needed;
- Create and start a thread:
 - Instantiate the **Thread** subclass;
 - Call the **start** method on the thread object – creates a thread context and invokes **run** to be executed in a separate thread

Another Way to Program and Create Java Threads

2. Implement the **Runnable** interface in a class that represents a class of *tasks* to be execute in threads
 - Implement the **run** method;
 - Create and start a thread with the **Runnable** object, i.e. the thread is given a **Runnable** task to execute
 2. Create a **Runnable** object;
 3. Create a thread to execute that task by passing the **Runnable** object to a **Thread** constructor
 4. Call the **start** method on the thread object to start the thread.

Thread Class and Runnable Interface

```
public class Thread extends Object implements Runnable {
    public Thread();
    public Thread(Runnable target);
    public Thread(String name);
    public Thread(Runnable target, String name);
    ...
    public synchronized native void start();
    public void run();
    ...
}

public interface Runnable{
    public void run();
}
```

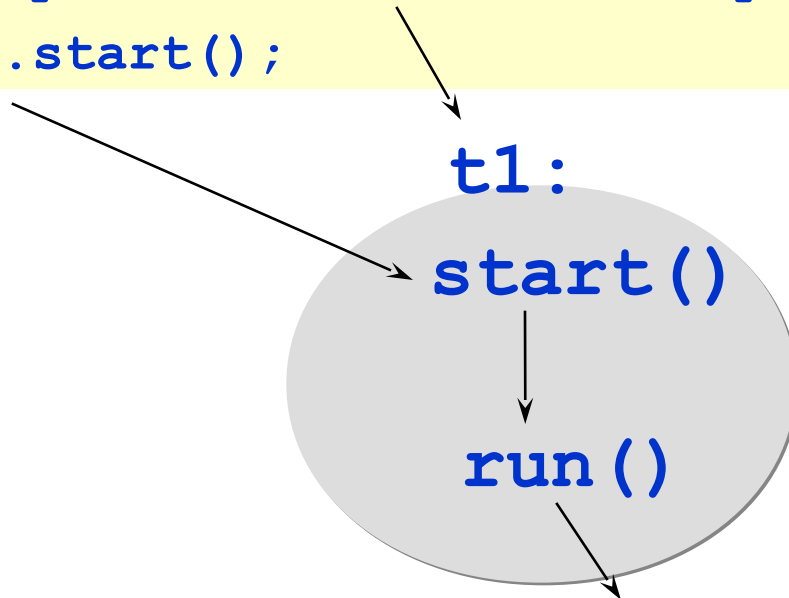
Example 1: Extending Thread

```
public class RunThreads {
    public static void main(String[] args) {
        OutputThread t1 = new OutputThread("One");
        OutputThread t2 = new OutputThread("Two");
        t1.start();
        t2.start();
    }
}

class OutputThread extends Thread {
    OutputThread(String name){ super(name); }
    public void run() {
        for (int i = 0; i < 3; i++) {
            System.out.println(getName());
            yield();
        }
    }
}
```

Starting a Thread

```
OutputThread t1 = new OutputThread("One");  
t1.start();
```



```
for (int i = 0; i < 3; i++) {  
    System.out.println(getName());  
    yield();  
}
```

Example 2. Implementing Runnable

```
public class RunThreads1 {  
    public static void main(String[] args) {  
        OutputClass out1 = new OutputClass("One");  
        OutputClass out2 = new OutputClass("Two");  
        Thread t1 = new Thread(out1);  
        Thread t2 = new Thread(out2);  
        t1.start();  
        t2.start();  
    }  
}
```

```
class OutputClass implements Runnable {
```

```
    String name;
```

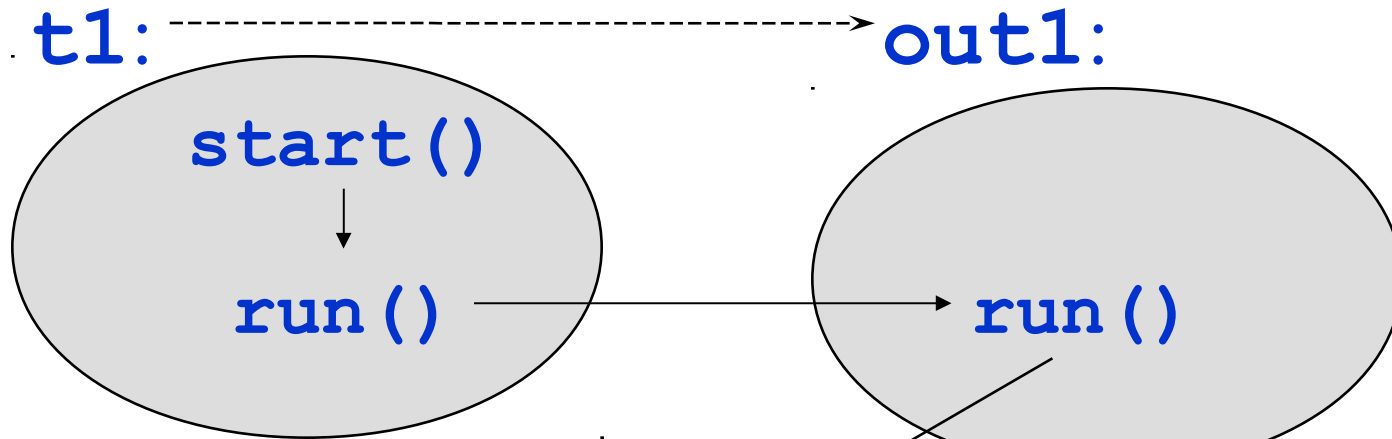
```
    OutputClass(String s) {  
        name = s;  
    }
```

```
    public void run() {  
        for ( int i=0; i<3; i++ ) {  
            System.out.println(name);  
            Thread.currentThread().yield();  
        }  
    }
```

Runnable interface

Thread with a **Runnable** Task

```
OutputClass out1 = new OutputClass("One");  
Thread t1 = new Thread(out1);  
t1.start();
```



```
for ( int i=0; i<3; i++ ) {  
    System.out.println(name);  
    yield();  
}
```

Starting a Thread

```
t.start();
```

- Starts the new thread
- Caller returns immediately
- Caller & thread run in parallel

Joining a Thread

- What if you want to wait for a thread to finish?

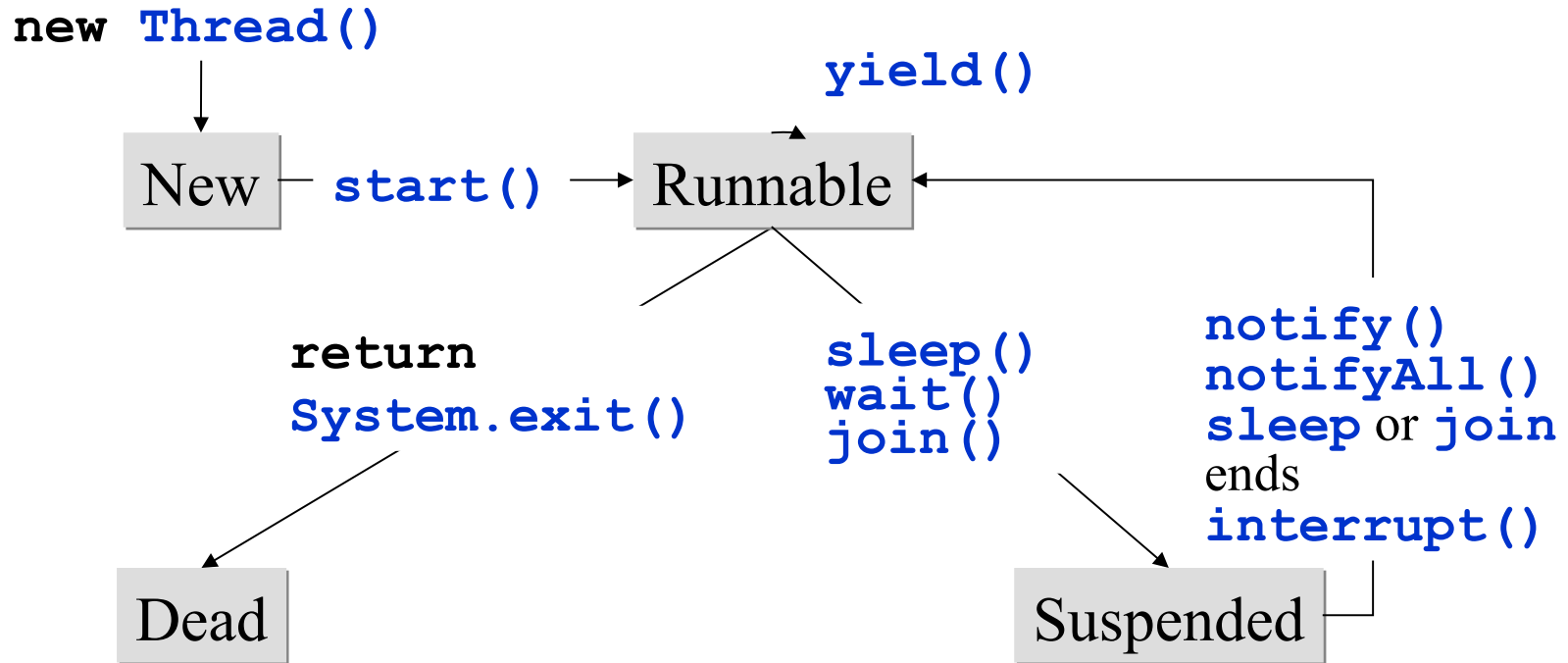
```
t.join();
```

- Blocks the caller
- Waits for the thread to finish
- Returns when the thread is done

Some Methods of the **Thread** Class

- **run()**
 - Should be overridden (the code of thread is placed here), otherwise does nothing and returns;
 - Should not be invoked directly but rather calling start().
- **start()**
 - Start the thread; JVM invokes the run method of this thread.
- **join()**
 - Wait for this thread to die.
- **yield()**
 - Causes a context switch.
- **sleep(long)**
 - The thread pauses for the specified number of milliseconds.
- **interrupt()**
 - Interrupt this thread.
- Get / set / check thread attributes:
 - **setPriority(int),**
getPriority(),
 - **setName(String),**
getName(),
 - **setDaemon(boolean),**
isDaemon()

Thread State Diagram



- IO operations affect states Runnable and Suspended in the ordinary way

Thread Interactions

- Threads in Java execute concurrently – at least conceptually.
- Threads communicate
 - By calling methods and accessing variables of Thread (Runnable) objects like ordinary objects;
 - Via pipes, TCP connections;
 - **Via shared objects.**
- An object is **shared** when multiple concurrent threads invoke its methods or access its variables.

Race Condition

```
int value = 0;  
  
public int incrementValue() {  
  
    return ++value;  
  
}
```

T1: read the value, get 0, add 1, so value = 1

T2: read the value, get 0, add 1, so value = 1

T1: write 1 to the field value and return 1

T2: write 1 to the field value and **return 1**

To avoid race condition we need to run the method **atomically**.

synchronized Methods and Blocks

- A shared object may have **synchronized methods or code blocks** to be executed with mutual exclusion
- The **synchronized** modifier defines mutual exclusion for an entire method or a code block

synchronized Methods and Blocks (cont'd)

- Each object has an **implicit lock** associated with the object
- Each object has also **one implicit condition variable** called **wait set**
- Each synchronized block requires a lock object to be explicitly indicated
- A thread must obtain the lock when calling a synchronized method or block
 - A thread may hold locks of more than one objects; nested synchronized calls are closed
- A thread may wait on and signal to the wait set

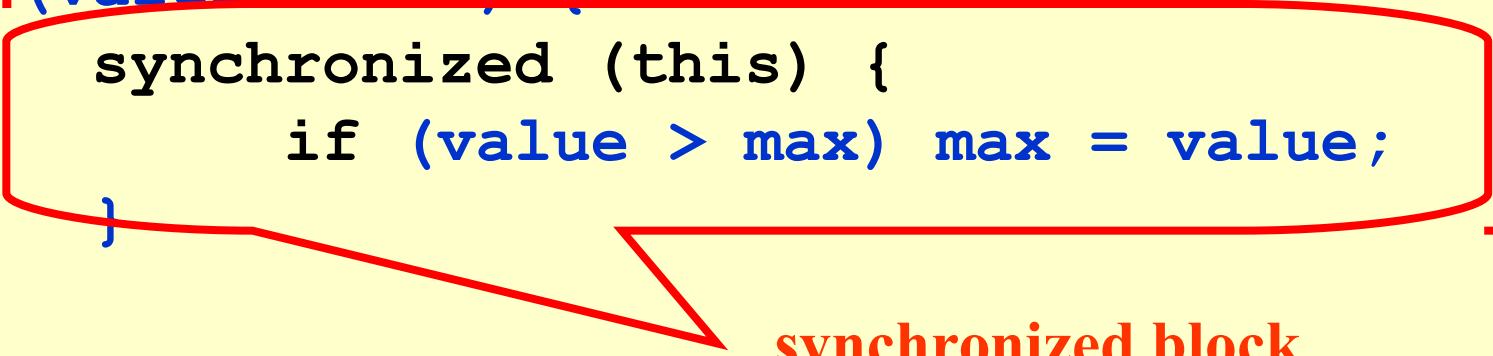
synchronized Method

```
public class ComputeMax {  
    private int max = Number.MIN_VALUE;  
    public synchronized int getMax(int value) {  
        if (value > max) max = value;  
        return max;  
    }  
}
```

synchronized method

synchronized Block

```
public class ComputeMax {  
    private int max = Number.MIN_VALUE;  
  
    public int getMax(int value) {  
        if (value > max) {  
            synchronized (this) {  
                if (value > max) max = value;  
            }  
        }  
        return max;  
    }  
}
```



synchronized block

Monitors in Java

- **Java monitor** is an object of a class with **synchronized** methods, which can be invoked by one thread at a time.
 - A class may contain synchronized and ordinary non-synchronized methods – the latter are executed without synchronization.
- Each monitor has an implicit **monitor lock**
- Each monitor has an implicit **condition variable** (a.k.a. **wait set**)
 - **wait()**, **notify()** and **notifyAll()** in scope of a **synchronized** method;
 - No priority wait;
 - Signal-and-Continue policy of **notify()** and **notifyAll()**

Java Synchronized Methods (1/5)

```
public class Queue<T> {  
  
    int head = 0, tail = 0;  
    T[QSIZE] items;  
  
    public synchronized T deq() {  
        while (tail - head == 0)  
            this.wait();  
        T result = items[head % QSIZE]; head++;  
        this.notifyAll();  
        return result;  
    }  
    ...  
}
```

Java Synchronized Methods (2/5)

```
public class Queue<T> {  
  
    int head = 0, tail = 0;  
    T[QSIZE] items;  
  
    public synchronized T deq() {  
        while (tail - head == 0)  
            this.wait();  
        T result = items[head % QSIZE]; head++;  
        this.notifyAll();  
        return result;  
    }  
    ...  
}
```

Each object has an implicit lock with an implicit condition

Java Synchronized Methods (3/5)

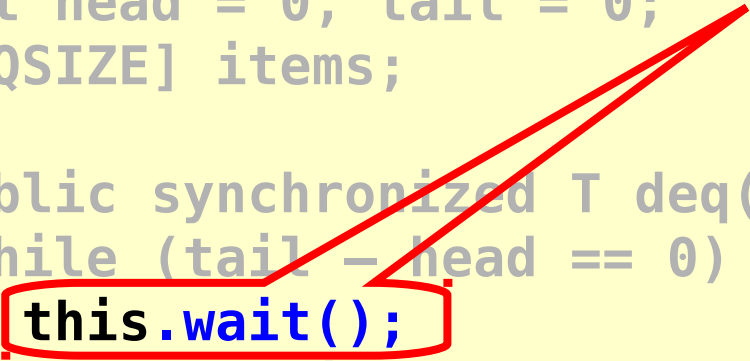
```
public class Queue<T> {  
    int head = 0, tail = 0;  
    T[QSIZE] items;  
  
    public synchronized T deq() {  
        while (tail - head == 0)  
            this.wait();  
        T result = items[head % QSIZE]; head++;  
        this.notifyAll();  
        return result;  
    }  
    ...  
}
```

Lock on entry,
unlock on return

Java Synchronized Methods (4/5)

```
public class Queue<T> {  
  
    int head = 0, tail = 0;  
    T[QSIZE] items;  
  
    public synchronized T deq() {  
        while (tail - head == 0)  
            this.wait();  
        T result = items[head % QSIZE]; head++;  
        this.notifyAll();  
        return result;  
    }  
    ...  
}
```

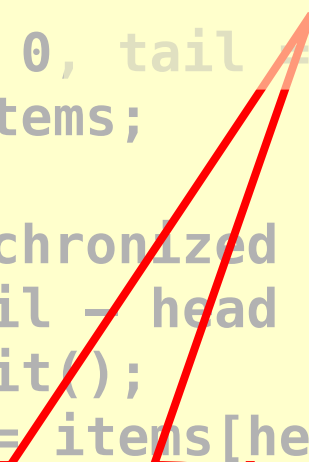
Wait on implicit condition



Java Synchronized Methods (5/5)

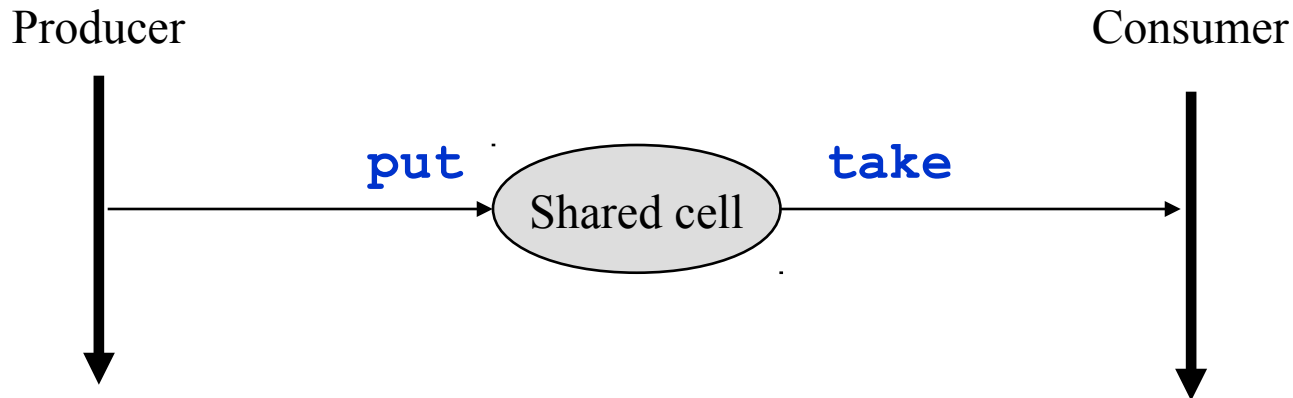
```
public class Queue<T> {  
    int head = 0, tail = 0;  
    T[QSIZE] items;  
  
    public synchronized T deq() {  
        while (tail - head == 0)  
            this.wait();  
        T result = items[head % QSIZE]; head++;  
        this.notifyAll();  
        return result;  
    }  
    ...  
}
```

Signal all threads waiting on condition



Example 1: Producer/Consumer

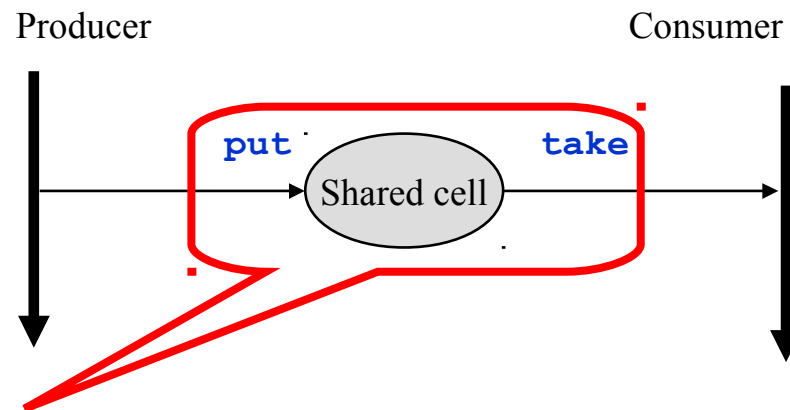
- Producer and Consumer threads are using a **shared object** (Shared Cell monitor) to interact in a dataflow fashion



- The “shared cell” (buffer) is a monitor
 - Methods **put** and **take** are synchronized to be executed with mutual exclusion.
 - An implicit condition variable (“wait set”) is used for condition synchronization of Producer and Consumer.

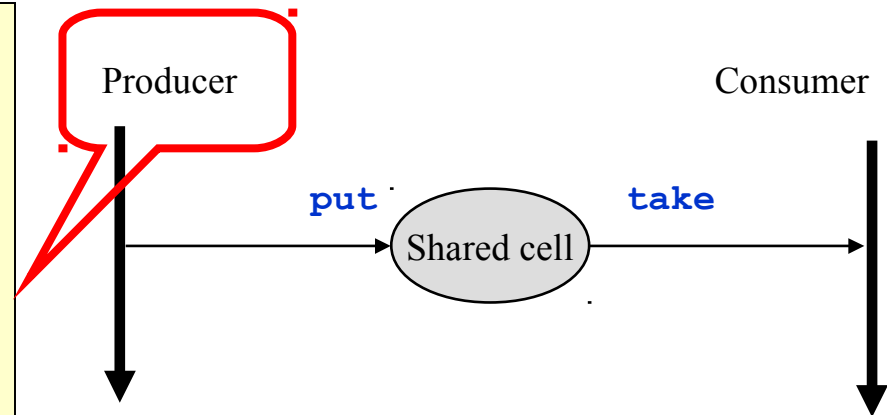
The Shared Cell Monitor

```
public class SharedCell {
    private int value;
    private boolean empty = true;
    public synchronized int take() {
        while (empty) {
            try {
                wait ();
            } catch (InterruptedException e) { }
        }
        empty = true;
        notify ();
        return value;
    }
    public synchronized void put(int value) {
        while (!empty) {
            try {
                wait ();
            } catch (InterruptedException e) { }
        }
        this.value = value;
        empty = false;
        notify ();
    }
}
```



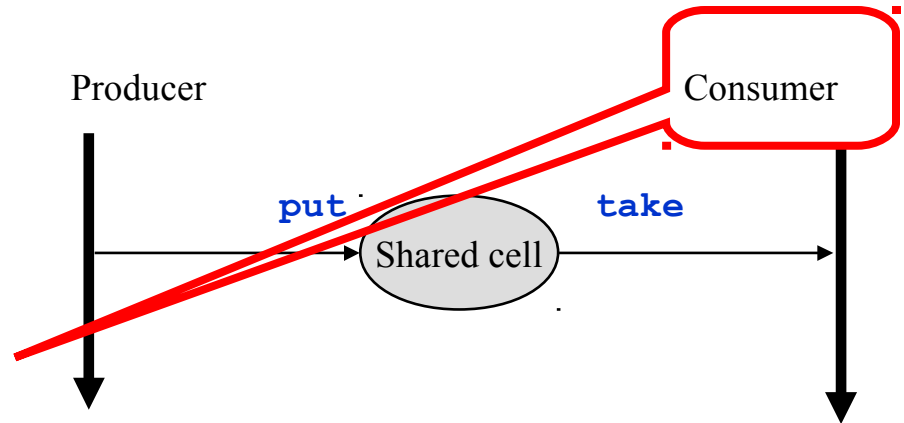
Producer Class

```
class Producer extends Thread {  
    private SharedCell cell;  
    private boolean Stop = false;  
    public Producer (SharedCell cell) {  
        this.cell = cell;  
    }  
    public void setStop () {  
        Stop = true;  
    }  
    public void run () {  
        int value;  
        while (!Stop) {  
            value = (int) (Math.random () * 100);  
            cell.put (value);  
            try {  
                sleep (value);  
            } catch (InterruptedException e) { }  
        }  
    }  
}
```



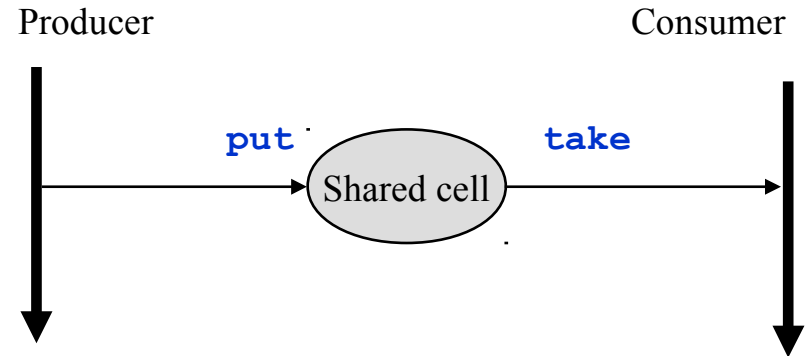
Consumer Class

```
class Consumer extends Thread {  
    private SharedCell cell;  
    private int n;  
    public Consumer(SharedCell cell, int n)  
    {  
        this.cell = cell;  
        this.n = n;  
    }  
    public void run () {  
        int value;  
        for (int i = 0; i < n; i++) {  
            value = cell.take ();  
            System.out.println ("Consumer: " +  
i + " value = " + value);  
        }  
    }  
}
```



A Test Application

```
public class Exchange {  
    public static void main(String args[])  
    {  
        SharedCell cell = new SharedCell ();  
        Producer p = new Producer (cell);  
        Consumer c = new Consumer (cell, 10);  
        p.start ();  
        c.start ();  
        try {  
            c.join ();  
        } catch (InterruptedException e) { };  
        p.setStop ();  
        p.interrupt();  
    }  
}
```



Example 2: Synchronized Bounded Buffer

```
public class Bounded_Buffer {  
    private Object[] items;  
    private int count = 0, front = 0, rear = 0;  
    private int n;  
  
    public Bounded_Buffer(int n) {  
        this.n = n;  
        items = new Object[n];  
    }  
}
```

Synchronized Bounded Buffer (cont'd)

```
public synchronized void put(Object x) {
    while (count == n)
        try { wait(); }
        catch (InterruptedException e) { }
    items[rear] = x; rear = (rear + 1) % n; count++;
    notifyAll();
}

public synchronized Object take() {
    while (count == 0)
        try { wait(); }
        catch (InterruptedException e) { }
    Object x = items[front];
    front = (front + 1) % n; count--;
    notifyAll();
    return x;
}
}
```

Java Concurrency Utilities: java.util.concurrent

- **Locks and Conditions**
- **Synchronizers**
 - General purpose synchronization classes, including semaphores, mutexes, barriers, latches, and exchangers
- **The Executor framework**
 - for scheduling, execution, and control of asynchronous tasks (**Runnable** objects)
- **Nanosecond-granularity timing**
 - The actual precision of **System.nanoTime** is platform-dependent
 - Used for time-stamps and time estimates

Concurrency Utilities: (cont'd)

- **Atomic Variables**

- Classes for atomically manipulating single variables (of primitive types and references)
- E.g. **AtomicBoolean**, **AtomicInteger**, **AtomicLong**
- For object references and arrays
- E.g. **AtomicReference<V>**,
AtomicMarkableReference<V>,
AtomicStampedReference<V>
- Used to implement concurrent collection classes

- **Concurrent Collections**

- Pools of items
- **Queue** and **BlockingQueue** interfaces
- Concurrent implementations of **Map**, **List**, and **Queue**.

Locks and Conditions

- **java.util.concurrent.locks**
 - Classes and interfaces for locking and waiting for conditions
- **ReentrantLock** class
 - Represents a reentrant mutual exclusion lock
 - Allows to create condition variables to wait for conditions
- **Condition** interface
 - Represents a condition variable associated with a lock
 - Allows one thread to suspend execution (to "wait") until notified by another thread
 - The suspended thread releases the lock
- **ReentrantLock** locks (like **synchronized** objects) are monitors
 - Allow blocking on a condition rather than spinning
- Threads:
 - acquire and release lock
 - wait on a condition

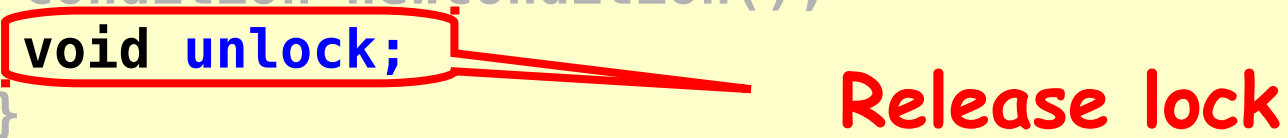
The Java Lock Interface (1/5)

```
public interface Lock {  
    void lock();  
    void lockInterruptibly() throws InterruptedException;  
    boolean tryLock();  
    boolean tryLock(long time, TimeUnit unit);  
    Condition newCondition();  
    void unlock;  
}
```

Acquire lock

The Java Lock Interface (2/5)

```
public interface Lock {  
    void lock();  
    void lockInterruptibly() throws InterruptedException;  
    boolean tryLock();  
    boolean tryLock(long time, TimeUnit unit);  
    Condition newCondition();  
void unlock;  
}
```



Release lock

The Java Lock Interface (3/5)

```
public interface Lock {  
    void lock();  
    void lockInterruptibly() throws InterruptedException;  
    boolean tryLock();  
    boolean tryLock(long time, TimeUnit unit):  
    Condition newCondition();  
    void unlock;  
}
```

Try for lock, but not too hard

The Java Lock Interface (4/5)

```
public interface Lock {  
    void lock();  
    void lockInterruptibly() throws InterruptedException;  
    boolean tryLock();  
    boolean tryLock(long time, TimeUnit unit);  
    Condition newCondition();  
    void unlock;  
}
```

Create condition to wait on

The Java Lock Interface (5/5)

```
public interface Lock {  
    void lock();  
void lockInterruptibly() throws InterruptedException;  
    boolean tryLock();  
    boolean tryLock(long time, TimeUnit unit);  
    Condition newCondition();  
    void unlock();  
}
```

Guess what this method does?

Lock Conditions (1/4)

```
public interface Condition {  
    void await();  
    boolean await(long time, TimeUnit unit);  
    ...  
    void signal();  
    void signalAll();  
}
```

Lock Conditions (2/4)

```
public interface Condition {
```

```
void await();
```

```
boolean await(long time, TimeUnit unit);
```

```
...
```

```
void signal();
```

```
void signalAll();
```

```
}
```

**Release lock and
wait on condition**

Lock Conditions (3/4)

```
public interface Condition {  
    void await();  
    boolean await(long time, TimeUnit unit);  
    void signal();  
    void signalAll();  
}
```

Wake up one waiting thread

Lock Conditions (4/4)

```
public interface Condition {  
    void await();  
    boolean await(long time, TimeUnit unit);  
    ...  
    void signal();  
    void signalAll();  
}
```

Wake up all waiting threads

Await, Signal and Signal All

q.await()

- Releases lock associated with **q**
- Sleeps (gives up processor)
- Awakens (resumes running) when signaled by **Signal** or **SignalAll**
- Reacquires lock & returns

q.signal();

- Awakens **one** waiting thread
 - Which will reacquire lock associated with **q**

q.signalAll();

- Awakens **all** waiting threads
 - Which will each reacquire lock associated with **q**

Example 3: Lock-Based Blocking Bounded Buffer

```
public class BoundedBuffer {
    final Lock lock = new ReentrantLock();
    final Condition notFull = lock.newCondition();
    final Condition notEmpty = lock.newCondition();
    final Object[] items;
    int rear, front, count, n;

    public BoundedBuffer(int n) {
        this.n = n;
        items = new Object[n];
    }
}
```

```

public void put(Object x) throws InterruptedException {
    lock.lock();
    try {
        while (count == n) notFull.await();
        items[rear] = x; rear = (rear + 1) % n; count++;
        notEmpty.signal();
    } finally {
        lock.unlock();
    }
}

public Object take() throws InterruptedException {
    lock.lock();
    try {
        while (count == 0) notEmpty.await();
        Object x = items[front];
        front = (front + 1) % n; count--;
        notFull.signal();
        return x;
    } finally {
        lock.unlock();
    }
}
}

```

The Executor Framework

- For scheduling, execution, and control of **asynchronous tasks** in concurrent threads according to a set of execution policies
- Allows creating an executor (a pool of threads) and assigning tasks to the executor
- An **Executor** object executes submitted tasks
- For example:

```
Executor e =  
    Executors.newFixedThreadPool (numThreads) ;  
e.execute (new RunnableTask1 () ) ;  
e.execute (new RunnableTask2 () ) ;
```

Executor Interfaces

- An executor can have one of the following interfaces:
- **Executor**
 - A simple interface to launch void Runnable tasks
 - `execute(Runnable)`
- **ExecutorService**
 - Executor subinterface with additional features to manage lifecycle
 - To launch and control void Runnable tasks and Callable tasks, which return results
 - `submit(Runnable)` , `submit(Callable<T>)` , `shutdown()` , `invokeAll(...)` , `awaitTermination(...)`
 - `Future<V>` represents the result of an asynchronous computation
- **ScheduledExecutorService**
 - ExecutorService subinterface with support for future or periodic execution
 - For scheduling Runnable and Callable tasks

Example: Using an Executer (a Thread Pool)

```
public class Handler implements Runnable {
    private Socket socket;
    public Handler(Socket socket) { this.socket = socket; }
    public void run() {
        try {
            BufferedReader rd = new BufferedReader(
                new InputStreamReader(socket.getInputStream()));
            PrintWriter wr = new PrintWriter(socket.getOutputStream());
            String str;
            while ((str = rd.readLine()) != null) {
                for ( int i=str.length(); i > 0; i-- ) wr.print(str.charAt(i-1));
                wr.println();
                wr.flush();
            }
            socket.close();
        } catch ( IOException e ) {;}
    }
}
```



```

public class ReverseServer {
    public static void main(String[] args) throws IOException {
        int poolSize = 3, port = 4444;
        ServerSocket serverSocket = null;
        try {
            if (args.length >1) poolSize = Integer.parseInt(args[1]);
            if (args.length >0) port = Integer.parseInt(args[0]);
        } catch (NumberFormatException e) {
            System.out.println("USAGE: java ReverseServer [poolSize] [port]");
            System.exit(1);
        }
        try {
            serverSocket = new ServerSocket(port);
        } catch (IOException e) {
            System.out.println("Can not listen on port: " + port);
            System.exit(1);
        }
        ExecutorService executor = Executors.newFixedThreadPool(poolSize);
        while (true) {
            Socket socket = serverSocket.accept();
            executor.execute( new Handler(socket) );
        }
    }
}

```

Java Collections Framework

- The **Java collections framework** (package `java.util`)
 - Includes collection interfaces and classes, e.g. `HashSet<E>`, `LinkedList<E>`
- A **collection** is an object that represents a group of elements (objects) of a specified type, i.e. `Vector<E>`
 - Operations: add, remove, put, replace, get, peek, poll, contains, size, list, isEmpty, etc.
- **Concurrent Collections** (`java.util.concurrent`)
 - Extends the Java Collection framework (`java.util`) with concurrent collections including the `Queue`, `BlockingQueue` and `BlockingDeque` interfaces, and high-performance, concurrent implementations of `Map`, `List`, and `Queue`.

Concurrent Collections

(`java.util.concurrent`)

- Concurrent versions of some collections
 - `ConcurrentHashMap<K, V>`
 - `CopyOnWriteArrayList`
 - `CopyOnWriteArraySet`
- Different from similar "synchronized" classes
- **A concurrent collection is thread-safe, but not governed by a single exclusion lock.**
 - For example, `ConcurrentHashMap`, safely permits any number of concurrent reads as well as a tunable number of concurrent writes.

Unsynchronized, Synchronized, Concurrent Collections

- When to use which
- Unsynchronized collections
 - preferable when either collections are unshared, or are accessible only when holding other locks.
- "Synchronized" versions
 - when you need to govern all access to a collection via a single lock, at the expense of poorer scalability.
- "Concurrent" versions
 - normally preferable when multiple threads are expected to access a common collection.