CS399J: *Programming with Java*

The Java™ Programming Platform was designed with the internet in mind. Java provides an object-oriented view of networking that allows data to be easily sent between computers. Additionally, Java provides straightforward, yet powerful, mechanisms for multi-threaded concurrent programming that are often used in conjunction with networking.

Java Networking

- Networking Essentials
- Object Serialization
- Concurrent Programming

Copyright ©2000-2006 by David M. Whitlock. *Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and full citation on the first page. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or fee. Request permission to publish from whitlock@cs.pdx.edu.*

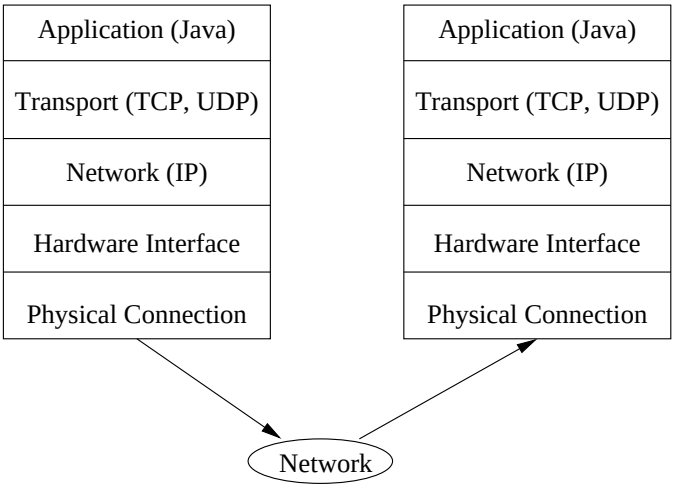
Networking With Java

Networking is ubiquitous throughout Java

- Applets are downloaded over the Internet
- Support for HTTP and URLs
- Classes that model data sent using the TCP and UDP protocols
- The `java.net` package

Network Layering

Networking can be conceptualized in terms of “layers”



Each layer only communicates with the layers directly above and below it.

Allows for modular network design

Transport Protocols: TCP and UDP

TCP (Transmission Control Protocol) provides reliable point-to-point connection-based communication

- Like making a phone call
- Used with applications such as FTP and telnet

UDP (User Datagram Protocol) provides non-guaranteed packet-based communication

- Order of packets is not important – packets are independent of each other
- Like sending a letter
- Less overhead than TCP
- Used for ping

Ports

A *port* is a conceptual hole in your computer through which data flows

An application communicates with the outside world via a port

Data that arrives from the network knows the which port on which computer, it is destined for

Ports are identified by a 16-bit number. The first 1024 ports are reserved for use by the operating system and certain protocols such as HTTP.

`java.net` package

The `java.net` package contains classes that allow programs to communicate over a network in a platform-independent way

`java.net.URL`

A Uniform Resource Locator (URL) provides an address and an obtaining protocol for a resource on the internet.

```
http://www.cs.pdx.edu/~whitlock
```

A URL is modeled by the `java.net.URL` class

- A `URL` can be created from a `String` or can be described by its protocol, host, port number, etc.
- `openStream` returns an `java.io.InputStream` that can be used to read the contents of a URL
- `openConnection` returns a `URLConnection` that can be use to interact with the URL (e.g. send CGI requests)

Recall that `java.io.File` has a method named `toURL` that returns the `URL` for a file.

`java.net.URL` example

```
package edu.pdx.cs399J.net;
import java.io.*;
import java.net.*;

public class DumpURL {
  public static void main(String[] args) {
    try {
      URL url = new URL(args[0]);
      InputStream urlStream = url.openStream();
      InputStreamReader isr =
        new InputStreamReader(urlStream);
      BufferedReader br = new BufferedReader(isr);
      while (br.ready()) {
        String line = br.readLine();
        System.out.println(line);
      }
      br.close();

    } catch (MalformedURLException ex) {
      System.err.println("** Bad URL: " + args[0]);
      System.exit(1);
    } catch (IOException ex) {
      System.err.println("** IOException: " + ex);
      System.exit(1);
    }
  }
}
```


`java.net.URL` example

```
$ java -cp ~/classes edu.---.DumpURL \
  http://www.cs.pdx.edu/~whitlock

<HTML>

<HEAD>
<TITLE>CS399J Homepage</TITLE>
</HEAD>

<BODY bgcolor="white" text="black">

<P align="center"><font size=+2>CS399J: Progamming Wi

<P align="center">
  Starring: Herr Professor David Whitlock<br>
  Wednesday nights 6:00-9:30 in PCAT 138
</P>

<P><B>Goal:</B> To learn how to use the Java<SUP>TM</
platform and to have fun doing it.</P>

<P><B>Textbook:</B> <A href="http://www.bruceeckel.co
Thinking in Java</a> by Bruce Eckel</P>

<P align="center"><A href="docs">CS399J docs</A> |
<A href="src">CS399J source</A></P>

...
```

Sockets

URLs are good for performing high-level internet-related networking

Sockets are used for more specific application-level networking

A socket is an endpoint of a TCP communication between a server and a client

A server program (process) listens on a given port

A client program attempts to connect to a given port on a given machine

If a connection is established between a client and a server, each receives a socket through which they can communicate

The `java.net.Socket` and `java.net.ServerSocket` classes are used to establish network connections between Java programs

`java.net.ServerSocket`

A `ServerSocket` is used by a server and waits for a client to request a connection

- A `ServerSocket` is created by giving the port it listens on (can also specify the number of backlog messages)
- `accept` waits for a client to establish a connection, the resulting `Socket` is returned
- `setSoTimeout` sets the number of milliseconds a `SocketServer` waits before timing out

`java.net.Socket`

A `Socket` is created by a client that wants to communicate with a server

- A `Socket` is created with a host name and port number
- `getInputStream` returns an `InputStream` for reading data from the socket
- `getOutputStream` returns an `OutputStream` for writing data to the socket

A sample server

```
package edu.pdx.cs399J.net;
import java.io.*;
import java.net.*;

/**
 * Reads strings from a socket until "bye" is
 * encountered.
 */
public class Listener {
  private static PrintStream out = System.out;
  private static PrintStream err = System.err;

  public static void main(String[] args) {
    int port = 0;

    try {
      port = Integer.parseInt(args[0]);
    } catch (NumberFormatException ex) {
      err.println("** Bad port number: " + args[0]);
      System.exit(1);
    }

    // continued...
```


A sample server

```
    try {
      // Backlog of 5 messages
      ServerSocket server =
        new ServerSocket(port, 5);

      // Wait for the Speaker to connect
      Socket socket = server.accept();

      // Read from the Socket
      InputStreamReader isr =
        new InputStreamReader(socket.getInputStream()
      BufferedReader listener =
        new BufferedReader(isr);

      String line = "";
      while (!line.equals("bye")) {
        line = listener.readLine();
        out.println(line);
      }

      listener.close();

    } catch (IOException ex) {
      err.println("** IOException: " + ex);
      System.exit(1);
    }
  }
}
```

A sample client

```
package edu.pdx.cs399J.net;
import java.io.*;
import java.net.*;

public class Speaker {
  public static void main(String[] args) {
    // <snip> Read the host and port number

    try {
      Socket socket = new Socket(host, port);
      OutputStreamWriter osw =
        new OutputStreamWriter(socket.getOutputStream
      PrintWriter speaker = new PrintWriter(osw);

      for (int i = 2; i < args.length; i++) {
        speaker.println(args[i]);
      }
      speaker.close();

    } catch (UnknownHostException ex) {
      System.err.println("** Could not connect " +
                         "to host: " + host);
    } catch (IOException ex) {
      System.err.println("** IOException: " + ex);
      System.exit(1);
    }
  }
}
```


Working with our samples

Start the server in one shell...

```
$ java -cp ~/classes edu.---.Listener 12345
```

Start the client in another shell...

```
$ java -cp ~/classes edu.---.Speaker \
  localhost 12345 Java networking is cool! bye
```

In the server shell...

```
Java
networking
is
cool!
bye
```


Other classes in the `java.net` package

Datagram classes for UDP connections: `DatagramSocket` and `DatagramPacket`

Support for multicast networking: `MulticastSocket`

Utility classes for working with URLs: `URLEncoder` and `URLDecoder`

Some security related classes: `NetPermission`, `SocketPermission`, `Authenticator`, and `PasswordAuthentication`

And, of course, a bunch of exceptions

Objects and bytes

Networking and sockets communicate at the byte level

However, Java works with objects

Object serialization is a process that converts objects into bytes

Serialized objects can be sent over the network, accessed from binary files, etc.

Any class that implements the `java.io.Serializable` interface may be serialized

- Contains no methods – serialization happens “under the covers”
- `static` and `transient` fields are not serialized
- For more control over serialization, implement `writeObject` and `readObject`

Note that the non-serializable superclass of a serializable class must have a zero-argument constructor

Some classes that are serializable include the “wrapper” classes, `Throwable`, `DateFormat`, `Date`, `Calendar`, and many of the collection classes

Objects and streams

`java.io.ObjectOutputStream` is used to write objects to an `OutputStream`

- Has method to write all of the primitive types and `Object`s to the underlying `OutputStream`

`java.io.ObjectInputStream` is used to read objects from an `InputStream`

- The constructor for `ObjectInputStream` blocks until an object can be read!
- Methods to read all of the primitive types and `Object`s from the underlying `InputStream`
- If the class of a serialized object cannot be found, then a `ClassNotFoundException` is thrown

Writing an object to an `ObjectOutputStream`

```
package edu.pdx.cs399J.net;
import java.io.*;
import java.util.*;

public class WriteDate {
  public static void main(String[] args) {
    String fileName = args[0];

    try {
      FileOutputStream fos =
        new FileOutputStream(fileName);
      ObjectOutputStream out =
        new ObjectOutputStream(fos);
      Date date = new Date();
      System.out.println("Writing " + date);
      out.writeObject(date);
      out.flush();
      out.close();

    } catch (IOException ex) {
      System.err.println("**IOException: " + ex);
      System.exit(1);
    }
  }
}
```


Reading from an `ObjectInputStream`

```
package edu.pdx.cs399J.net;
import java.io.*;
import java.text.*;
import java.util.*;

public class ReadDate {
  public static void main(String[] args) {
    String fileName = args[0];

    try {
      FileInputStream fis =
        new FileInputStream(fileName);
      ObjectInputStream in =
        new ObjectInputStream(fis);
      Date date = (Date) in.readObject();
      in.close();
      System.out.println("Read " + date);

    } catch (ClassNotFoundException ex) {
      System.err.println("** No class " + ex);
      System.exit(1);
    } catch (IOException ex) {
      System.err.println("**IOException: " + ex);
      System.exit(1);
    }
  }
}
```


Working with our example

```
$ java -cp ~/classes edu.---.WriteDate date.out
Writing Sat Oct 28 13:39:40 PDT 2000
```

The file `date.out` is binary

```
$ java -cp ~/classes edu.---.ReadDate date.out
Read Sat Oct 28 13:39:40 PDT 2000
```

Class versioning

To ensure version compatibility between classes, each class has a `long` serial version UID that is based on its fields and methods

If the class of the object being de-serialized does not have the same serial UID as the class in the VM that de-serializes it, then an exception will be thrown.

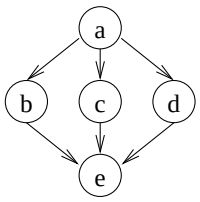
A class's serial UID may be explicitly set in its `static serialVersionUID` field

The `serialver` tool will tell you a class's serial UID

```
$ serialver java.util.ArrayList
java.util.ArrayList:
  static final long serialVersionUID =
  8683452581122892189L;
```


A more complex serialization example

To demonstrate Java serialization's support for *referential integrity* we will serialize an object graph made up of `Node` objects



When we serialize node `a`, all of its descendents will get serialized, too.

The `Node` class

```
package edu.pdx.cs399J.net;
import java.util.*;

public class Node implements java.io.Serializable {
  private Collection children = new ArrayList();
  private transient boolean beenVisited = false;

  public void addChild(Node child) {
    this.children.add(child);
  }

  /**
   * Returns this node's number of unvisited
   * descendents
   */
  public int traverse() {
    int total = 1;
    this.beenVisited = true;
    Iterator iter = children.iterator();
    while (iter.hasNext()) {
      Node child = (Node) iter.next();
      if (!child.beenVisited) {
        total += child.traverse();
      }
    }
    return total;
  }
}
```


Writing an object graph

```
package edu.pdx.cs399J.net;
import java.io.*;

public class WriteNodes {
  public static void main(String[] args) {
    String fileName = args[0];
    Node a = new Node(); Node b = new Node();
    Node c = new Node(); Node d = new Node();
    Node e = new Node();
    a.addChild(b); a.addChild(c);
    a.addChild(d); b.addChild(e);
    c.addChild(e); d.addChild(e);
    System.out.println("Graph has " + a.traverse()
                       + " nodes");
    try {
      FileOutputStream fos =
        new FileOutputStream(fileName);
      ObjectOutputStream out =
        new ObjectOutputStream(fos);
      out.writeObject(a);
      out.flush();
      out.close();
    } catch (IOException ex) {
      System.err.println("** IOException: " + ex);
      System.exit(1);
    }
  }
}
```

Reading an object graph

```
package edu.pdx.cs399J.net;
import java.io.*;

public class ReadNodes {
  public static void main(String[] args) {
    String fileName = args[0];

    try {
      FileInputStream fis =
        new FileInputStream(fileName);
      ObjectInputStream in =
        new ObjectInputStream(fis);
      Node root = (Node) in.readObject();
      System.out.println("Graph has " +
                  root.traverse() + " nodes");

    } catch (ClassNotFoundException ex) {
      System.err.println("** No class: " + ex);
      System.exit(1);

    } catch (IOException ex) {
      System.err.println("** IOException: " + ex);
      System.exit(1);
    }
  }
}
```


Working with the object graph example

```
java -cp ~/classes edu.---.WriteNodes nodes.out
Graph has 5 nodes
```

File `nodes.out` is binary

```
$ java -cp ~/classes edu.---.ReadNodes nodes.out
Graph has 5 nodes
```

As expected, the same number of nodes were serialized and de-serialized

Java serialization maintains referential integrity: you refer to the same object both before and after serialization

Why was it important that `Node`'s `beenVisited` field be declared `transient`?

Concurrent Programming

As programs become more complex, they tend to perform actions that are independent of each other.

As a result, it becomes desirable to have multiple threads of execution that run concurrently.

The Java programming language provides several built-in mechanisms for handling concurrent operations

These mechanisms include

- `java.lang.Thread` class
- `synchronized` methods and code blocks
- `wait`/`notify` methods

`java.lang.Thread`

`Thread` represents a thread of execution within a Java Virtual Machine

- `run`: Contains the code that is executed by a `Thread`
- `start`: Begins execution of a `Thread`, calls the `run` method (like `fork` in UNIX)
- `isAlive`: Determines whether or not a `Thread` is running (may be blocked)
- `setPriority`: Sets the "priority" of a `Thread` (varies by implementation, not very effective)
- `setDaemon`: Sets whether a thread is a user or daemon thread. The JVM exits when only daemon threads are running.
- `join`: Blocks the calling thread until the target `Thread` dies ("wait for the other thread to die before continuing")
- `interrupt`: Interrupts a `Thread`. If the thread is waiting, it will receive an `InterruptedException`.

java.lang.Thread

Thread has several interesting static methods

- currentThread: Returns the Thread that is currently executing
- sleep: Causes the currently executing Thread to wait for a given amount of time (in milliseconds)
- yield: Pauses the currently executing Thread and lets other threads run (just a hint)
- holdsLock(Object o): Returns true if the currently executing Thread holds the "lock" on a given Object
  - Especially useful with assertions:
    `assert Thread.holdsLock(lock);`

java.lang.Thread example

Big Bird and Mr. Snuffleupagus decide to have a race to see who can count to six first.

```
package edu.pdx.cs399J.net;
public class Counter extends Thread {
  private String name;

  public Counter(String name) {
    this.name = name;
  }

  public void run() {
    // Wait for a random amount of time and
    // then print a number
    for (int i = 1; i <= 6; i++) {
      try {
        long time = (long) (Math.random() * 1000);

        Thread.sleep(time);

      } catch (InterruptedException ex) {
        return;
      }

      System.out.println(this.name + ": " + i);
    }
  }
}
```


hava.lang.Thread example

```
package edu.pdx.cs399J.net;

public class CountingRace {
  public static void main(String[] args) {
    Counter bigBird = new Counter("BigBird");
    Counter snuffy = new Counter("Snuffy");

    bigBird.start();
    snuffy.start();
  }
}
```

```
$ java -cp ~/classes edu.---.CountingRace
BigBird: 1
Snuffy: 1
Snuffy: 2
Snuffy: 3
BigBird: 2
BigBird: 3
Snuffy: 4
BigBird: 4
Snuffy: 5
Snuffy: 6
BigBird: 5
BigBird: 6
```


Group Threads Together

A Thread may belong to a ThreadGroup

- Each ThreadGroup knows information about its Threads such as the number that are active
- Some operations (such as interrupt) can be performed on all of the threads in the group
- ThreadGroups may offer some security
  - You must be a member of thread group in order to access the threads in the group
- A ThreadGroup may be nested inside another ThreadGroup (its "parent")
- A ThreadGroup's exceptionOccurred method is invoked when an uncaught exception occurs in a thread
  - Often overridden to log the exception

Thread Coordination

Very often, you have a bunch of threads doing something, and you want them to stop

- Each thread has an *interrupt status* that denotes whether or not it has been interrupted
- `Thread`'s `interrupt` method sets the interrupt status
- The interrupt status can be queried with `isInterrupted` (instance method) or `Thread.interrupted` (static method that applies to current thread – clears interrupt status)
- If a thread is interrupted while it is sleeping, joining, or waiting, then an `InterruptedException` is thrown
  - Note that the interrupt status is **not** set

The following example uses interrupts to tell a bunch of working threads to stop

Thread Coordination Example

```
package edu.pdx.cs399J.net;
import java.util.Random;

public class WorkingThread extends Thread {

  public void run() {
    Random random = new Random();
    while (true) {
      if (this.isInterrupted()) {
        System.out.println(this + " is done");
        return;
      }

      int work = Math.abs(random.nextInt(100000));
      System.out.println(this + " working for " +
                         work);
      for (int l = 0; l < work; l++);

      try {
        int sleep = random.nextInt(2000);
        System.out.println(this + " sleeping for " +
                           sleep + " ms");
        Thread.sleep(sleep);

      } catch (InterruptedException ex) {
        System.out.println(this +
                  " interrupted while sleeping");
        return; } } } }
```


Thread Coordination Example

```
package edu.pdx.cs399J.net;

public class InterruptingThread extends Thread {
  private ThreadGroup group;
  private int sleep;

  public void run() {
    System.out.println(this + " sleeping for " +
                       this.sleep + " ms");
    try {
      Thread.sleep(this.sleep);

    } catch (InterruptedException ex) {
      System.err.println("WHY?");
      System.exit(1);
    }

    System.out.println(this +
                       " interrupting workers");
    this.group.interrupt();
  }
```


Thread Coordination Example

```
  public static void main(String[] args) {
    int sleep = Integer.parseInt(args[0]) * 1000;

    ThreadGroup group =
      new ThreadGroup("Worker threads");
    for (int i = 0; i < 5; i++) {
      Thread thread =
        new WorkingThread(group, "Worker " + i);
      thread.start();
    }

    InterruptingThread interrupting =
      new InterruptingThread("interrupter");
    interrupting.group = group;
    interrupting.sleep = sleep;
    interrupting.start();
  }
}
```

In your thread's "work loop" you should always check to see whether or it has been interrupted before doing a lot of work.

java.lang.Runnable

Extending `Thread` is rarely the right thing to do

Single inheritance only allows one superclass, what if your class needs other behavior in addition to thread semantics?

Interface `Runnable` has a `run` method

`Thread`s can be built around `Runnable` objects

Using `Runnable` is highly recommended

Concurrent Access to Data

When there are multiple threads operating on the same data, we have to ensure that the data is accessed in a consistent state

Classic example: A bank account

- If two transactions operate on a bank account at the same time, the balance may not be correct

```
package edu.pdx.cs399J.net;

public class BankAccount {
  private int balance;

  public int getBalance() {
    try {
      long time = (long) (Math.random() * 1000);
      Thread.sleep(time);
    } catch (InterruptedException ex) { }
    return this.balance;
  }
  public void setBalance(int balance) {
    try {
      long time = (long) (Math.random() * 1000);
      Thread.sleep(time);
    } catch (InterruptedException ex) { }
    this.balance = balance;
  }
}
```


A Naive ATM Machine

```
package edu.pdx.cs399J.net;
import java.io.PrintStream;

public class ATM implements Runnable {
  protected static PrintStream out = System.out;

  protected String name;
  protected BankAccount account;
  protected int[] transactions;

  public ATM(String name, BankAccount account,
             int[] transactions) {
    this.name = name;
    this.account = account;
    this.transactions = transactions;
  }

  public void run() {
    for (int i = 0; i < transactions.length; i++) {
      int balance = account.getBalance();
      balance += transactions[i];
      account.setBalance(balance);
    }
  }

  // continued...
```


A Naive ATM Machine

```
  public static void main(String[] args) {
    BankAccount account = new BankAccount();
    account.setBalance(1000);
    out.println("Initial balance: " +
                account.getBalance());

    int[] trans1 = {-200, 400, 100, -300};
    ATM atm1 = new ATM("ATM1", account, trans1);
    int[] trans2 = {400, 100, -300, -200};
    ATM atm2 = new ATM("ATM2", account, trans2);
    int[] trans3 = {-300, -200, 100, 400};
    ATM atm3 = new ATM("ATM3", account, trans3);

    Thread t1 = new Thread(atm1); t1.start();
    Thread t2 = new Thread(atm2); t2.start();
    Thread t3 = new Thread(atm3); t3.start();

    // Wait for all threads to finish
    try {
      t1.join(); t2.join(); t3.join();
    } catch (InterruptedException ex) {
      return;
    }
    out.println("Final balance: " +
                account.getBalance());
  }
}
```

A Naive ATM Machine

```
$ java -cp ~/classes edu.pdx.cs399J.net.ATM
Initial balance: 1000
ATM2 got balance 1000
ATM2 perform 400
ATM2 set balance to 1400
ATM1 got balance 1000     # Wrong!
ATM1 perform -200
ATM1 set balance to 800
ATM3 got balance 1000     # Wrong!
ATM3 perform -300
ATM3 set balance to 700
ATM3 got balance 800      # Wrong!
...
Final balance: 1400
```

Each ATM performs a net change of zero, but because they saw an inaccurate view of the balance, there was a net change.

How do we ensure that the ATMs always see the correct view of the balance?

Synchronized Access to Data

Every Java object has a monitor associated with it

A monitor is a thread synchronization device that can be thought of as a “lock”

Only one thread may hold the lock at a time. Other threads that want the lock must wait in line

In Java, a lock is obtained with the `synchronized` statement

```
Object lock;
...
// Wait to obtain the lock
synchronized(lock) {
  // I am the only thread that will run this code
  // This is a ‘‘critical section’’
  ...
}
// Give up the lock
...
```

We will modify our ATM example so that it obtains a lock on the account before performing the transaction.

A `synchronized` ATM Machine

```
  public void run() {
    for (int i = 0; i < transactions.length; i++) {
      // Get the lock on the account
      synchronized(account) {
        int balance = account.getBalance();
        balance += transactions[i];
        account.setBalance(balance);
      }
      // Give up the lock
    }
  }
```

```
$ java -cp ~/classes edu.---.SynchronizedATM
Initial balance: 1000
ATM1 got balance 1000
ATM1 perform -200
ATM1 set balance to 800
ATM2 got balance 800       # Right!
ATM2 perform 400
ATM2 set balance to 1200
ATM3 got balance 1200      # Right!
ATM3 perform -300
ATM3 set balance to 900
ATM1 got balance 900       # Right
...
Final balance: 1000
```

Note that the synchronized code runs slower

`synchronized` methods

Methods can be declared `synchronized`

```
public synchronized void doTransaction(int i) {
  ...
}
```

In `synchronized` methods, the lock on the `this` object is obtained before executing the code

```
public void doTransaction(int i) {
  synchronized(this) {
    ...
  }
}
```

`synchronized` methods let the object instead of the caller worry about synchronization.

`Vector` is synchronized, but the JDK1.2 collection classes are not

`Collections.synchronizedCollection` returns a collection whose methods are synchronized

A `synchronized` Bank Account

```
package edu.pdx.cs399J.net;

/**
 * Synchronized methods ensure that the data in
 * the balance is accessed correctly.
 */
public class SynchronizedBankAccount
    extends BankAccount {
  private static int nextId = 1;
  private int id = nextId++;
  private int balance;

  public synchronized int getBalance() {
    return super.getBalance();
  }

  public synchronized void setBalance(int balance) {
    super.setBalance(balance);
  }

  public synchronized void doTransaction(int trans) {
    // Will not attempt to re-obtain lock
    int balance = this.getBalance();
    balance += trans;
    this.setBalance(balance);
  }
}
```


Transferring between two accounts

```
package edu.pdx.cs399J.net;

public class Transfer implements Runnable {
  private BankAccount src;
  private BankAccount dest;
  private int amount;

  public Transfer(BankAccount src,
                  BankAccount dest, int amount) {
    this.src = src;
    this.dest = dest;
    this.amount = amount;
  }

  public void run() {
    System.out.println("Transferring " + this.amount)

    // Have to obtain locks on both accounts
    synchronized(this.src) {
      int srcBalance = src.getBalance();

      synchronized(this.dest) {
        int destBalance = dest.getBalance();
        src.setBalance(srcBalance - this.amount);
        dest.setBalance(destBalance + this.amount);
      }
    }
  }
```


Transferring between two accounts

```
  /**
   * Creates and performs a <code>Transfer</code>
   */
  public static void main(String[] args) {
    BankAccount acc1 = new BankAccount();
    acc1.setBalance(1000);
    BankAccount acc2 = new BankAccount();
    acc2.setBalance(500);

    (new Thread(new Transfer(acc1, acc2, 300))).start
    (new Thread(new Transfer(acc2, acc1, 100))).start
  }
}
```

```
$ java -cp ~/classes edu.---.Transfer
Transferring 300
Transferring 100
...
```

What happened?

Each thread held a lock that the other was waiting on – Deadlock!

Avoiding Deadlock

When threads access shared data, the program designer must be very careful to ensure that deadlock does not occur

Obtain a monolithic lock before accessing any shared data

- `static Object lock` object in `Transfer` class
- Slow and may unnecessarily prevent other threads from running

Order the shared data and always obtain the locks in order

- Obtain the lock on the account with the lowest id first, regardless of whether it is the source or destination
- Will all of the threads always finish?

The Producer/Consumer Problem

We often have a situation where one thread (the producer) is making something that some other thread (the consumer) wants.

How do we coordinate activity between the two?

The easiest way to do this is Java is to use `wait` and `notify`

Every `Object` has three thread-related methods

- `wait`: Have the current thread wait until the target object is notified by another thread
- `notify`: Let one of the threads that is waiting on this object know that it can continue executing
- `notifyAll`: Notify all threads waiting on the target object

Note that the thread must obtain the lock on an object before invoking its `wait` or `notify` method

`wait` releases the lock before waiting

You want fries with that?

As an example, we are going to model a McDonalds that has a bunch of customers who each want a BigMac™ and a number of employees who cook the BigMacs.

```
public class McDonalds {
  private static java.io.PrintStream err =
    System.err;
  private int nBigMacs;

  public McDonalds(int nBigMacs) {
    this.nBigMacs = nBigMacs;
  }

  public synchronized boolean moreBigMacs() {
    if (this.nBigMacs <= 0) {
      return false;
    } else {
      this.nBigMacs--;
      return true;
    }
  }

  // continued...
```


Can I super-size that, ma'am?

```
  public static void main(String[] args) {
    int nCustomers = 0;
    int nEmployees = 0;

    try {
      nCustomers = Integer.parseInt(args[0]);
      nEmployees = Integer.parseInt(args[1]);

    } catch (NumberFormatException ex) {
      err.println("** NumberFormatException");
      System.exit(1);
    }

    McDonalds mcDonalds = new McDonalds(nCustomers);

    for (int i = 0; i < nCustomers; i++) {
      McCustomer customer =
        new McCustomer(i, mcDonalds);
      (new Thread(customer)).start();
    }

    for (int i = 0; i < nEmployees; i++) {
      McEmployee employee =
        new McEmployee(i, mcDonalds);
      (new Thread(employee)).start();
    }
  }
}
```


One order of death-sticks to go!

```
package edu.pdx.cs399J.net;

public class McCustomer implements Runnable {
  private String name;
  private McDonalds mcDonalds;

  public McCustomer(int id, McDonalds mcDonalds) {
    this.name = "Customer " + id;
    this.mcDonalds = mcDonalds;
  }

  public void run() {
    System.out.println(this.name +
                       " wants a BigMac");

    try {
      synchronized(this.mcDonalds) {
        this.mcDonalds.wait();
      }

    } catch (InterruptedException ex) {
      return;
    }

    System.out.println(this.name + " got a BigMac");
  }
}
```

Can I take your order, please?

```
package edu.pdx.cs399J.net;
public class McEmployee implements Runnable {
  private String name;
  private McDonalds mcDonalds;

  public McEmployee(int id, McDonalds mcDonalds) {
    this.name = "Employee " + id;
    this.mcDonalds = mcDonalds;
  }

  public void run() {
    java.io.PrintStream out = System.out;
    out.println(this.name + " arrives at work");
    while (this.mcDonalds.moreBigMacs()) {
      out.println(this.name + " starts a BigMac");
      long wait  = (long) (Math.random() * 10000);
      try {
        Thread.sleep(wait);
      } catch (InterruptedException ex) {
        return;
      }
      out.println(this.name + " finishes a BigMac");
      synchronized(this.mcDonalds) {
        this.mcDonalds.notify();
      }
    }
  }
}
```


Hey, you got to be “Grimace” last week!

```
$ java -cp ~/classes edu.---.McDonalds 4 2
Customer 0 wants a BigMac
Customer 1 wants a BigMac
Customer 2 wants a BigMac
Customer 3 wants a BigMac
Employee 0 arrives at work
Employee 0 starts a BigMac
Employee 1 arrives at work
Employee 1 starts a BigMac
Employee 0 finishes a BigMac
Customer 0 got a BigMac
Employee 0 starts a BigMac
Employee 1 finishes a BigMac
Customer 1 got a BigMac
Employee 1 starts a BigMac
Employee 0 finishes a BigMac
Customer 2 got a BigMac
Employee 1 finishes a BigMac
Customer 3 got a BigMac
```


Putting it all together

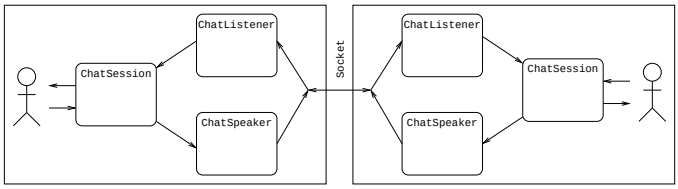
So far, we’ve learned about networking, object serialization, and writing multi-threaded applications.

We are going to demonstrate these three concepts with a rudimentary chat program that allows two people to talk with one another

One user starts up the chat program on a given port and waits for the other to connect via a socket

Each user types a message that is sent to the other over the network as a serialized object

Mutiple threads are used to listen for incoming messages and to send outgoing messages



`ChatMessage`s are sent back and forth

```
package edu.pdx.cs399J.net;
import java.io.Serializable;
import java.text.*;
import java.util.*;

public class ChatMessage implements Serializable {
  private String sender;
  private Date   date;
  private String text;

  public ChatMessage(String sender, String text) {
    this.sender = sender;
    this.date = new Date();
    this.text = text;
  }
  public boolean isLastMessage() {
    return this.text.trim().equals("bye");
  }
  public String toString() {
    DateFormat df =
      DateFormat.getTimeInstance(DateFormat.MEDIUM);
    StringBuffer sb = new StringBuffer();
    sb.append(this.sender); sb.append(" [");
    sb.append(df.format(this.date));
    sb.append("]> "); sb.append(this.text);
    return sb.toString();
  }
}
```

A `ChatSpeaker` sends `ChatMessages`

```
package edu.pdx.cs399J.net;
import java.io.*;
import java.util.*;
import java.net.*;

public class ChatSpeaker implements Runnable {
  private static PrintStream err = System.err;

  private List outgoing;  // Outgoing messages
  private BufferedOutputStream bos;

  public ChatSpeaker() {
    this.outgoing = new ArrayList();
  }

  public void setSocket(Socket socket) {
    try {
      // Make streams for reading and writing
      this.bos =
        new BufferedOutputStream(
                    socket.getOutputStream());
    } catch (IOException ex) {
      err.println("** IOException: " + ex);
      System.exit(1);
    }
  }
```


`ChatSpeaker` continued

```
  public void run() {
    while (true) {
      try {
        // Is there a message to send?
        synchronized(this.outgoing) {
          if (!this.outgoing.isEmpty()) {
            ChatMessage m =
              (ChatMessage) this.outgoing.remove(0);
            ObjectOutputStream out =
               new ObjectOutputStream(bos);
            out.writeObject(m);
            out.flush();

            if (m.isLastMessage()) {
              // Send the last message and
              // then go home
              break;
            }
          }
          // Wait for a message
          this.outgoing.wait();
        }

      } catch (InterruptedException ex) {
        break;
```


`ChatSpeaker` concludes

```
      } catch (IOException ex) {
        err.println("** IOException: " + ex);
        System.exit(1);
        break;  // Need for compilation

      }
    }
  }

  public void sendMessage(ChatMessage message) {
    synchronized(this.outgoing) {
      this.outgoing.add(message);
      this.outgoing.notify();
    }
  }
}
```

Note synchronization on the `outgoing` queue

`ChatMessage` objects are serialized and sent over the wire

A `ChatListener` receives `ChatMessage`s

```
package edu.pdx.cs399J.net;
import java.io.*;
import java.util.*;
import java.net.*;

public class ChatListener implements Runnable {
  private static PrintStream err = System.err;

  private List incoming;  // Incoming messages
  private BufferedInputStream bis;

  public ChatListener() {
    this.incoming = new ArrayList();
  }

  public void setSocket(Socket socket) {
    try {
      this.bis =
        new BufferedInputStream(socket.getInputStream

    } catch (IOException ex) {
      err.println("** IOException: " + ex);
      System.exit(1);
    }
  }
```

`ChatListener` continued

```
  public void run() {
    while (true) {
      try {
        // Is there a message to receive?
        ObjectInputStream in =
          new ObjectInputStream(bis);
        ChatMessage m = (ChatMessage) in.readObject()
        if (m != null) {
          synchronized(this.incoming) {
            this.incoming.add(m);
          }

          if (m.isLastMessage()) {
            break;
          }
        }

      } catch (ClassNotFoundException ex) {
        err.println("** Could not find class: " + ex)
        System.exit(1);

      } catch (IOException ex) {
        err.println("** IOException: " + ex);
        System.exit(1);
      }
    }
  }
```


`ChatListener` concluded

```
  public List getMessages() {
    List messages = new ArrayList();
    synchronized(this.incoming) {
      // Why can't we just return this.incoming?

      messages.addAll(this.incoming);
      this.incoming.clear();
    }

    return messages;
  }
}
```

The fact that the `ObjectInputStream`'s constructor blocks until the stream is available complicates our design

Each message must know if it is the last

It would be more elegant if some other thread could tell the listener to stop

A `ChatCommunicator` handles communication

A `ChatCommunicator` attempts to make a connection to the other chat program. Then, it starts the speaker and the listener.

```
package edu.pdx.cs399J.net;
import java.io.*;
import java.util.*;
import java.net.*;

public class ChatCommunicator implements Runnable {
  private static PrintStream err = System.err;
  private int port;       // Where the socket is
  private ChatSpeaker speaker;    // Send messsages
  private ChatListener listener;  // Receives message

  public ChatCommunicator(int port) {
    this.port = port;
  }

  public void startup() {
    this.speaker = new ChatSpeaker();
    this.listener = new ChatListener();
    (new Thread(this)).start();
  }

  // continued...
```


`ChatCommunicator` continued...

```
public void run() {
  // Attempt to make a socket
  Socket socket = null;
  try {
    socket = new Socket("localhost", port);

  } catch (IOException ex) {
    // Nobody listening
  }

  if (socket == null) {
    // Listen
    try {
      ServerSocket server =
        new ServerSocket(port, 10);
      socket = server.accept();
    } catch (IOException ex) {
      err.println("** IOException: " + ex);
      System.exit(1);
    }
  }

  this.speaker.setSocket(socket);
  this.listener.setSocket(socket);

  (new Thread(this.speaker)).start();
  (new Thread(this.listener)).start();
}
```

`ChatCommunicator` delegates

`ChatCommunicator` delegates some calls to the speaker and listener. Note that these methods will be invoked in a thread other than the ones that run the speaker and listener.

```
  /**
   * Delegates to the <code>ChatSpeaker</code>
   */
  public void sendMessage(ChatMessage message) {
    this.speaker.sendMessage(message);
  }

  /**
   * Gets messages from the <code>ChatListener</code>
   */
  public List getMessages() {
    return this.listener.getMessages();
  }
}
```


The user runs a `ChatSession`

```
package edu.pdx.cs399J.net;
import java.io.*;
import java.net.*;
import java.util.*;

public class ChatSession {
  private static PrintStream out = System.out;
  private static PrintStream err = System.err;

  public static void main(String[] args) {
    String owner = args[0];
    int port = 0;

    try {
      port = Integer.parseInt(args[1]);
    } catch (NumberFormatException ex) {
      err.println("** Bad port number: " + args[1]);
      System.exit(1);
    }

    out.println("Establishing connection");

    // Make a new ChatCommunicate and start it up
    ChatCommunicator communicator =
      new ChatCommunicator(port);
    communicator.startup();
```


`ChatSession` continued

```
    // Prompt for input, read from the command line
    // until the "bye" message is inputted.
    try {
      InputStreamReader isr =
        new InputStreamReader(System.in);
      BufferedReader br = new BufferedReader(isr);

      String line = "";
      while (!line.trim().equals("bye")) {
        // Print and read messages from the listener
        Iterator messages =
          communicator.getMessages().iterator();
        while (messages.hasNext()) {
          out.println(messages.next());
        }

        // Prompt for user input
        out.print(owner + "> ");
        out.flush();

        line = br.readLine();

        if (!line.trim().equals("")) {
          ChatMessage message =
            new ChatMessage(owner, line);
          communicator.sendMessage(message);
        }
      }
```


`ChatSession` concluded

```
      out.println("Waiting for other side to " +
                  "shut down");

    } catch (IOException ex) {
      err.println("** IOException: " + ex);
      System.exit(1);
    }
  }
}
```

```
$ java -cp ~/classes edu.---.ChatSession Dave1 12345
Establishing connection
Dave1> Are you there?
Dave2 [2:55:41 PM]> Hello
Dave1> bye
Waiting for other side to shut down
```

```
$ java -cp ~/classes edu.---.ChatSession Dave2 12345
Establishing connection
Dave2> Hello
Dave2> This is cool
Dave1 [2:55:45 PM]> Are you there?
Dave2> bye
Waiting for other side to shut down
```

Summary

Java has built-in facilities for performing networking

- `URL` for accessing locations on the internet
- Sockets with stream-like behavior
- Also supports UDP datagrams

Object serialization is used to convert Java objects into byte streams

- Happens automagically – usually little work on the programmer's part

Java also supports multiple threads of execution and has a number of mechanisms to control concurrent programs

- `Thread` class and `Runnable` interface
- `synchronized` methods and code blocks
- `wait`/`notify` methods