Lecture 9: Java Threads Programming 2

NCCU 高等軟體設計

Fall 2005

Nov. 15, 2005

Topics

- Review: Thread Synchronization in Java
- Readers-Writers Problem
- Implementing Shared Queues
- Socket Introduction
- Server programming
  - Single threaded server
  - One thread per request
  - Thread pool
  - Worker thread Pattern
  - Executor Framework in java.util.concurrent (Java 5)

Concurrency Control in Java

- Mutual Exclusion
  - Object lock & entry set (queue)
  - Synchronized blocks
  - Synchronized methods
- Conditional Synchronization
  - Wait set
  - wait, notify, notifyAll (methods of the *Object* class)
  - Issue: Single wait set, no conditional variables

Wait and Notify: Code

- Consumer:

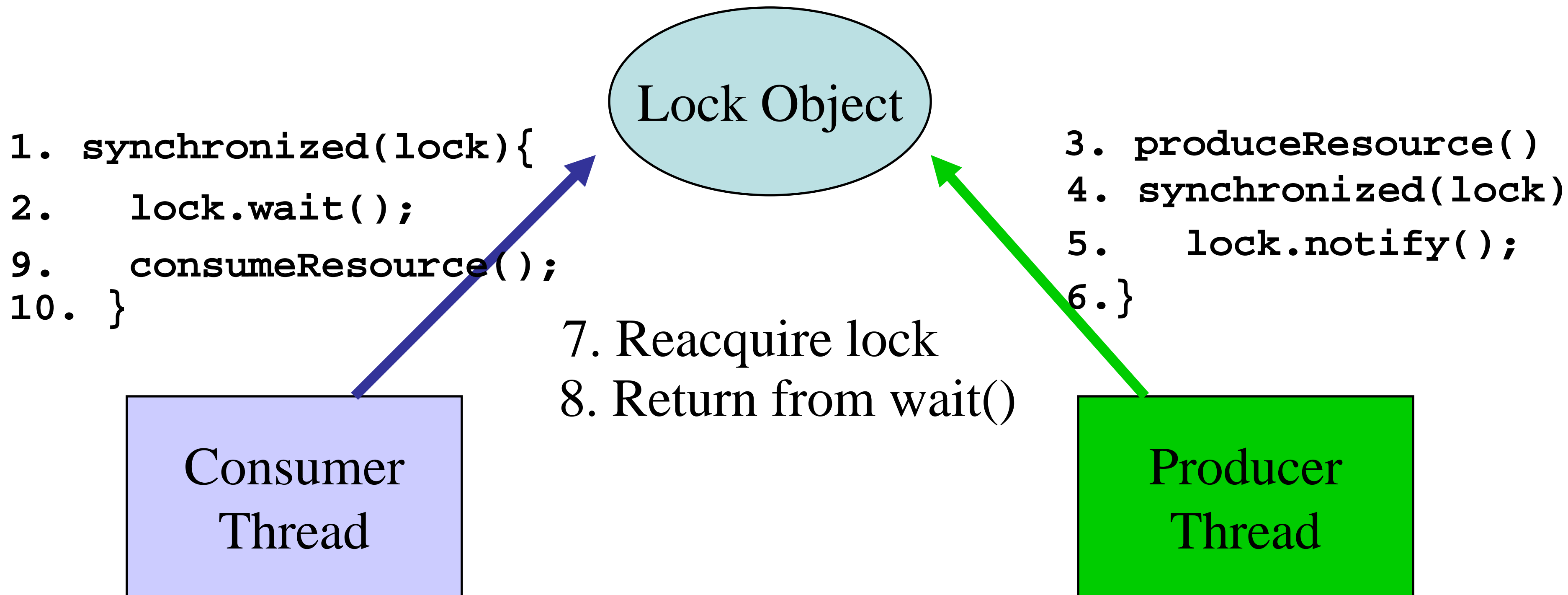
```
synchronized (lock) {
  while (!resourceAvailable()) {
    lock.wait();
  }
  consumeResource();
}
```

Wait and Notify: Code

- Producer:

```
produceResource();
synchronized (lock) {
     lock.notifyAll();
}
```

Wait/Notify Sequence



Wait/Notify Sequence

```
1. synchronized(lock){
2.    lock.wait();
9.    consumeResource();
10. }
```

```
3. produceResource()
4. synchronized(lock)
5.    lock.notify();
6.}
```

7. Reacquire lock
8. Return from wait()

Consumer Thread

Producer Thread

Wait/Notify Sequence

Lock Object

```
1. synchronized(lock){
2.   lock.wait();
9.   consumeResource();
10. }
```

```
3. produceResource()
4. synchronized(lock)
5.   lock.notify();
6.}
```

7. Reacquire lock
8. Return from wait()

Consumer Thread

Producer Thread

Wait/Notify Sequence

Lock Object

```
1. synchronized(lock){
2.   lock.wait();
9.   consumeResource();
10. }
```

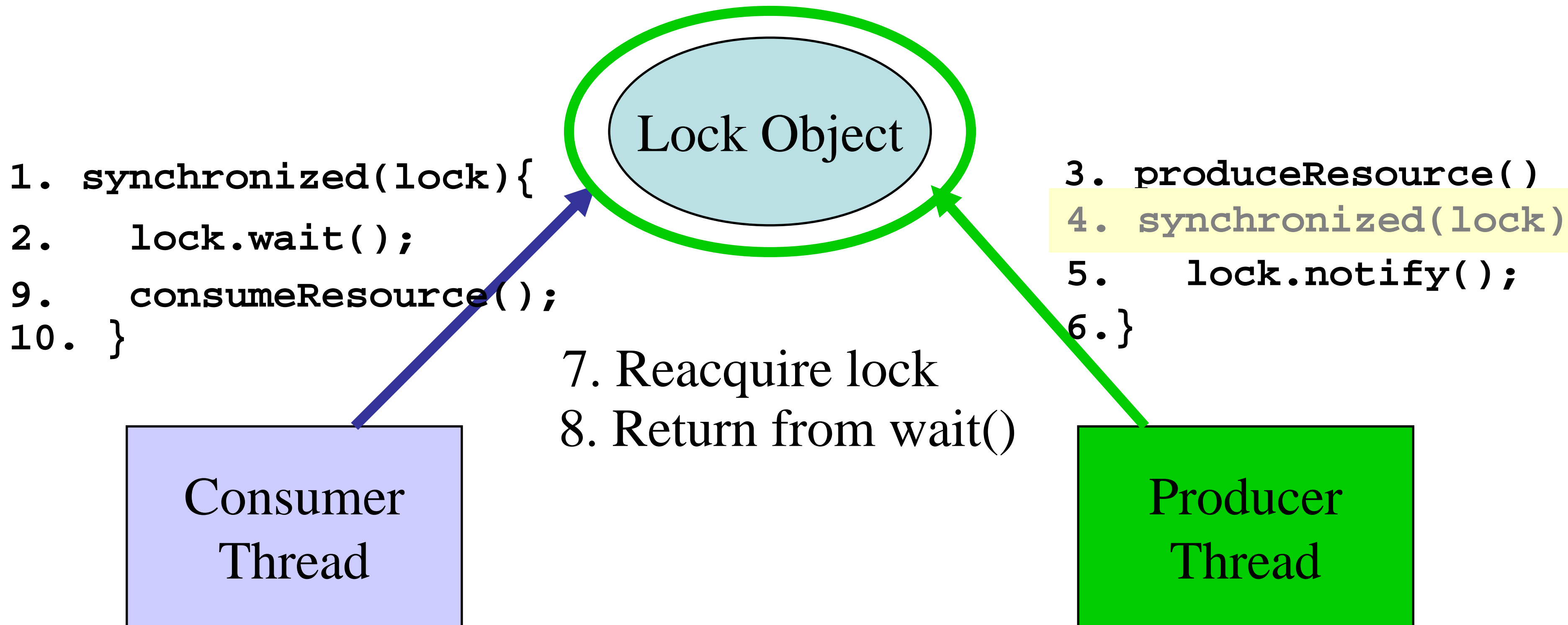
```
3. produceResource()
4. synchronized(lock)
5.   lock.notify();
6.}
```

7. Reacquire lock
8. Return from wait()

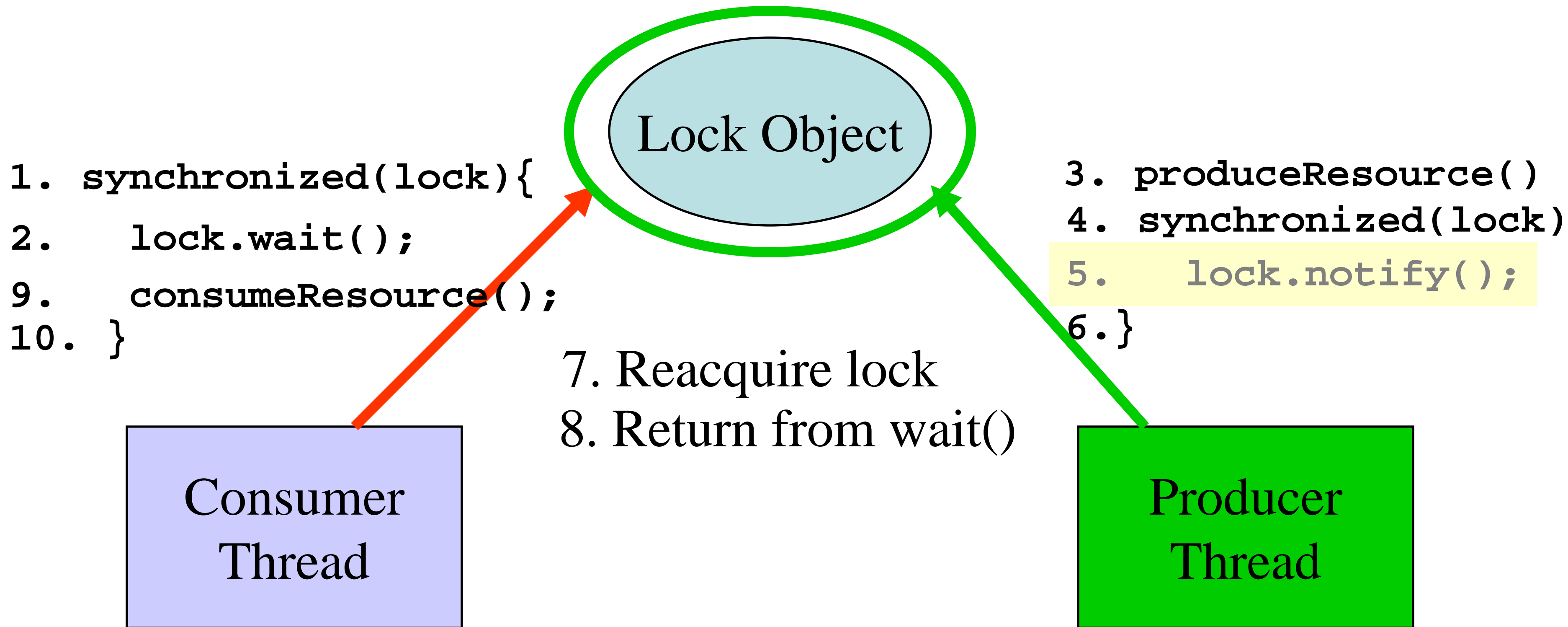
Consumer Thread

Producer Thread

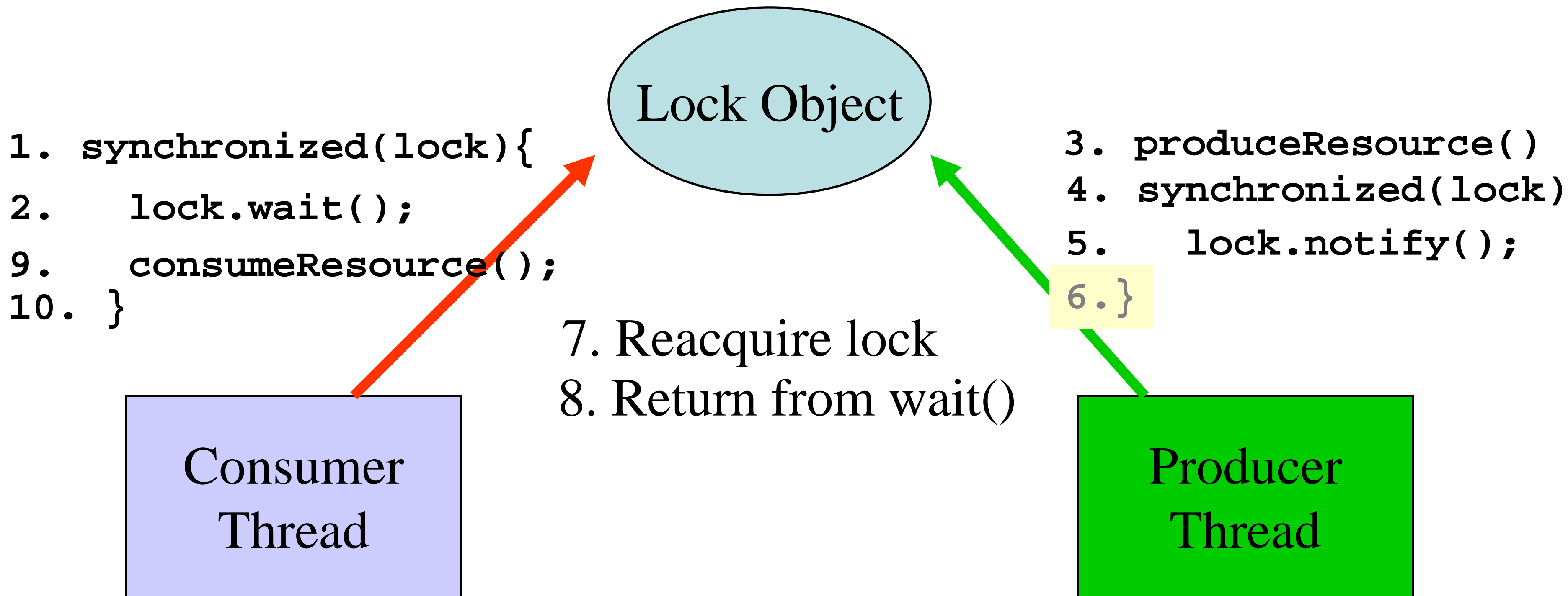
Wait/Notify Sequence



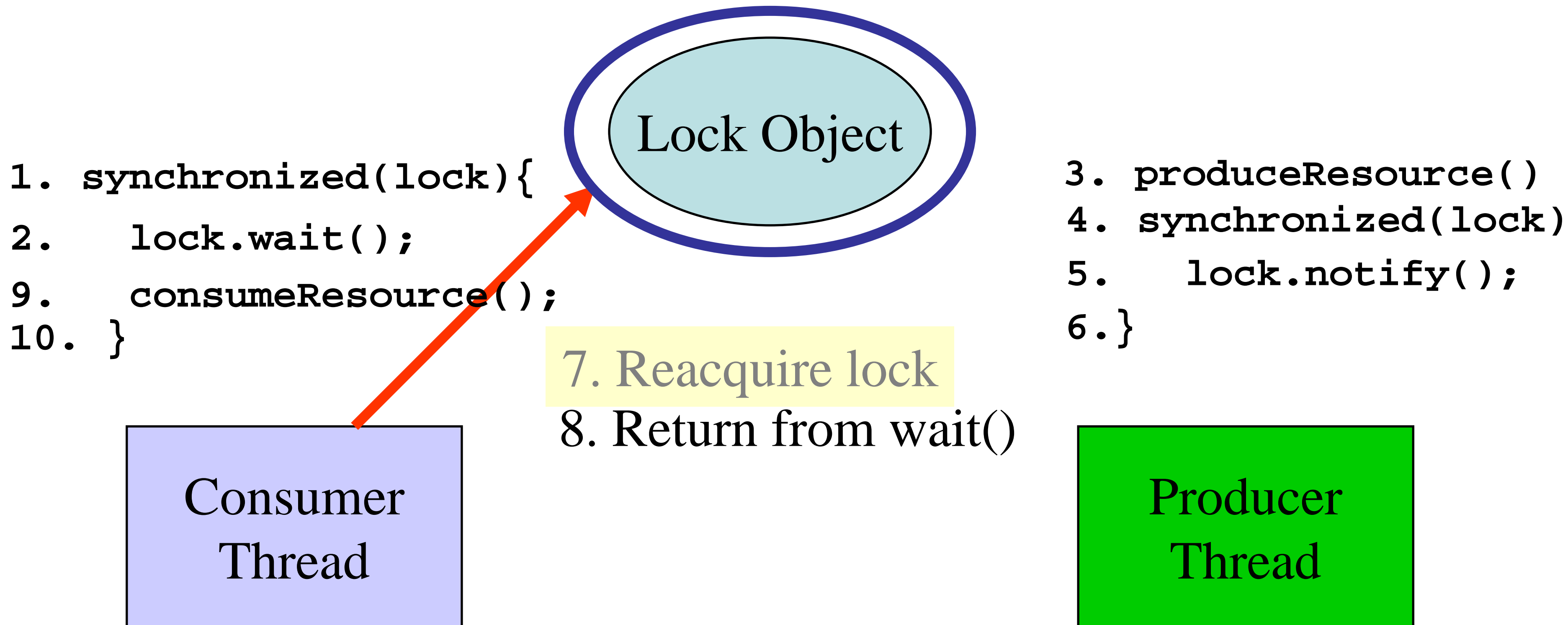
Wait/Notify Sequence



Wait/Notify Sequence



Wait/Notify Sequence



Wait/Notify Sequence

```
1. synchronized(lock){
2.   lock.wait();
9.   consumeResource();
10. }
```

```
3. produceResource()
4. synchronized(lock)
5.   lock.notify();
6.}
```

7. Reacquire lock

8. Return from wait()

Consumer Thread

Producer Thread

Wait/Notify Sequence

```
1. synchronized(lock){
2.   lock.wait();
9.   consumeResource();
10. }
```

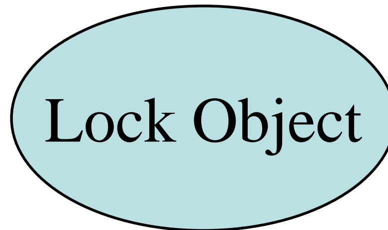
```
3. produceResource()
4. synchronized(lock)
5.   lock.notify();
6.}
```

7. Reacquire lock
8. Return from wait()

Consumer Thread

Producer Thread

Wait/Notify Sequence




```
1. synchronized(lock){
2.    lock.wait();
9.    consumeResource();
10. }
```

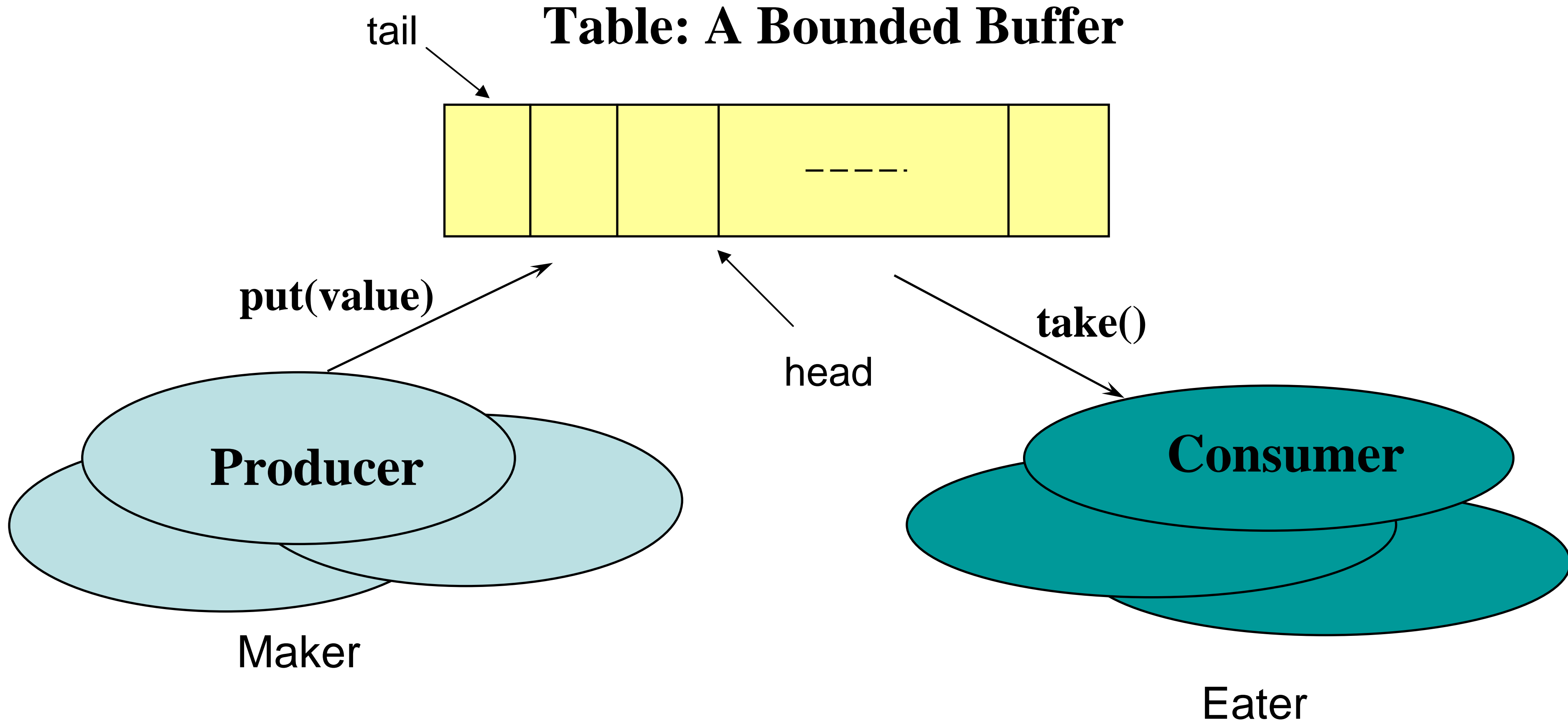
```
3. produceResource()
4. synchronized(lock)
5.    lock.notify();
6.}
```

7. Reacquire lock
8. Return from wait()

Consumer Thread

Producer Thread

Bounded Buffer Problem (Producer & Consumer)



```
public class Table {
    private final String[] buffer;
    private int tail;         // 下一個put的地方
    private int head;        // 下一個take的地方
    private int count;      // buffer內的蛋糕數
    public Table(int count) {
        this.buffer = new String[count];
        this.head = 0;         this.tail = 0;         this.count = 0;
    }
   public synchronized void put(String cake) throws InterruptedException {// 放置蛋糕
        System.out.println(Thread.currentThread().getName() + " puts " + cake);
        while (count >= buffer.length) {       wait();          }
        buffer[tail] = cake;
        tail = (tail + 1) % buffer.length;
        count++;
        notifyAll();
    }
    public synchronized String take() throws InterruptedException { //取得蛋糕
        while (count <= 0) {       wait();      }
        String cake = buffer[head];
        head = (head + 1) % buffer.length;
        count--;
        notifyAll();
        System.out.println(Thread.currentThread().getName() + " takes " + cake);
        return cake;
    }
}
```

```
public class MakerThread extends Thread {
    private final Random random;
    private final Table table;
    private static int id = 0; // 蛋糕的流水號(所有廚師共通)
    public MakerThread(String name, Table table, long seed) {
         super(name);
         this.table = table;
         this.random = new Random(seed);
    }
    public void run() {
        try {
            while (true) {
                Thread.sleep(random.nextInt(1000));
                String cake = "[ Cake No." + nextId() + " by " + getName() + " ]";
                table.put(cake);
            }
        } catch (InterruptedException e) {
        }
    }
    private static synchronized int nextId() {
        return id++;
    }
}
```

```java
public class EaterThread extends Thread {
    private final Random random;
    private final Table table;
    public EaterThread(String name, Table table, long seed) {
        super(name);
        this.table = table;
        this.random = new Random(seed);
    }
    public void run() {
        try {
            while (true) {
                String cake = table.take();
                Thread.sleep(random.nextInt(1000));
            }
        } catch (InterruptedException e) {
        }
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        Table table = new Table(3);     // 建立可以放置3個蛋糕的桌子
        new MakerThread("MakerThread-1", table, 31415).start();
        new MakerThread("MakerThread-2", table, 92653).start();
        new MakerThread("MakerThread-3", table, 58979).start();
        new EaterThread("EaterThread-1", table, 32384).start();
        new EaterThread("EaterThread-2", table, 62643).start();
        new EaterThread("EaterThread-3", table, 38327).start();
    }
}
```

Looks So Easy?

- Concurrent programming is far from easy!
- Full of pitfalls and traps
- Appreciate the object locks built into Java
- Try to write a mutex lock by ourselves.

Self-Implemented Mutex Locks

Consider the following naïve implementation of Mutex lock:

```
public final class Mutex {
    private boolean busy = false;
    public synchronized void lock() {
        while (busy) {
            try {
                wait();
            } catch (InterruptedException e) {
            }
        }
        busy = true;
    }
    public synchronized void unlock() {
        busy = false;
        notifyAll();
    }
}
```

What's wrong with This Mutex class?

```
void method(…)
{
    mutex.lock();

    // critical section
      …

    mutex.unlock();
}
```

Self-Implemented Mutex Locks

Consider the following naïve implementation of Mutex lock:

```
public final class Mutex {
    private boolean busy = false;
    public synchronized void lock() {
        while (busy) {
            try {
                wait();
            } catch (InterruptedException e) {
            }
        }
        busy = true;
    }
    public synchronized void unlock() {
        busy = false;
        notifyAll();
    }
}
```

- Unlock() must be executed after lock()

```
void method(…)
{
    mutex.lock();
    try {
      // critical section
      …

    } finally {
       mutex.unlock();
    }
}
```

More Problems with the Mutex Class

1. What if a thread calls lock() twice continuously without calling the unlock()?
2. Unlock() can be called without calling lock() first.

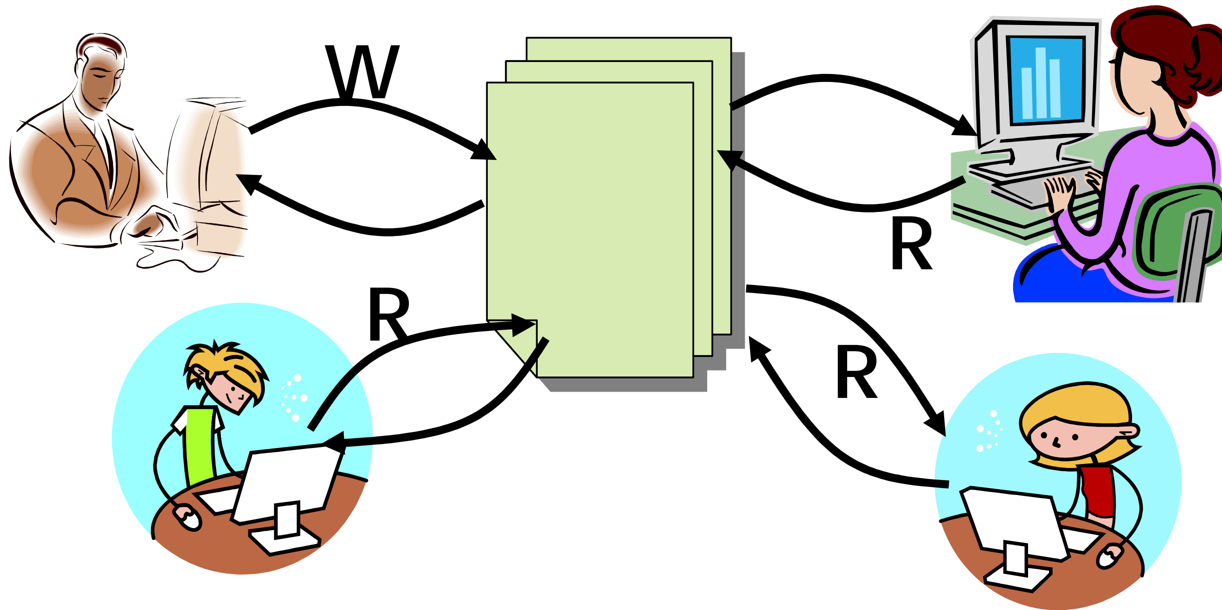
```
public final class Mutex {
    private boolean busy = false;
    public synchronized void lock() {
        while (busy) {
            try {
                wait();
            } catch (InterruptedException e) { }
        }
        busy = true;
    }
    public synchronized void unlock() {
         busy = false;
         notifyAll();
    }
}
```

Refined Mutex Class

```
public final class Mutex {
    private long locks = 0;
    private Thread owner = null;
    public synchronized void lock() {
        Thread me = Thread.currentThread();
        while (locks > 0 && owner != me) {
            try {
                wait();
            } catch (InterruptedException e) {   }
        }
        // locks == 0 || owner == me
        owner = me;
        locks++;
    }
}
```

```
public synchronized void unlock() {
        Thread me = Thread.currentThread();
        if (locks == 0 || owner != me) {
            return;
        }
        // locks > 0 && owner == me
        locks--;
        if (locks == 0) {
            owner = null;
            notifyAll();
        }
    }
```

Readers/Writers Problem



- Motivation: Consider a shared database
  - Two classes of users:
    - Readers – never modify database
    - Writers – read and modify database
  - Is using a single lock on the whole database sufficient?
    - Like to have many readers at the same time
    - Only one writer at a time

No!

Single-Lock Solution

- Not good for many situations.

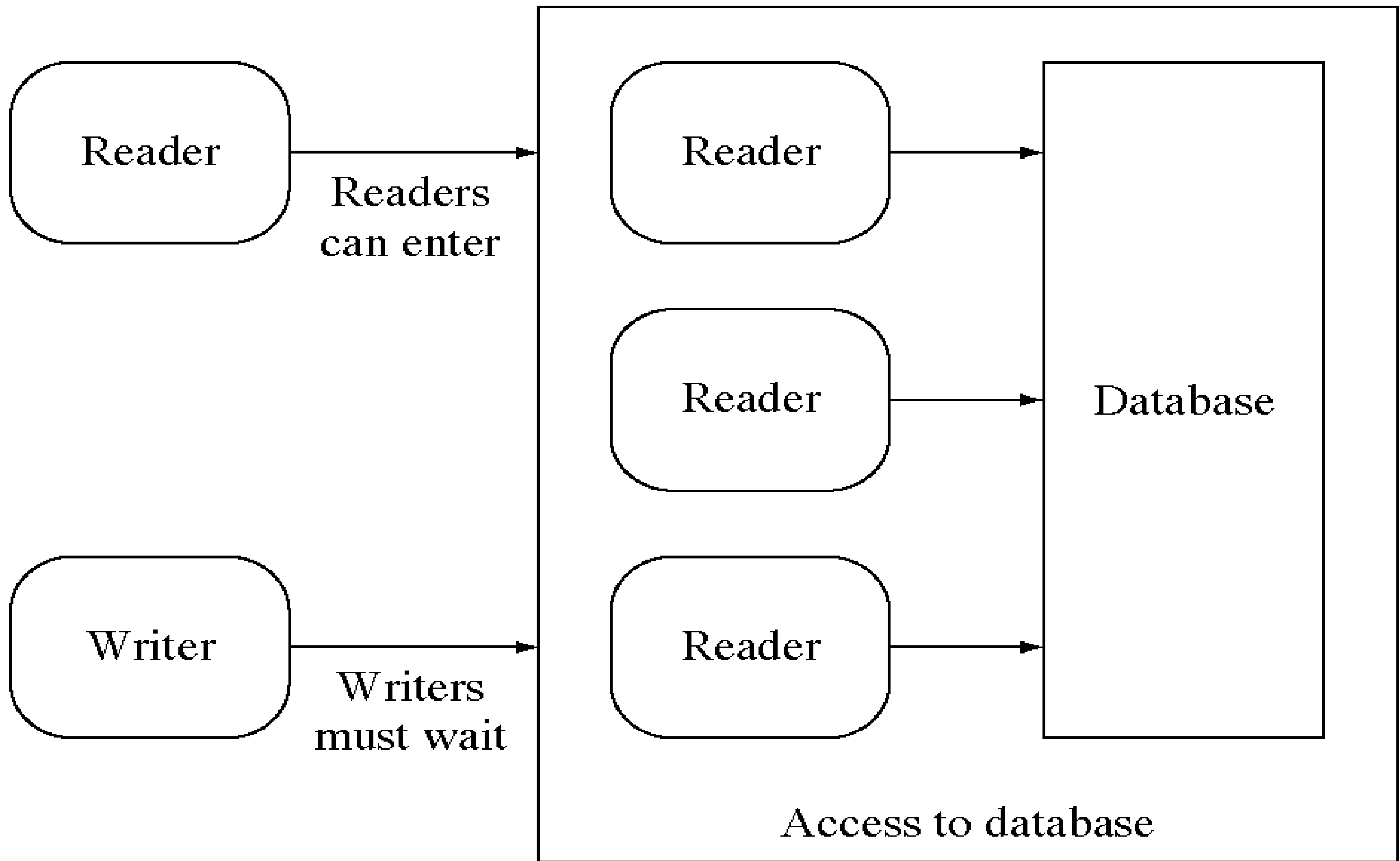
Number of readers >> Numbers of writers

```
public class Data {
    private final char[] buffer;
    public Data(int size) {
        this.buffer = new char[size];
        for (int i = 0; i < buffer.length; i++) {
            buffer[i] = '*';
        }
    }
    public synchronized char[] read() throws InterruptedException {
        return doRead();
    }
    public synchronized void write(char c) throws InterruptedException {
        doWrite(c);
    }
    private char[] doRead() {  … }
    private void doWrite(char c) { …}
   …
}
```

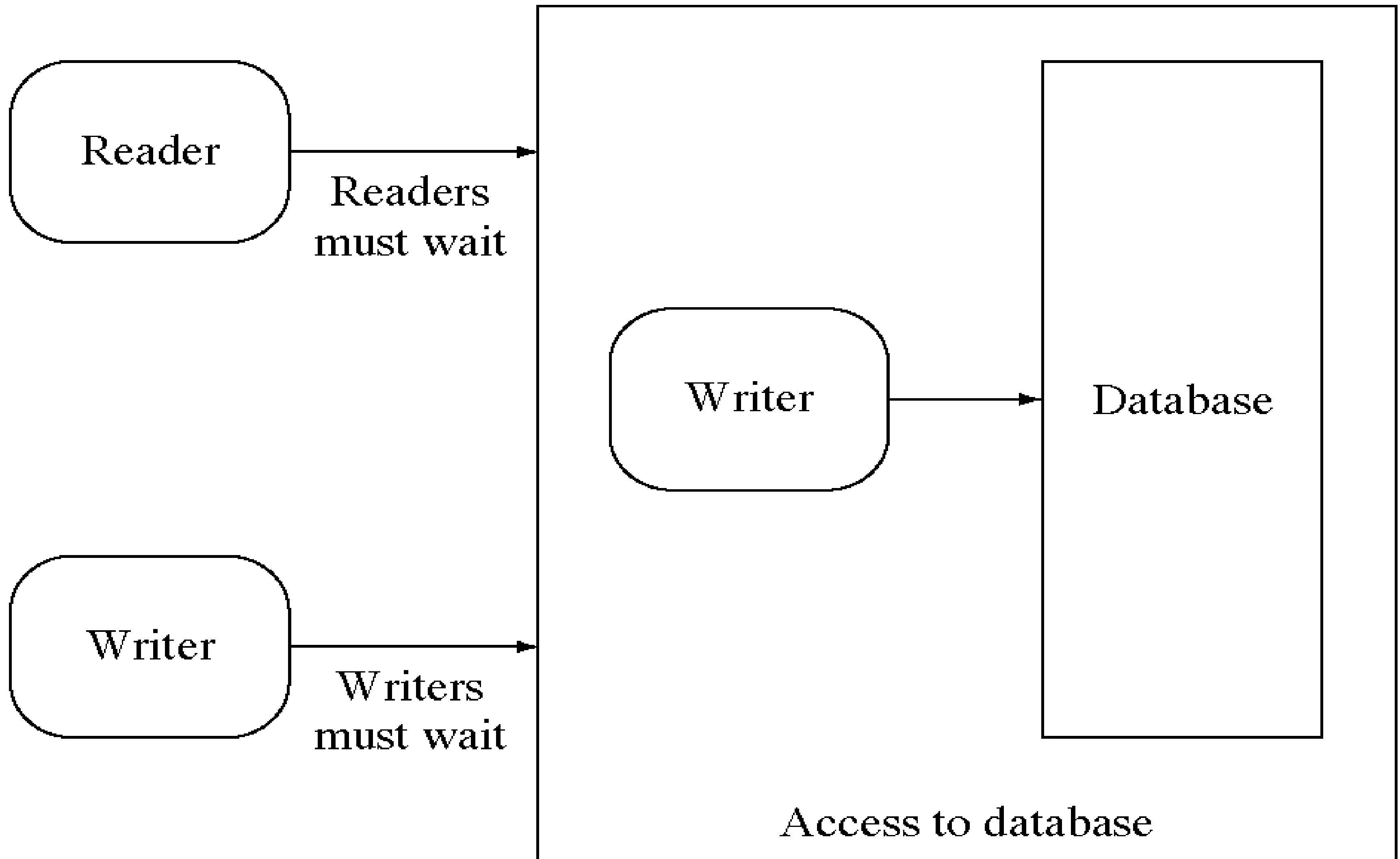
Readers-Writers Problem

- Any number of reader activities and writer activities are running.
- At any time, a reader activity may wish to read data.
- At any time, a writer activity may want to modify the data.
- During the time a writer is writing, no other reader or writer may access the shared data.
- Any number of readers may access the data simultaneously.

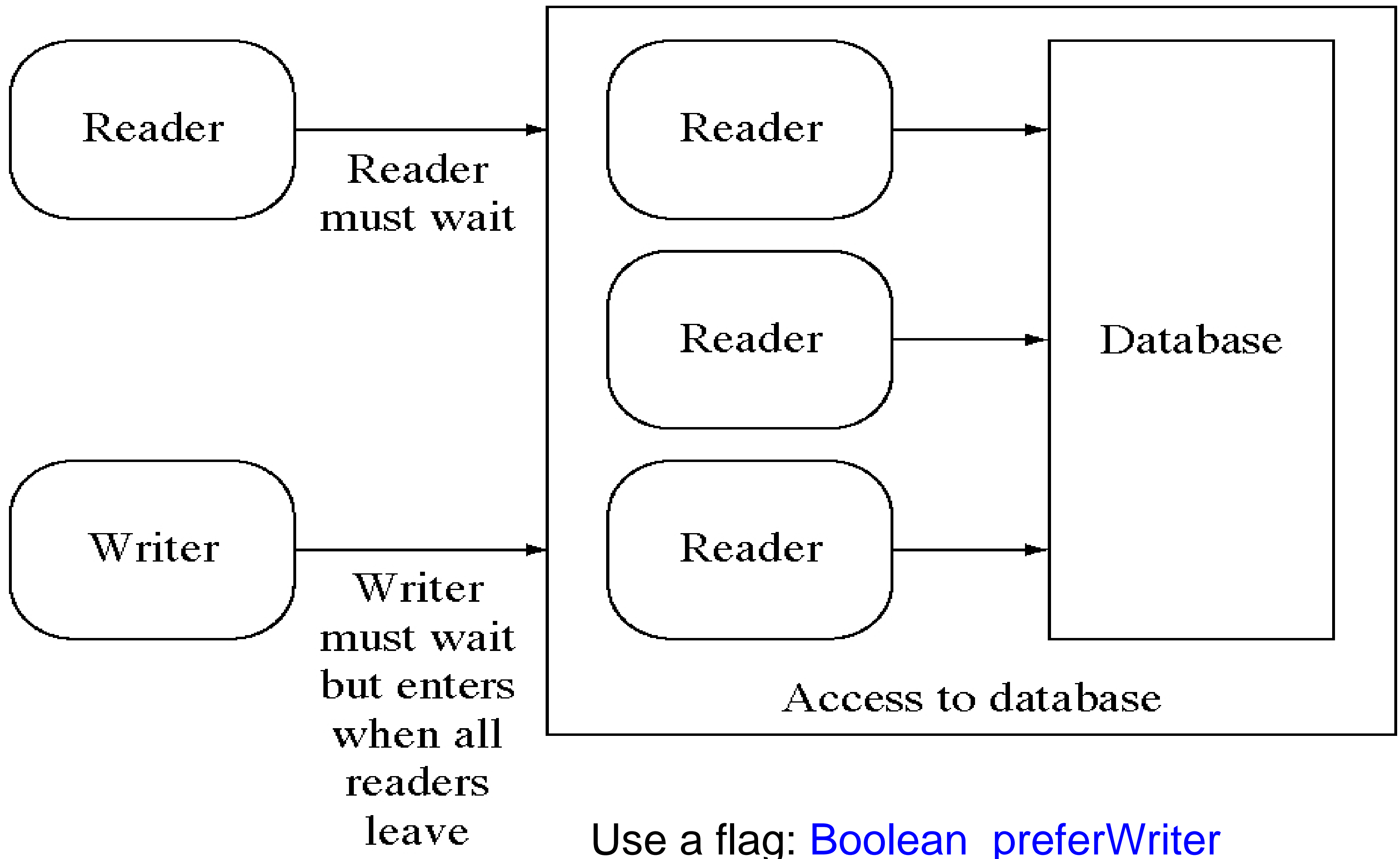
Readers-Writers with active readers



Readers-writers with an active writer



Should readers wait for waiting writer?



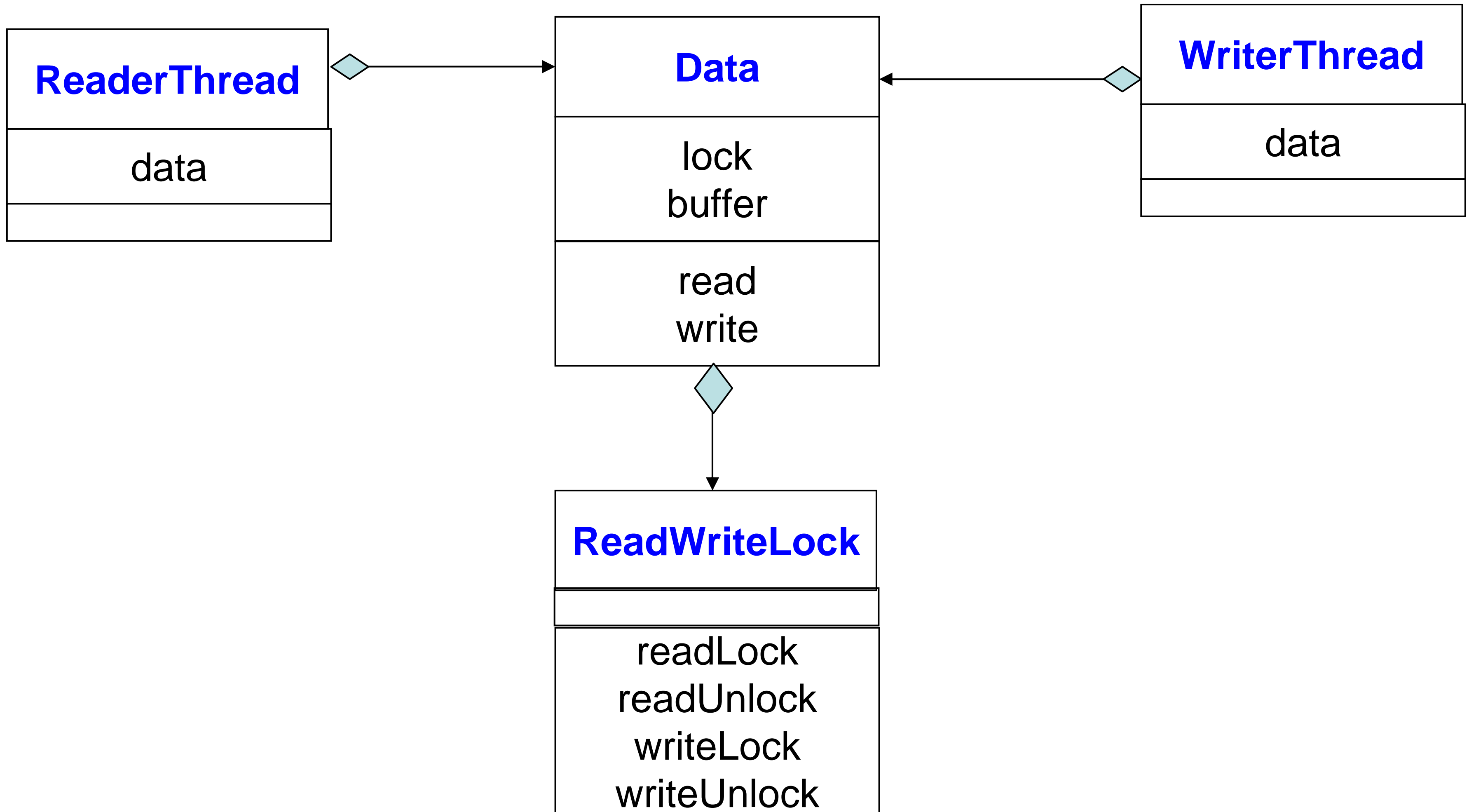
Use a flag: Boolean preferWriter

Readers-Writers problem

- Use multiple locks
  - readlock, writelock
- There are various versions with different readers and writers preferences:

1. The **first** readers-writers problem (A), requires that no reader will be kept waiting unless a writer has obtained access to the shared data.
2. The **second** readers-writers problem (B), requires that once a writer is ready, no new readers may start reading.
3. In a solution to the **first** case writers may starve; In a solution to the **second** case readers may starve.
4. Use a boolean flag: *preferWriter* to switch between A and B.

Participants & Class Diagram



Basic Ideas

- Basic structure of a solution:
  - Reader()
    ```
    Wait until no writers
    Access data base
    Check out – wake up a waiting writer
    ```
  - Writer()
    ```
    Wait until no active readers or writers
    Access database
    Check out – wake up waiting readers or writer
    ```
- State variables (Protected by a lock called "lock):
  - int ReadingReaders: Number of active readers; initially = 0
  - int WaitingReaders: Number of waiting readers; initially = 0
  - int WritingWriter: Number of active writers; initially = 0
  - int WaitingWriter: Number of waiting writers; initially = 0

  - Boolean PreferWriter

Solution A

```
public final class ReadWriteLock {
  private int readingReaders = 0; // (a)...實際正在讀取的執行緒數量
  private int writingWriters = 0; // (b)...實際正在寫入的執行緒數量

  public synchronized void readLock() throws InterruptedException {
    while (writingWriters > 0) {
        wait();
    }
    readingReaders++;                 // (a)實際正在讀取的執行緒數量加1
  }

  public synchronized void readUnlock() {
    readingReaders--;                 // (a)實際正在讀取的執行緒數量減1
    notifyAll();
  }

  public synchronized void writeLock() throws InterruptedException {
    while (readingReaders > 0 || writingWriters > 0) {
       wait();
    }
    writingWriters++;                 // (b)實際正在寫入的執行緒數量加1
  }

  public synchronized void writeUnlock() {
    writingWriters--;                 // (b)實際正在寫入的執行緒數量減1
    notifyAll();
  }
}
```

writers may starve

Alternative Solutions

1. The **first** readers-writers problem (A), requires that no reader will be kept waiting unless a writer has obtained access to the shared data.
2. The **second** readers-writers problem (B), requires that once a writer is ready, no new readers may start reading.
3. In a solution to the **first** case writers may starve; In a solution to the **second** case readers may starve.
4. Use a boolean flag: *preferWriter* to switch between A and B.

```java
public class Data {
    private final char[] buffer;
    private final ReadWriteLock lock = new ReadWriteLock();
    public Data(int size) {
        this.buffer = new char[size];
        for (int i = 0; i < buffer.length; i++) {
            buffer[i] = '*';
        }
    }
    public char[] read() throws InterruptedException {
        lock.readLock();
        try {           return doRead();
        }  finally {     lock.readUnlock();          }
    }
    public void write(char c) throws InterruptedException {
        lock.writeLock();
        try {           doWrite(c);
        } finally {      lock.writeUnlock();          }
    }
    private char[] doRead() {
        char[] newbuf = new char[buffer.length];
        for (int i = 0; i < buffer.length; i++) {    newbuf[i] = buffer[i];      }
        slowly();
        return newbuf;
    }
    private void doWrite(char c) {
        for (int i = 0; i < buffer.length; i++) {     buffer[i] = c;      slowly();
        }
    }
```

```java
private void slowly() {
        try {
                Thread.sleep(50);
        } catch (InterruptedException e) {
        }
    }
```

```
public final class ReadWriteLock {
    private int readingReaders = 0; // (A)...實際正在讀取的執行緒數量
    private int waitingWriters = 0;   // (B)...正在等待寫入的執行緒數量
    private int writingWriters = 0;   // (C)...實際正在寫入的執行緒數量
    private boolean preferWriter = true; // 寫入優先的話，值爲true
    public synchronized void readLock() throws InterruptedException {
        while (writingWriters > 0 || (preferWriter && waitingWriters > 0)) {
                wait();             }
        readingReaders++;                          // (A)實際正在讀取的執行緒數量加1
    }
    public synchronized void readUnlock() {
        readingReaders--;                          // (A)實際正在讀取的執行緒數量減1
        preferWriter = true;
        notifyAll();
    }
    public synchronized void writeLock() throws InterruptedException {
        waitingWriters++;                          // (B)正在等待寫入的執行緒數量加1
        try {
            while (readingReaders > 0 || writingWriters > 0) {
                wait();           }
        } finally {
            waitingWriters--;                      // (B)正在等待寫入的執行緒數量減1
        }
        writingWriters++;                          // (C)實際正在寫入的執行緒數量加1
    }
    public synchronized void writeUnlock() {
        writingWriters--;                          // (C)實際正在寫入的執行緒數量減1
        preferWriter = false;
        notifyAll();
    }
}
```

```
public class WriterThread extends Thread {
    private static final Random random = new Random();
    private final Data data;
    private final String filler;
    private int index = 0;
    public WriterThread(Data data, String filler) {
        this.data = data;          this.filler = filler;
    }
    public void run() {
        try {
            while (true) {
                char c = nextchar();
                data.write(c);
                Thread.sleep(random.nextInt(3000));
            }
        } catch (InterruptedException e) {    }
    }
    private char nextchar() {
        char c = filler.charAt(index);
        index++;
        if (index >= filler.length()) {   index = 0;     }
        return c;
    }
}
```

```
public class ReaderThread extends Thread {
    private final Data data;
    public ReaderThread(Data data) {
        this.data = data;
    }
    public void run() {
        try {
            while (true) {
                char[] readbuf = data.read();
                System.out.println(Thread.currentThread().getName() + " reads " +
                      String.valueOf(readbuf));
            }
        } catch (InterruptedException e) {
        }
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Data data = new Data(10);
        new ReaderThread(data).start();
        new ReaderThread(data).start();
        new ReaderThread(data).start();
        new ReaderThread(data).start();
        new ReaderThread(data).start();
        new ReaderThread(data).start();
        new WriterThread(data, "ABCDEFGHIJKLMNOPQRSTUVWXYZ").start();
        new WriterThread(data, "abcdefghijklmnopqrstuvwxyz").start();
    }
}
```

Implementations of Shared Queues

From Single-Threaded to Multi-Threaded Queues

A Simple Queue (No threads)

```
import java.util.*;
public class Queue {
    private LinkedList list = new LinkedList();
    public void add(Object v) {
        list.addFirst(v);              // not synchronized
    }
    public Object remove() {
        return list.removeLast();  // not synchronized
    }
    public boolean isEmpty() {
        return list.isEmpty();
    }
    public static void main(String[] args) {
        Queue queue = new Queue();
         for (int i = 0; i < 10; i++)
            queue.add(Integer.toString(i));
        while (!queue.isEmpty())
            System.out.println(queue.remove());
  }
}
```

interface java.util.List

- `boolean add(Object x)`
  // appends `x` to the end of list; returns true
- `int size()`

  // returns the number of elements in this list
- `Object get(int index)`
  // returns the element at the specified position in this list.
- `Object set(int index, Object x)`
  // replaces the element at `index` with `x`
  // returns the element formerly at the specified position
- `Iterator iterator()`
- `ListIterator listIterator()`
- …
- LinkedList
  - `public void addFirst(Object o) {`
  - `public Object removeLast()`

Sun Documentation: LinkedList

```
java.util
Class LinkedList

java.lang.Object
  |
  +--java.util.AbstractCollection
        |
        +--java.util.AbstractList
              |
              +--java.util.AbstractSequentialList
                    |
                    +--java.util.LinkedList

All Implemented Interfaces:
    Cloneable, Collection, List, Serializable

public class LinkedList
extends AbstractSequentialList
implements List, Cloneable, Serializable
```

Blocking Queues for Multi-threaded Env.

Using inheritance

```
class BlockingQueue extends Queue {
  public synchronized Object remove() {
    while (isEmpty()) {
       wait();      // really this.wait()
    }
    return super.remove();
  }
  public synchronized void add(Object o) {
      super.add(o);
      notifyAll();  // this.notifyAll()
  }
}
```

Can we have the option of non-blocking remove?

A Flexible Queue: non-blocking option

```
package org.apache.commons.threadpool; //Apache project
import java.util.LinkedList;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

public class MTQueue {
    /** The Log to which logging calls will be made. */
    private Log log = LogFactory.getLog(MTQueue.class);
    private LinkedList list = new LinkedList(); // not thread-safe
    private long defaultTimeout = 10000;
    public MTQueue() { }

    public synchronized int size() {
        return list.size();
    }
    public synchronized void add(Object object) {
         list.add( object );
         notify();
    }
```

```
public synchronized Object removeNoWait() {
        if ( ! list.isEmpty() ) {
            return list.removeFirst();
        }
        return null;
    }
}
```

```
public synchronized Object remove() {
    while (true) {
        Object answer = removeNoWait();
        if ( answer != null )  {
            return answer;     }
        try {
             wait( defaultTimeout );
        }
        catch (InterruptedException e) {
            log.error( "Thread was interrupted: " + e, e );
        }
    }
  }
  public synchronized Object remove(long timeout) {
      Object answer = removeNoWait();
      if (answer == null)  {
          try {
                wait( timeout );  }
          catch (InterruptedException e) {
              log.error( "Thread was interrupted: " + e, e );
          }
          answer = removeNoWait();
      }
      return answer;
  }
}
```

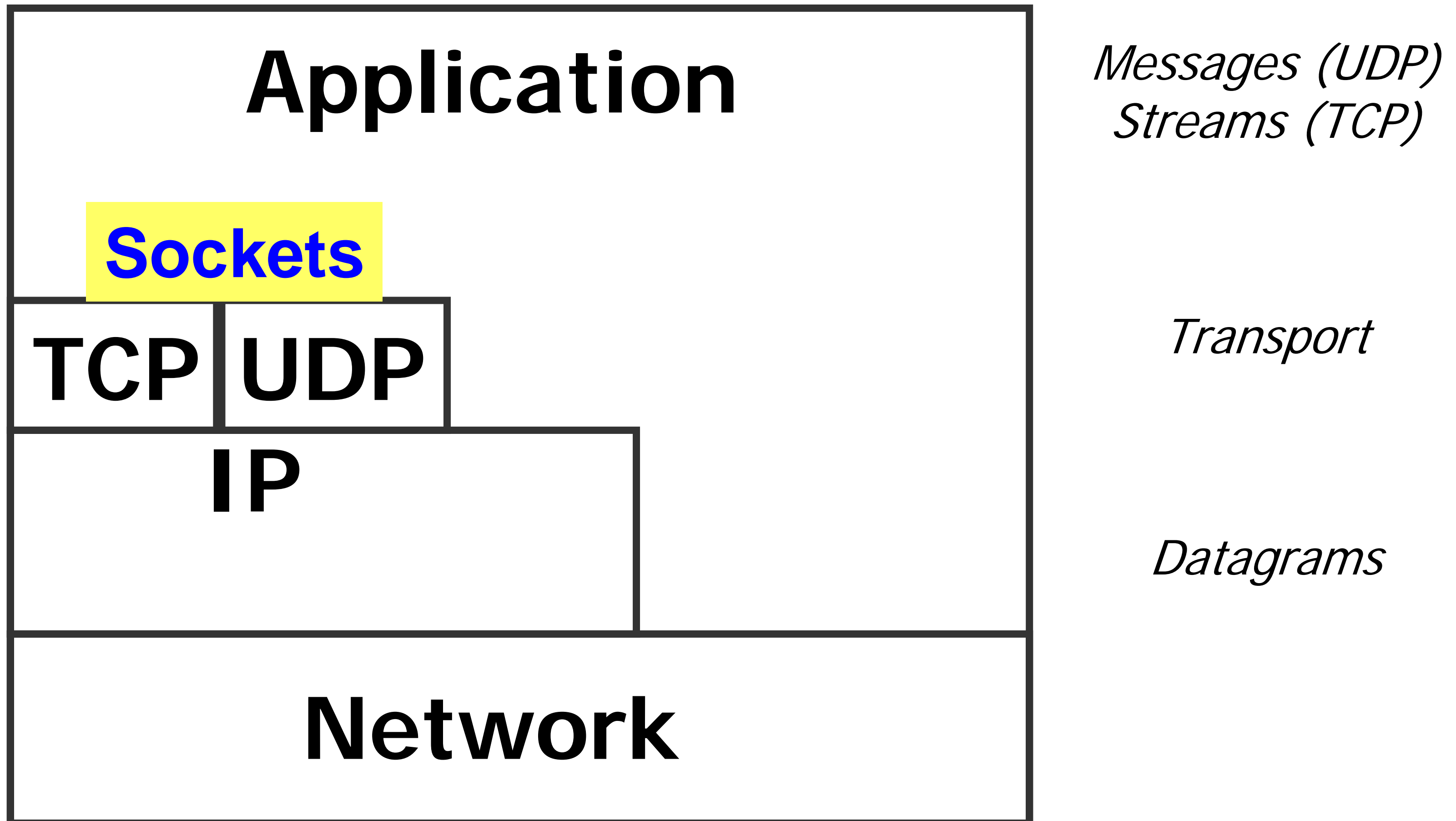
Bounded FIFO Queue

```
class Queue { //thread safe
  int c;       // capacity of queue
  int[] q;     // contents of queue
  int first=0;  // index of element at head of queue
  int n=0;      // number of elements in queue
  Queue(int c1) {
     c=c1;      q = new int[c];
  }
 synchronized void insert(int i) {
     while (n==c) {  // full
        try { wait(); }
       catch (InterruptedException e) {}
    }
    q[(first+n)%c] = i;
    n++;
    // if a thread might be waiting to remove an element, wake everyone up.
    if (n==1) notifyAll();
 }
 synchronized int remove() {
    while (n==0) {
        try { wait(); }
        catch (InterruptedException e) {}
    }
    int i = q[first];
    first++;
    n--;
    // if a thread might be waiting to insert an element, wake everyone up.
    if (n==c-1) notifyAll();
    return i;
 }
}
```

Sockets

- 源自加州柏克萊大學(UC Berkeley)
- 爲BSD UNIX 4.1版所添加的網路通訊系統呼叫
- UNIX的標準設施
- 稱爲Berkeley Sockets (BSD Sockets)
- WinSock (Windows Socket)
  - Microsoft 替 Windows 提供的一套 TCP/IP API(Application Programming Interface)
- Java Socket
  - Sun Microsystems 公司爲 Java 制定的較低階網路通訊 API

The Internet Application Architecture



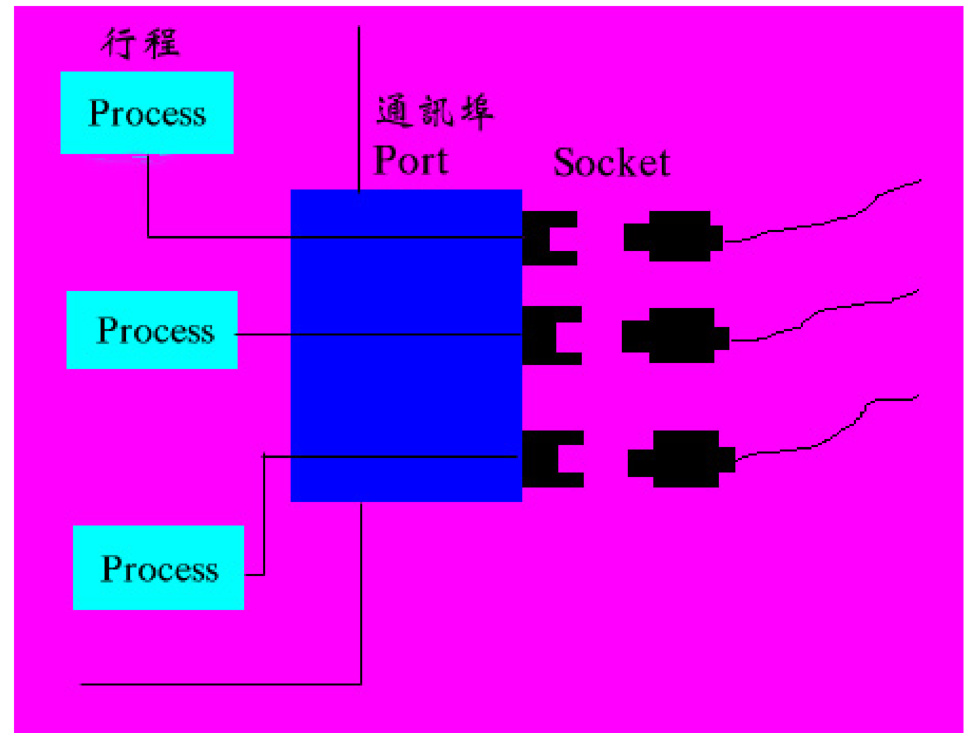
何謂Socket?

- 從網路的角度：

  通訊連結的端點

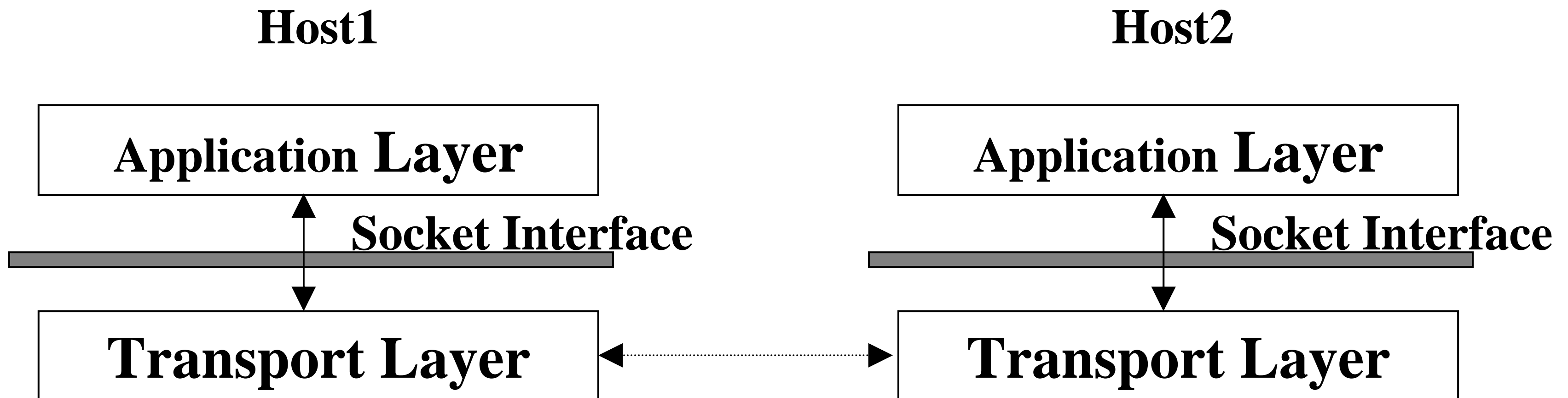
- 從程式設計者的角度：

  撰寫網路通訊程式的API



Socket (as an API)

- 讓網路應用程式可以取得Transport Layer的服務：



Socket的分類

- 在TCP/IP下，sockets可分爲下面兩類

  1. *Stream sockets* (connection-oriented)
  -- It provides reliable, connected networking service
  – Error free; no out- of- order packets (uses TCP)

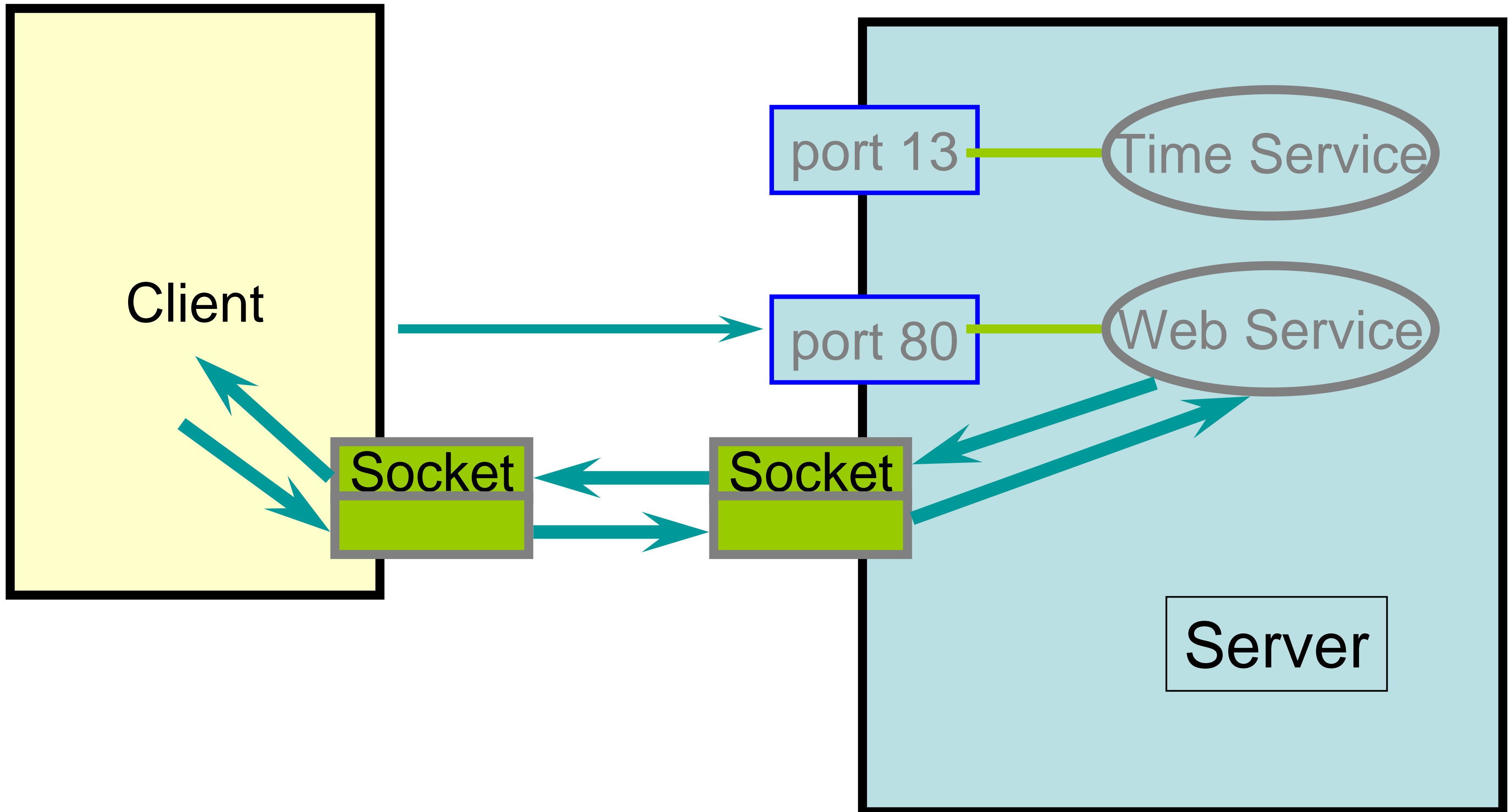
  2. *Datagram sockets* (connectionless)
  – It provides unreliable, best- effort networking service
  – Packets may be lost; may arrive out of order (uses UDP)

- 產生Socket必同時指定其種類

Sockets and Ports (Diagram)

IP + Port

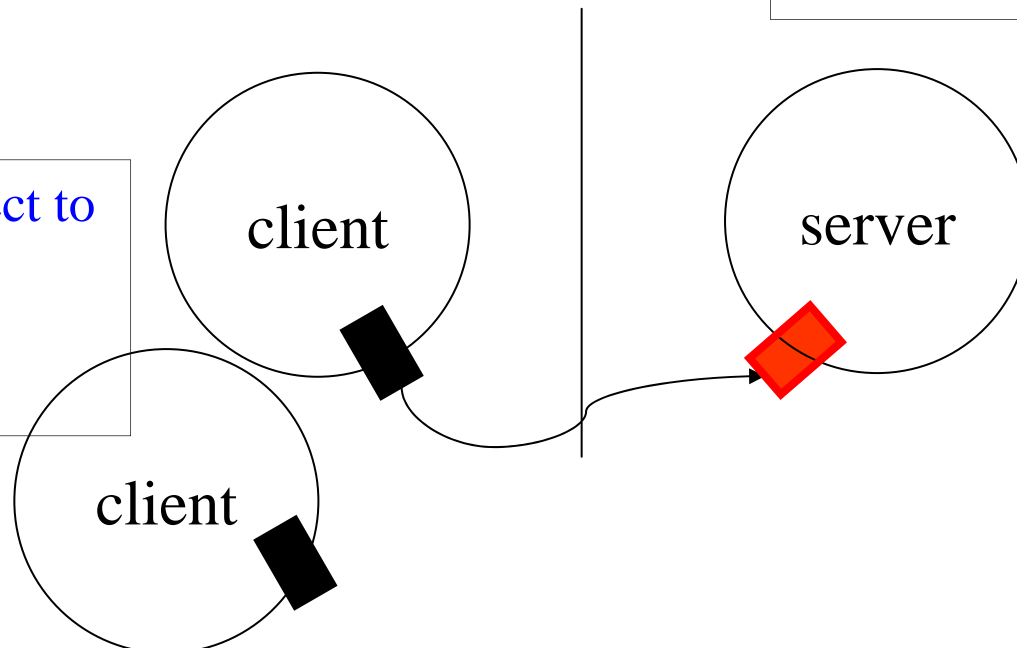
Client and Server (multiple services)



Working Model of Sockets

- Server: Waiting for client requests
- Client: Initiate the communication

(2) A "connection" is created, with the server getting a new socket. The "accepting socket" remains for future connections



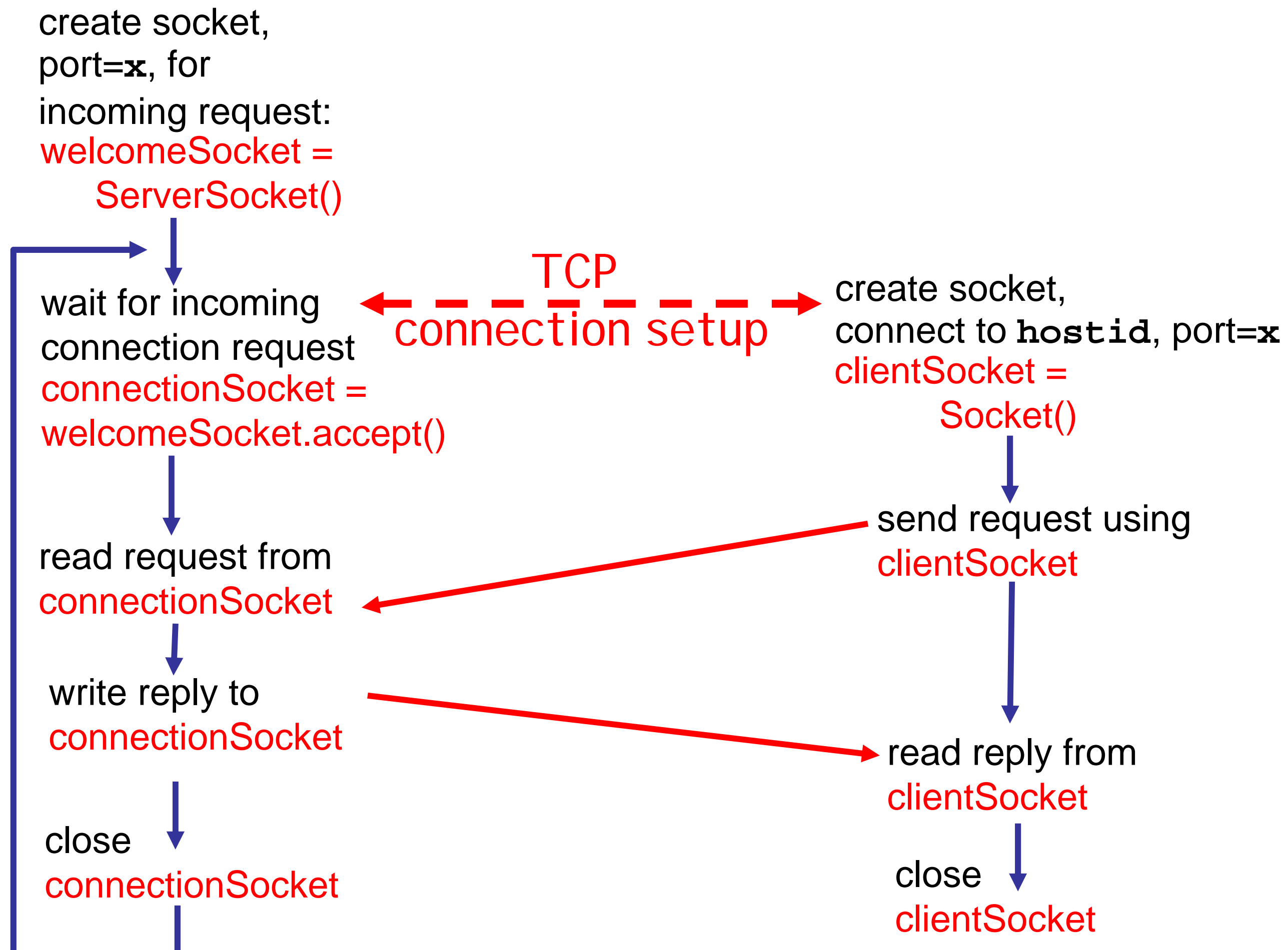
(1) Client connect to server. Server is "accepting" connections

Major Sockets API

Primitive	**Meaning**
Socket	Create a new communication endpoint
Bind	Attach a local address to a socket
Listen	Announce willingness to accept connections
Accept	Block caller until a connection request arrives
Connect	Actively attempt to establish a connection
Send	Send some data over the connection
Receive	Receive some data over the connection
Close	Release the connection

- Socket primitives for TCP/IP.

Client/server socket interaction: TCP



Pseudo code TCP server

Create socket (doorbellSocket)

Bind socket to a specific port where clients can contact you

Register with the kernel your willingness to listen that on socket for client to contact you

Loop

    Accept new connection (connectSocket)

    Read and Write Data Into connectSocket to Communicate with client

    Close connectSocket

End Loop

Close doorbellSocket

TCP Server vs. Client

- Server *waits* to accept connection on well known port
- Client initiates contact with the server
- Accept call returns a *new socket* for this client connection, freeing welcoming socket for other incoming connections
- Read and write only (addresses implied by the connection)

Concurrent TCP Servers

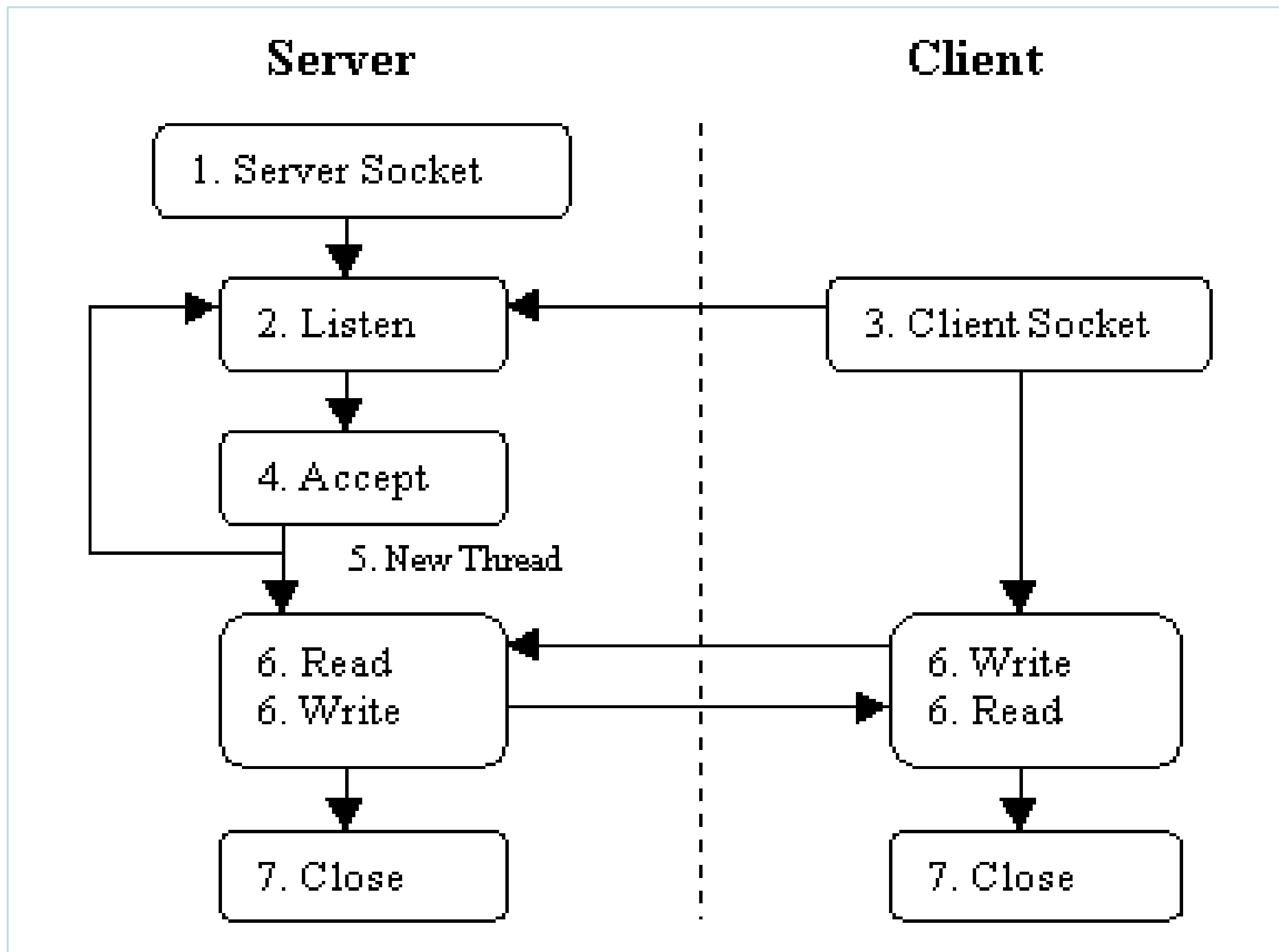
- What good is the doorbell socket? Can′t accept new connections until call accept again anyway?
- Benefit comes in ability to hand off processing to another process
  - Parent process creates the “door bell” or “welcome” socket on well-known port and waits for clients to request connection
  - When a client does connect, *fork off a child process* to *handle that connection* so that parent process can return to waiting for connections as soon as possible

- Multithreaded server: same idea, just spawn off another thread rather than a full process

Pseudo code concurrent TCP server

```
Create socket doorbellSocket
Bind
Listen
Loop
    Accept the connection, connectSocket
    Fork
    If I am the child
        Read/Write connectSocket
        Close connectSocket
        exit
EndLoop
Close doorbellSocket
```

Multithreaded Servers



Backlog and Others

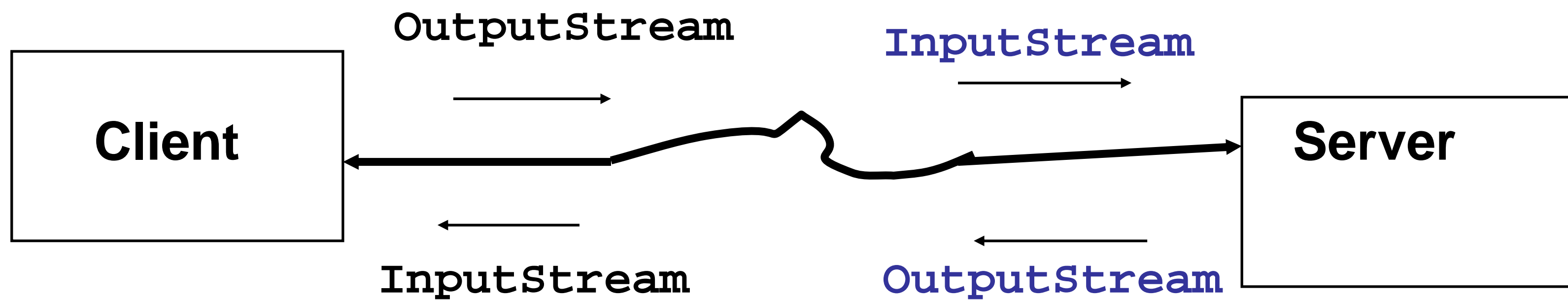
- Many implementations do allow a small fixed number (~5) of unaccepted connections to be pending, commonly called the backlog
- This helps avoid missing connections while process not sitting in the accept call
- Threads Pool
- Application Servers

A Quick Overview of Java Sockets

TCP Only

Java Socket Programming

- **java.net.Socket** is an abstraction of a bi-directional communication channel between hosts
- Send and receive data using I/O streams



- Socket types
  - Socket
  - Server Socket

The Socket Class

- `Socket(String host, int port)`
- `InputStream getInputStream()`
- `OutputStream getOutputStream()`
- `void close()`

Socket s = new Socket("www.cs.nccu.com", 80);

- Simple, huh? Client side only
- Creates a connection to the server
- Acquire I/O streams from the socket
- Server side:

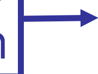
  ServerSocket welcomeSocket = new ServerSocket(6789);

Example: Java client (TCP)

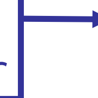
- Client: User keyboard input -> socket -> server

```
import java.io.*;
import java.net.*;
class TCPClient {

    public static void main(String argv[]) throws Exception
    {
        String sentence;
        String modifiedSentence;

        BufferedReader inFromUser =
          new BufferedReader(new InputStreamReader(System.in));

        Socket clientSocket = new Socket("hostname", 6789);

        DataOutputStream outToServer =
          new DataOutputStream( clientSocket.getOutputStream() );
```

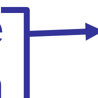
Create input stream



Create client socket, connect to server



Create output stream attached to socket



Example: Java client (TCP), cont.

Create input stream attached to socket

Send line to server

Read line from server

```
BufferedReader inFromServer =
  new BufferedReader(new
  InputStreamReader( clientSocket.getInputStream() ));

sentence = inFromUser.readLine();

outToServer.writeBytes(sentence + '\n');

modifiedSentence = inFromServer.readLine();

System.out.println("FROM SERVER: " + modifiedSentence);

clientSocket.close();

    }
}
```

Example: Java server (TCP)

Single threaded

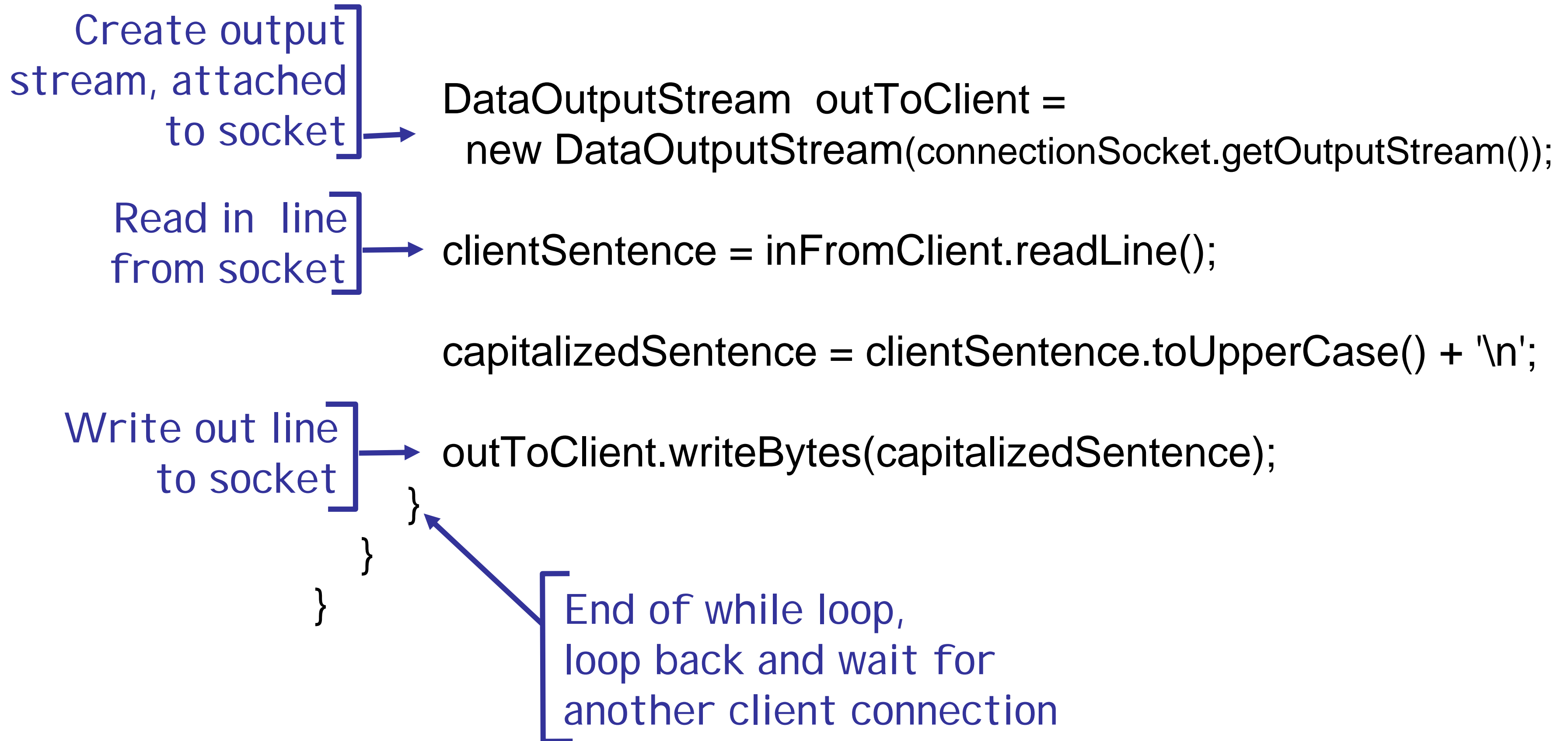
```
import java.io.*;
import java.net.*;

class TCPServer {

  public static void main(String argv[]) throws Exception
   {
     String clientSentence;
     String capitalizedSentence;

     ServerSocket welcomeSocket = new ServerSocket(6789);

     while(true) {

        Socket connectionSocket = welcomeSocket.accept();

       BufferedReader inFromClient =
         new BufferedReader(new
         InputStreamReader(connectionSocket.getInputStream()));
```

Create welcoming socket at port 6789 → `ServerSocket welcomeSocket = new ServerSocket(6789);`

Wait, on welcoming socket for contact by client → `Socket connectionSocket = welcomeSocket.accept();`

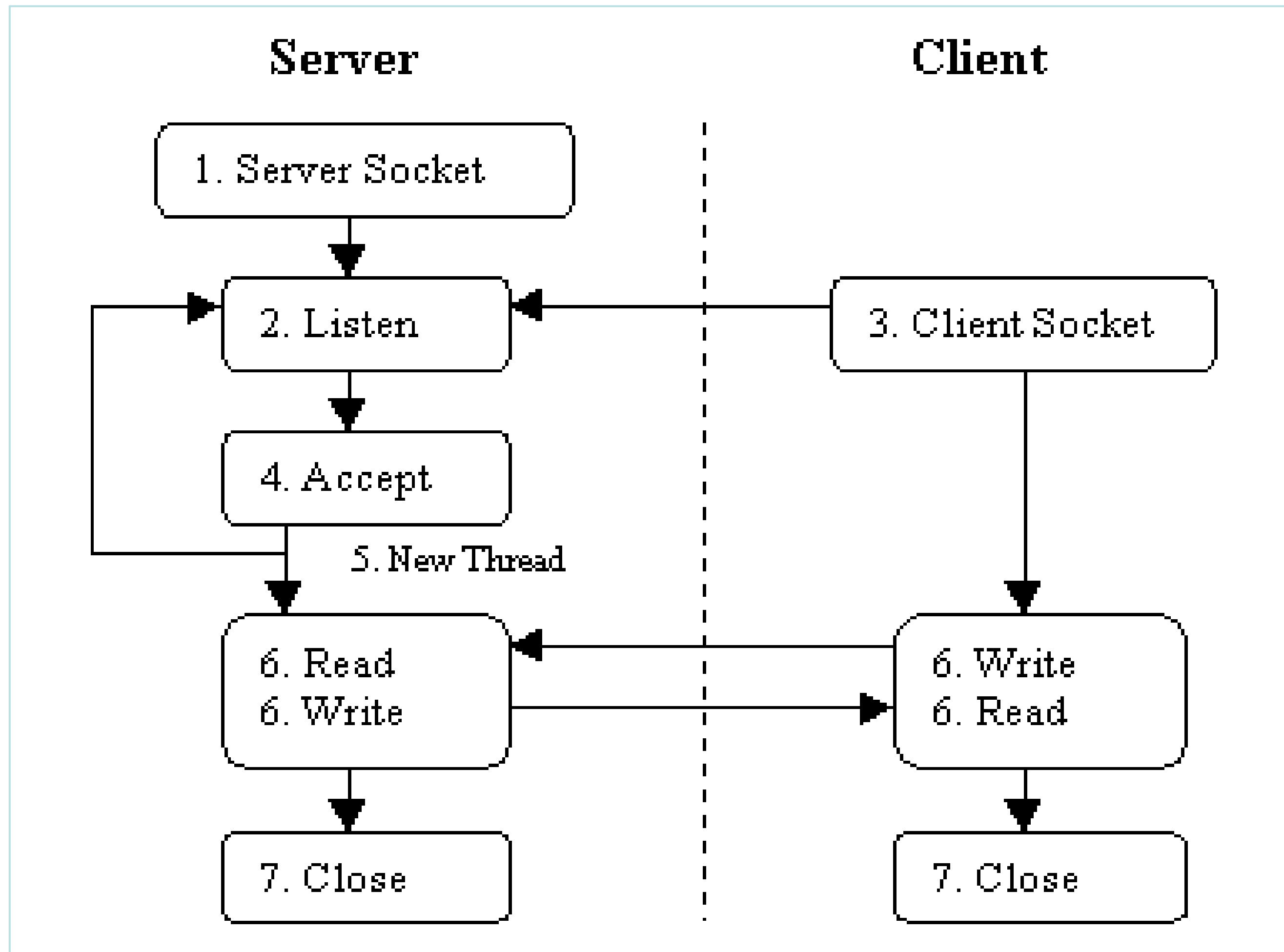
Create input stream, attached to socket → `BufferedReader inFromClient = ...`

Example: Java server (TCP), cont

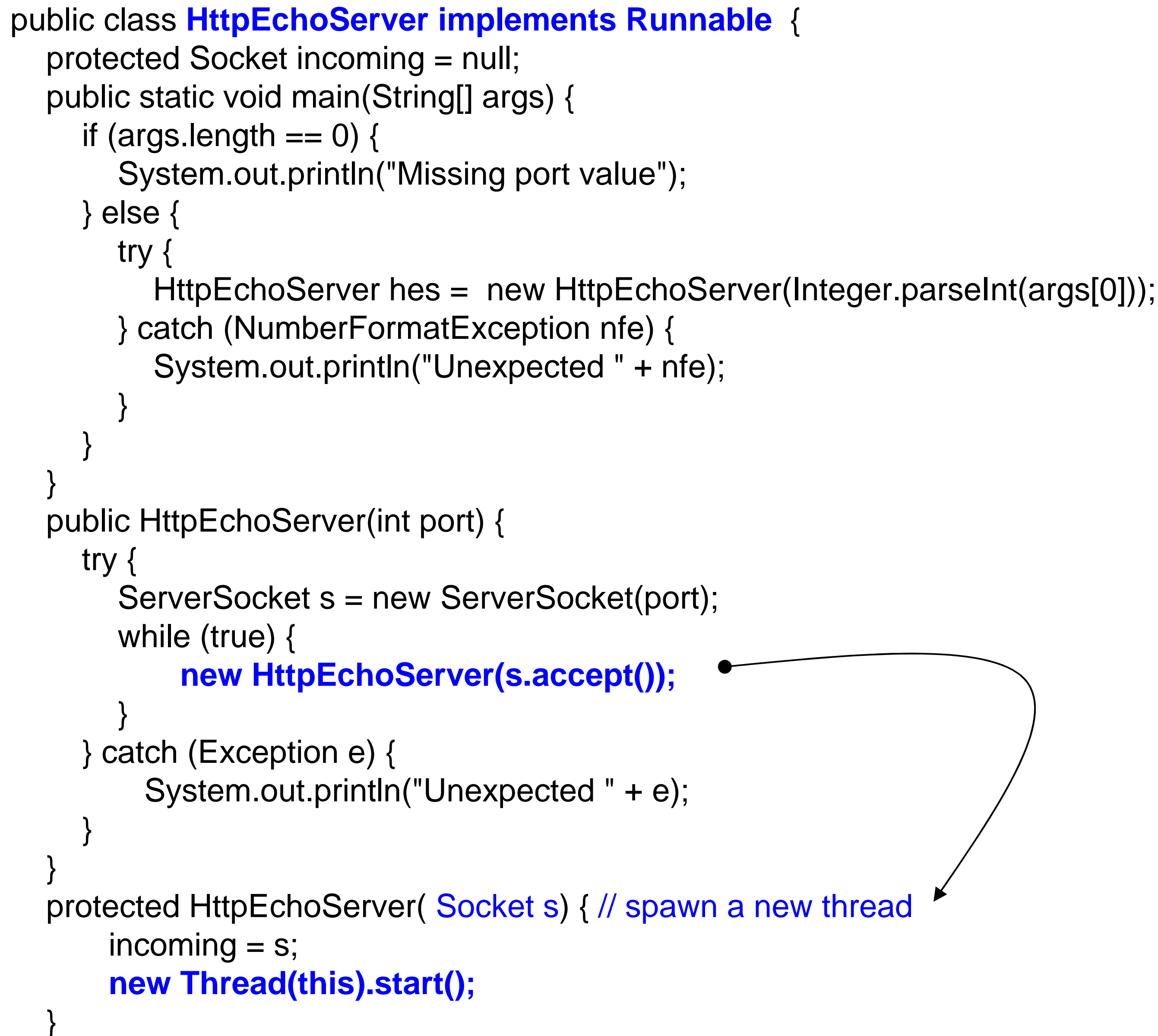


Multithreaded Servers

while (true) { Socket client = serverSocket.accept();
Create a new thread to handle request }



A Multithreaded Echo Server



```
public class HttpEchoServer implements Runnable {
    protected Socket incoming = null;
    public static void main(String[] args) {
        if (args.length == 0) {
            System.out.println("Missing port value");
        } else {
            try {
                HttpEchoServer hes =  new HttpEchoServer(Integer.parseInt(args[0]));
            } catch (NumberFormatException nfe) {
                System.out.println("Unexpected " + nfe);
            }
        }
    }
    public HttpEchoServer(int port) {
        try {
            ServerSocket s = new ServerSocket(port);
            while (true) {
                    new HttpEchoServer(s.accept());
            }
        } catch (Exception e) {
                System.out.println("Unexpected " + e);
        }
    }
    protected HttpEchoServer( Socket s) { // spawn a new thread
          incoming = s;
          new Thread(this).start();
    }
```

```
 public void run() {
      try {
         BufferedReader in = new BufferedReader(
               new InputStreamReader(incoming.getInputStream()));
         PrintWriter out = new PrintWriter(
               new OutputStreamWriter(incoming.getOutputStream()));
         out.println("HTTP/1.1 200 OK"); // http protocol header
         out.println("Content-Type: text/plain;\n");
         out.flush();
         while (true) {
            String str = in.readLine();
            if (str == null) {
               break; // client closed connection
            } else {
               out.println(str);
               out.flush();
               if (str.equals("")) // end of http request
                   break;
            }
         }
         incoming.close();
      } catch (Exception e) {
         System.out.println("Unexpected " + e);
      }
   }
}
```

A Simple Date Server

```
public class DateServer {
   private static Logger log = Logger.getLogger("dateLogger");

   public static void main (String args[]) throws IOException {
      ProgramProperties flags = new ProgramProperties( args);
      int port = flags.getInt( "port" , 8765);
      new DateServer().run(port);
   }

   public void run(int port) throws IOException {
      Thread.currentThread().setPriority(Thread.NORM_PRIORITY + 1);
      ServerSocket input = new ServerSocket( port );
      log.info("Server running on port " + input.getLocalPort());
      while (true) {
         Socket client = input.accept();
         log.info("Request from " + client.getInetAddress());
         // Create a new thread
         DateHandler clientHandler = new DateHandler( client.getInputStream(),
                                                      client.getOutputStream());
         clientHandler.setPriority( Thread.NORM_PRIORITY);
         clientHandler.start();
      }
   }
}
```

```
class DateHandler extends Thread {
  private InputStream in;
  private OutputStream out;

  public DateHandler(InputStream fromClient,
                     OutputStream toClient)
 {
    in = fromClient;
    out = toClient;
  }

  public void run() {
    try {
        processRequest();
    }
    catch (IOException error) {
      // Log and handle error
    }
  }
```

```
void processRequest() throws IOException {
      try {
        BufferedReader parsedInput =
            new BufferedReader(new InputStreamReader(in));

        boolean autoflushOn = true;
        PrintWriter parsedOutput =
                    new PrintWriter(out,autoflushOn);

        String inputLine = parsedInput.readLine();

        if ( inputLine.startsWith("date") ) {
           Date now = new Date();
           parsedOutput.println(now.toString());
        }
      }
      finally {
        in.close();
        out.close();
      }
   }
}
```

Thread Pools

When threads are good?

Let TC = time to create a thread

Let A = arrival time between two consecutive requests

We need TC << A

- Thread creation is not free.
- Used threads become garbage.
- Too many threads may crash the server.

- Use a thread pool of pre-forked threads to reduce the runtime overhead.

Concurrent Servers with Thread Pool

When usable:

TP = Time to process a request

A = arrival time between two consecutive requests

Then we need TP << A * N

```
Create N worker threads
while (true)
  {
    Socket client = serverSocket.accept();
    if worker thread is idle
       Use an existing worker thread to handle request
    else  // or wait if the number of worker thread is fixed
       create new worker thread to handle the request
  }
```

Approach

- **Issues**
  - Client requests are not constant over time
  - Requests can come in bursts
  - Threads consume resources
  - Don’t want a large pool of threads sitting idle

- **Common strategy**

  Have a minimum number of threads in a pool

  When needed add threads to the pool up to some maximum

  When traffic slows down remove idle threads

- Need Reusable threads! Separate threads and requets!

How to Reuse a thread?

- Classic idea
  - Server places client requests in a queue
  - Worker repeats forever

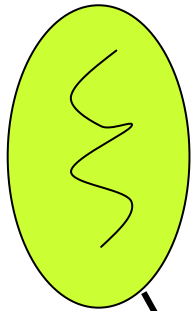
- Worker
  - Read request from queue
  - Process request

- Queue
  - Block on read if queue is empty
  - Signals waiting threads when data is added

Thread Pool: Basic Idea

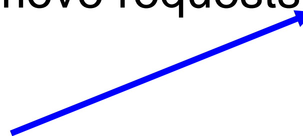
Decouples **request handling** **from** **thread creation**

Server thread

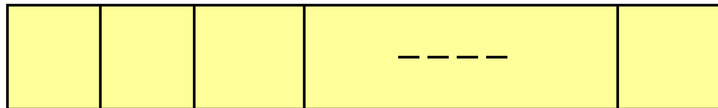


Add New requests

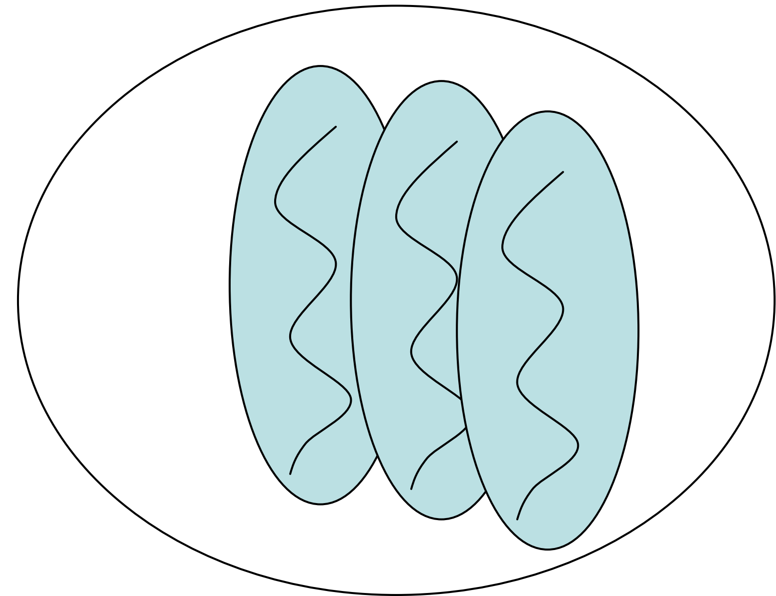
Remove requests



Thread Pool



Request Queue (sockets)



Worker Threads

Shared Queue

```
import java.util.ArrayList;
public class SharedQueue
 {
     ArrayList elements = new ArrayList();

     public synchronized void append( Object item )
     {
        elements.add( item); // Appends the specified element to the end of this list.
        notify();
     }
    public synchronized Object get( )
     {
       try  {
          while ( elements.isEmpty() )
             wait();
       } catch (InterruptedException threadIsDone )
       {      return null;      }
       return elements.remove( 0);  // Removes the element at the specified position in this list.
     }
    public int size()
    {
      return elements.size();
    }
 }
```

Date Server, 1

```
public class DateServer
{
  SharedQueue workQueue;
  ServerSocket listeningSocket;
  ArrayList workers = new ArrayList();  // thread pool, size limit?
  public static void main( String[] args ) {
      System.out.println( "Starting");
      new DateServer( 33333).run();
  }
  public DateServer( int port ) {
    try {
        listeningSocket = new ServerSocket(port);
        workQueue = new SharedQueue();
        // thread creation
        for (int k = 0; k < THREAD_NO; k++) // number of  threads
        {
           Thread worker = new DateHandler( workQueue);
           worker.start();
           workers.add( worker);
         }
    }  catch (IOException socketCreateError)
       {
          //log and exit here
       }
    }
```

Date Server, 2

```
public void run()
{
    Socket client = null;
    while (true)
    {
      try
        {
            client = listeningSocket.accept();  // new request
            workQueue.append( client );         // add request (socket)
        }
      catch (IOException acceptError)
        {
        // need to log error and make sure client is closed
        }
      }
    }
}
```

Worker Threads: DateHandler

```
public class DateHandler extends Thread
{
  SharedQueue workQueue;

  public DateHandler(SharedQueue workSource )
    {
        workQueue = workSource;
    }

  public void run()
  {
    while  (!isInterrupted() )
      try
        {
            Socket client = (Socket) workQueue.get();
            processRequest(client);
        }
      catch (Exception error )
        {
        /* log error*/
        }
  }
```

```
void processRequest(Socket client) throws IOException
{
    try
      {
        client.setSoTimeout( 10 * 1000 );
        processRequest( client.getInputStream(),   client.getOutputStream());
      }
    finally
    {     client.close();       }
}

  void processRequest(InputStream in, OutputStream out)   throws IOException
  {
    BufferedReader parsedInput =
          new BufferedReader(new InputStreamReader(in));
    PrintWriter parsedOutput = new PrintWriter(out,true);
    String inputLine = parsedInput.readLine();
    if (inputLine.startsWith("date"))
     {
        Date now = new Date();
        parsedOutput.println(now.toString());
     }
   }
}
```

Improvements

- Have a minimum number of threads in a pool
  - Min_Threads vs. Max_Threads
- When needed add threads to the pool up to some maximum
  - Dynamically create new threads
- When traffic slows down remove idle threads
  - Set a timeout value for idle threads
    - wait(TIME)

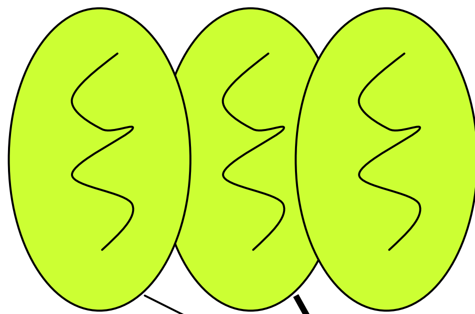
Worker Thread Pattern

Decouple method invocation from method execution

Worker Threads: Basic Idea

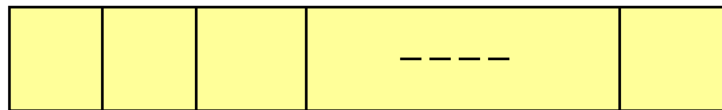
- ***Clients do not call the server for requests directly***
- ***Decouples*** **method invocation** **from** **method execution**

client threads



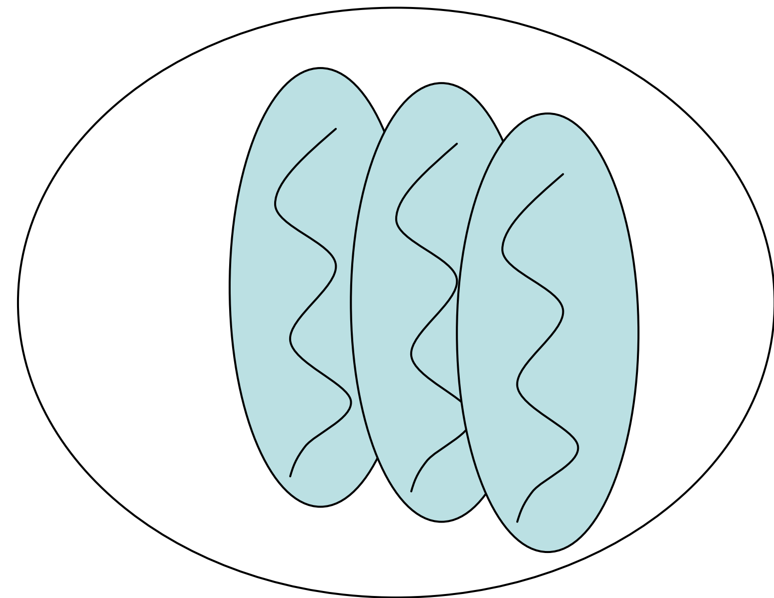
Add New requests

Remove requests



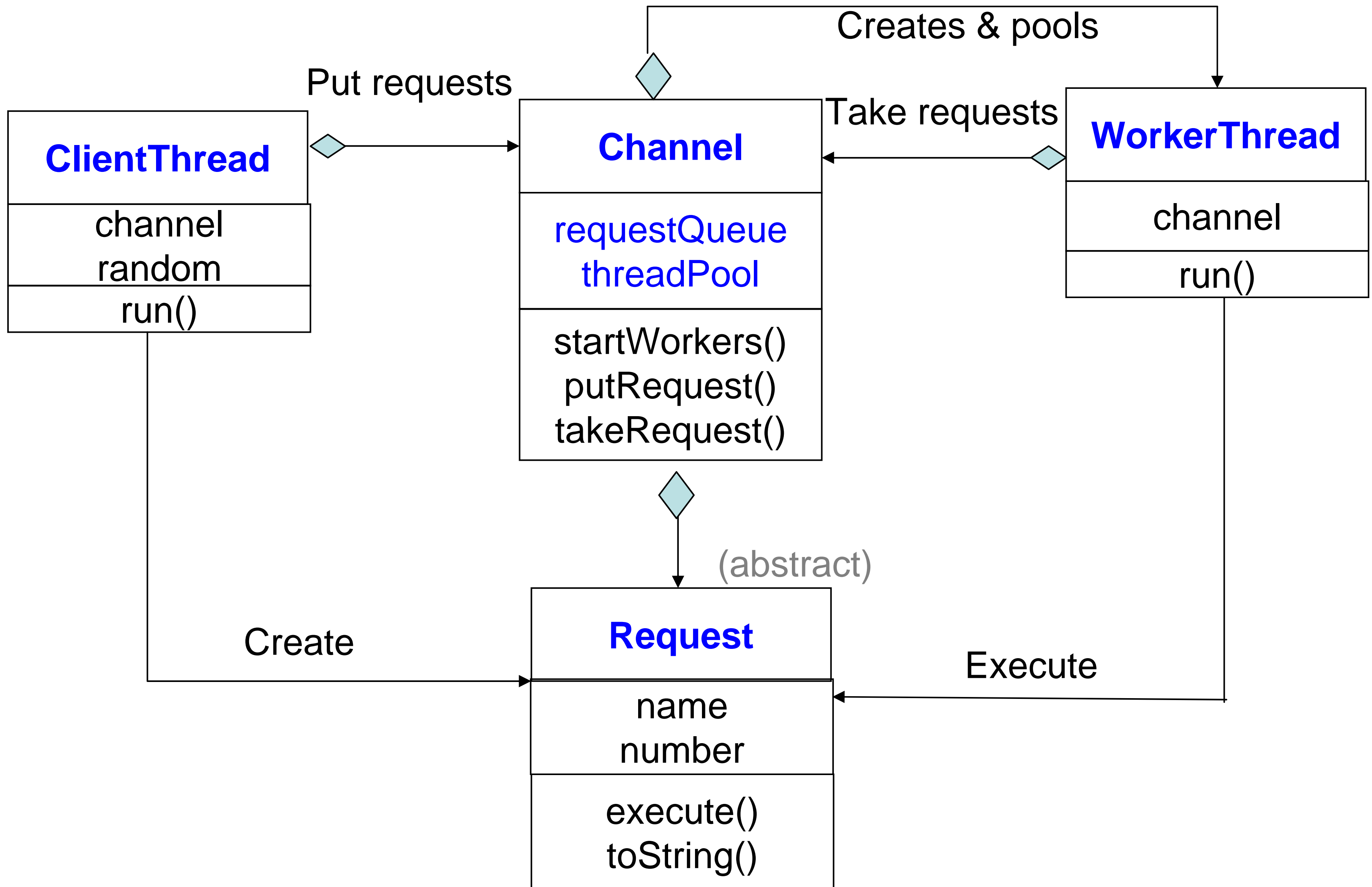
Request Queue

Thread Pool



Worker Threads

Participants & Class Diagram



Channel: Queue & Pool

```
public class Channel {
    private static final int MAX_REQUEST = 100;
    private final Request[] requestQueue;
    private int tail;  // 下一個putRequest的地方
    private int head;  // 下一個takeRequest的地方
    private int count; // Request的數量
    private final WorkerThread[] threadPool;

    public Channel (int threads) {
        this.requestQueue = new Request[MAX_REQUEST];
        this.head = 0;
        this.tail = 0;
        this.count = 0;
        // creating threads
        threadPool = new WorkerThread[threads];
        for (int i = 0; i < threadPool.length; i++) {
            threadPool[i] = new WorkerThread("Worker-" + i, this);
        }
    }
    public void startWorkers() {
        for (int i = 0; i < threadPool.length; i++) {
            threadPool[i].start();
        }
    }
```

```
public synchronized void putRequest(Request request) {
      while (count >= requestQueue.length) {
         try {
              wait();
         } catch (InterruptedException e) {
         }
      }
      requestQueue[tail] = request;
      tail = (tail + 1) % requestQueue.length;
      count++;
      notifyAll();
   }
   public synchronized Request takeRequest() {
      while (count <= 0) {
         try {
              wait();
         } catch (InterruptedException e) {
         }
      }
      Request request = requestQueue[head];
      head = (head + 1) % requestQueue.length;
      count--;
      notifyAll();
      return request;
   }
}
```

Client Threads

```
public class ClientThread extends Thread {
    private final Channel channel;
    private static final Random random = new Random();
    public ClientThread(String name, Channel channel) {
         super(name);
         this.channel = channel;
    }
    public void run() {
        try {
            for (int i = 0; true; i++) {
                Request request = new Request(getName(), i);
                channel.putRequest(request);
                Thread.sleep(random.nextInt(1000));
            }
        } catch (InterruptedException e) {
        }
    }
}
```

Request Class (Command Pattern)

```
public class Request { //could be abstract
    private final String name; // 委託者
    private final int number;  // 請求編號
    private static final Random random = new Random();
    public Request(String name, int number) {
        this.name = name;
        this.number = number;
    }
    public void execute() {  //could be abstract
        System.out.println(Thread.currentThread().getName() + " executes " + this);
        try {
            Thread.sleep(random.nextInt(1000));
        } catch (InterruptedException e) {
        }
    }
    public String toString() {
        return "[ Request from " + name + " No." + number + " ]";
    }
}
```

Worker Threads

```
public class WorkerThread extends Thread {
    private final Channel channel;
    public WorkerThread(String name, Channel channel) {
        super(name);
        this.channel = channel;
    }
    public void run() {
        while (true) {
            Request request = channel.takeRequest();
            request.execute();
        }
    }
}
```

Java 5: java.util.concurrent

Concurrency Utilities

- **Executors**
  - *Executor*
  - *ExecutorService*
  - *ScheduledExecutorService*
  - *Callable*
  - *Future*
  - *ScheduledFuture*
  - *Delayed*
  - *CompletionService*
  - ThreadPoolExecutor
  - ScheduledThreadPoolExecutor
  - AbstractExecutorService
  - Executors
  - FutureTask
  - ExecutorCompletionService
- Queues
  - *BlockingQueue*
  - ConcurrentLinkedQueue
  - LinkedBlockingQueue
  - ArrayBlockingQueue
  - SynchronousQueue
  - PriorityBlockingQueue
  - DelayQueue
- Concurrent Collections
  - *ConcurrentMap*
  - ConcurrentHashMap
  - CopyOnWriteArray{List,Set}
- Synchronizers
  - CountDownLatch
  - Semaphore
  - Exchanger
  - CyclicBarrier
- Timing
  - TimeUnit
- Locks
  - *Lock*
  - *Condition*
  - *ReadWriteLock*
  - AbstractQueuedSynchronizer
  - LockSupport
  - ReentrantLock
  - ReentrantReadWriteLock
- Atomics
  - Atomic[Type]
  - Atomic[Type]Array
  - Atomic[Type]FieldUpdater
  - Atomic{Markable,Stampable}Reference

Executor Framework

- The `java.util.concurrent` package contains a flexible thread pool implementation, but even more valuable, it contains an entire framework for managing the execution of tasks that implement Runnable. This framework is called the **Executor framework.**
- The Executor interface is quite simple. It describes an object whose job it is to run *Runnables*:

```
public interface Executor {
    void execute(Runnable command);
}
```

Old Thread Example

```
class Worker implements Runnable
{
     public void run()
     {
          System.out.println("New Worker");
      }
}

public class OrgThread
{
    public static void main(String args[])
{
     Thread t = new Thread(new Worker());
       t.start();
 }
}
```

Thread Executor Example

```
class Worker implements Runnable
{
    public void run()
    {
        System.out.println("New Worker");
     }
}

public class ConThread
{
     public static void main(String args[])
     {
          Executor e = Executors.newFixedThreadPool(5);
          e.execute(new Worker());
  }
}
```

Kinds of Thread Pools

- Executors.newCachedThreadPool() Creates a thread pool that is not limited in size, but which will reuse previously created threads when they are available. If no existing thread is available, a new thread will be created and added to the pool. Threads that have not been used for 60 seconds are terminated and removed from the cache.

- Executors.newFixedThreadPool(int n) Creates a thread pool that reuses a fixed set of threads operating off a shared unbounded queue. If any thread terminates due to a failure during execution prior to shutdown, a new one will take its place if needed to execute subsequent tasks.

- Executors.newSingleThreadExecutor() Creates an Executor that uses *a single worker thread* operating off an unbounded queue, much like the Swing event thread. Tasks are guaranteed to execute sequentially, and no more than one task will be active at any given time.

A Simple Web Server

```
class UnreliableWebServer {
    public static void main(String[] args) {
        ServerSocket socket = new ServerSocket(80);
        while (true) {
            final Socket connection = socket.accept();
            Runnable r = new Runnable() {
                            public void run() {
                                handleRequest(connection);
                            }
                        };
            // Don't do this!
            new Thread(r).start();
        }
    }
}
```

Rewrite the Web Server Using Executor

```
class ReliableWebServer {
    Executor pool = Executors.newFixedThreadPool(7);
    public static void main(String[] args) {
        ServerSocket socket = new ServerSocket(80);
        while (true) {
            final Socket connection = socket.accept();
            Runnable r = new Runnable() {
                            public void run() {
                                handleRequest(connection);
                            }
                        };
            pool.execute(r);
        }
    }
}
```