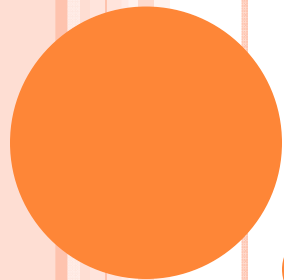
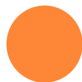
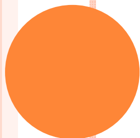
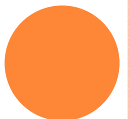




CS 4485

Threads, concurrency and serialization

SUMMARY

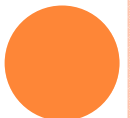
1. Threads
2. Concurrency
3. Synchronization
4. Serialization
5. Sockets



THREADS

- Processes have their own address space (variables and data structures)
- Threads share address space.
  - different program counter and call stack
- Threads allow: multiple concurrent interacting components

- e.g. web servers use multiple threads to handle multiple simultaneous web browser requests



THREADS

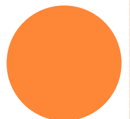
```
class MyThread extends Thread {
private String message;
public MyThread(String m) {message = m;}
public void run() {
for(int r=0; r<20; r++)
System.out.println(message);
}
}
public class ABC {
public static void main(String[] args) {
MyThread t1,t2;
t1=new  MyThread("primo thread");
t2=new  MyThread("secondo thread");
t1.start();
t2.start();
}}
```

Thread object is defined by extending the Thread class

Override the run()

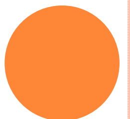
Thread object is instantiated using 'new'

To run the thread you must invoke the start()



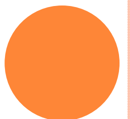
ALTERNATIVE METHOD FOR IMPLEMENTING THREADS (RUNNABLE INTERFACE)

```
class MyThread implements Runnable {
private String message;
public MyThread(String m) {message = m;}
public void run() {
for(int r=0; r<20; r++)
System.out.println(message);
}
}
public class ABC {
public static void main(String[] args) {
Thread t1, t2;
MyThread r1, r2;
r1 = new MyThread("primo thread");
r2 = new MyThread("secondo thread");
t1 = new Thread(r1);
t2 = new Thread(r2);
t1.start();
t2.start();
}
}
```



CONCURRENT PROGRAMMING IN JAVA

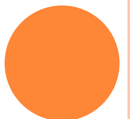
- Non determinism :
  - same program on different machines : different result
  - depends on how internal scheduling is done: hardware dependence.
- Concurrency issues:
  - Safety: Read/Write conflicts
  - Deadlocks: resource contention



EXAMPLES:

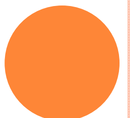
```
public class RGBColor {
private int r;
private int g;
private int b;

public void setColor(int r, int g, int b) {
checkRGBVals(r, g, b);
this.r = r;
this.g = g;
this.b = b;
```

write/write conflict!!



LOCKS

- Read- write and write – write conflicts avoided using locks.
- Use the 'synchronized' key word
- Two invocations of a *synchronized* method cannot interleave
- When a thread is executing a *synchronized* method
  - All other threads that invoke *synchronized* method on the same object, block until the first thread finishes
  - Synchronizing constructors does not make sense



EXAMPLE

```
synchronized (object){
// Lock is held
…
}
// Lock is released
```

Synchronize objects

```
synchronized void f() { /* body */ }
```

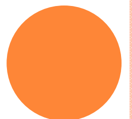
is equivalent to

```
void f() { synchronized(this) { /* body
   */ } }
```

Synchronize methods

```
synchronized (point) {
point.x = 5;
point.y = 7; }
```

```
synchronized(point) {
if(point.x > 0) {
...
}}
```

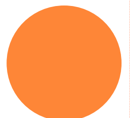


DEADLOCKS

- Circular wait
  - Deadlock is possible when two or more objects are mutually accessible from two or more threads, and each thread holds one lock while trying to obtain another lock already held by another thread
  - Although fully synchronized atomic objects are always safe, they may lead to deadlock

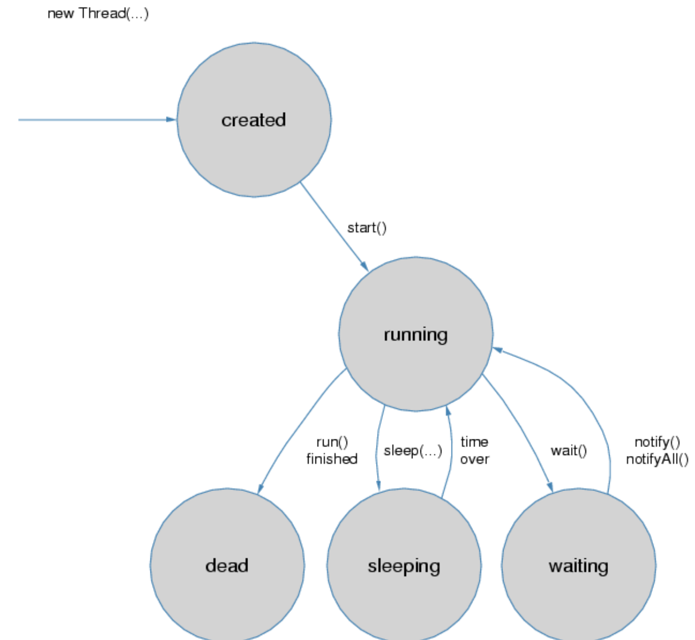
```
class C { // Do not use
private long value;
synchronized long getValue() { return
value; }
synchronized void setValue(long v) {
value = v; }

synchronized void swapValue(C other)
{
long t = getValue();
long v = other.getValue();
setValue(v);
other.setValue(t);
}
}
```

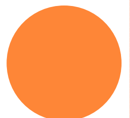
if 2 instances of **Class C, say a and b, are** concurrently invoking **swapValue, the program may** block indefinitely because **a** needs **b's lock and vice** versa



PRODUCER-CONSUMER EXAMPLE

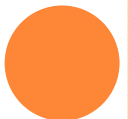
```
class Queue{
protected int spaces;
protected int capacity;
Queue(int n)
{capacity = spaces = n;}

synchronized void remove() throws
    InterruptedException {
while (spaces==0) wait();
++spaces;
notify();
}
synchronized void put() throws
    InterruptedException {
while (spaces==capacity) wait();
--spaces;
notify();
}
}
```



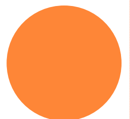


SERIALIZATION

- Serialization involves saving the current state of object to a stream and restoring an equivalent object from the stream.
- Complex data can be stored in an object form
- Objects can be copied over sockets to remote applications



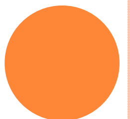
- An object can be serialized if:
  - its class is marked serializable
  - All its data members are serializable
- Each related copy is serialized only once
  - Eg. Object A contains B & C. Both B & C contain D.
  - Serialized copy contains A,B,C and one copy of D.
- If not serializable:
  - java.io.NotSerializableException is thrown



- What makes a class serializable?
  - It implements java.io.Serializable

```
public class MySerialObject implements Serializable {
private transient String password;
private String s;
MyObject mo;
....
}
```

  - Most java classes are serializable: Hashtables, String etc.
  - Objects not serializable: threads, sockets, sessions etc.



- Serialization is provided through two filters
  - ObjectOutputStream to write (serialize) objects
  - ObjectInputStream to read (deserialize) objects

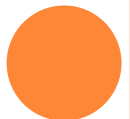
```
ObjectOutputStream out = null;
   try
   {
    PersistentTime time = new
PersistentTime();
    fos = new FileOutputStream(filename);
     out = new ObjectOutputStream(fos);
     out.writeObject(time);
     out.close();
   }
catch{
}
```

```
  PersistentTime time = null;
  FileInputStream fis = null;
  ObjectInputStream in = null;
   try
   {
     fis = new
FileInputStream(filename);
     in = new
ObjectInputStream(fis);
     time =
(PersistentTime)in.readObject();
     in.close();
   }
catch{
}
```

- To serialize a object:
  - Create an OutputStream to a destination
  - Use it to create an ObjectOutputStream
  - Call its writeObject() method to serialize one object
  - Close stream

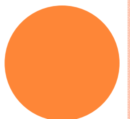
- If an object contains sockets or threads?
  - use "transient" eg. *private transient Thread _x;*
  - You must mark as transient any field that cannot or should not be serialized

- Externalizable: implementing class will handle serialization on its own:
  - Class defines writeExternal() method to write out the stream, and a corresponding readExternal() method to read.
  - *Encrypt and decrypt*

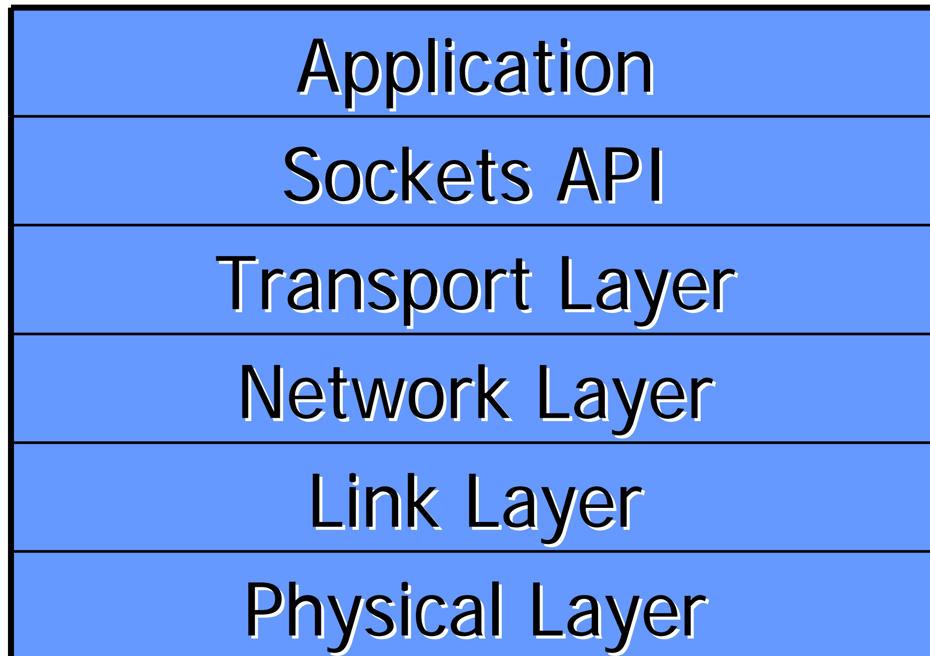


SOCKETS AND API

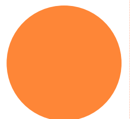
- API stands for Application Programming Interface
- Sockets are APIs which we use as an interface to the transport layer
- APIs hide all complexities of the lower layers



SOCKETS…



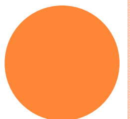
- Socket is an abstraction provided to a programmer to send or receive data to another process.
- Like a "door" at either side of the communication through which one can send receive data.
- *IP address and port number* identifies each socket



Opening a Socket:

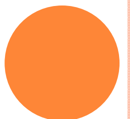
- On Client:

```
import java.net.*; // for Socket
    Socket MyClient;
    try {
            MyClient = new Socket("Machine name", PortNumber);
    }
    catch (IOException e) {
             System.out.println(e);
    }
```



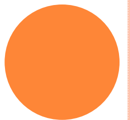
- Server:

```
import java.net.*; // for Socket
ServerSocket MyService;
    try {
             MyService = new ServerSocket(PortNumber);
    //e.g PortNumber=2000 always greater than 1024
    }
    catch (IOException e) {
            System.out.println(e);
     }
```



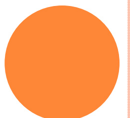
LISTEN AND ACCEPT ON SERVER

```
Socket clientSocket = null;
try {
clientSocket = MyService.accept();
}
catch (IOException e) {
System.out.println(e);
}
```



SEND AND RECEIVE DATA

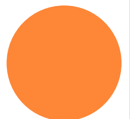
```
// Open input and output streams
try {
    clientSocket = MyService.accept();
    is = new DataInputStream(clientSocket.getInputStream());
//BufferedReader in = new BufferedReader(new
    InputStreamReader(clientSocket.getInputStream()));

    os = new PrintStream(clientSocket.getOutputStream());
    // As long as we receive data, echo that data back to the client.

    while (true) {
            line = is.readLine();
            os.println(line);
    }

}
catch (IOException e) {
     System.out.println(e);
}
```



Remember to close the socket : clientsocket.close()

CLIENT:

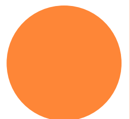
```
import java.net.*;
import java.io.*;

public class tcpClient0 {
  public static void main(String[] args)
      throws IOException {
      if (args.length != 1) {
              System.out.println("Usage: java tcpClient0  hostname");
              System.exit(0);
                      }
    // Connect to the given machine
    InetAddress addr = InetAddress.getByName(args[0]);

    // System.out.println("addr = " + addr);
    // try to connect tcpServer
    Socket socket =  new Socket(addr, 12345);
    // Guard everything in a try-finally to make
    // sure that the socket is closed:

     try {
      // Output is automatically flushed
      // by PrintWriter:
      PrintWriter out = new PrintWriter( new BufferedWriter(
          new OutputStreamWriter(
            socket.getOutputStream())),true);

      // send "HI"  to the tcpServer
      out.println("HI");
    } finally {
                  // System.out.println("closing the socket...");
      socket.close();
    }
  }
} ///:~
```



SERVER:

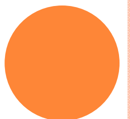
```
import java.io.*;
import java.net.*;

public class tcpServer0 {
  // Choose a port outside of the range 1-1024:
  public static final int PORT = 12345;
  public static void main(String[] args)
      throws IOException {

    // Create a Server Socket and binds it to the port 12345
    ServerSocket s = new ServerSocket(PORT);
    // System.out.println("Started: " + s);

   try {
      while (true)
                 {
      // Blocks until a connection occurs:
      Socket socket = s.accept();
      try {

        // System.out.println("Connection accepted: "+ socket);
                // Read from InputStream from the Socket
        BufferedReader in =
          new BufferedReader(
            new InputStreamReader(
              socket.getInputStream()));

          // get the Recived string
  String str = in.readLine();

        // print the recieved message to the
STND output
        System.out.println("message receievd: "
+ str);

     // Always close the two sockets...
     } finally {
                //
System.out.println("closing...");
       socket.close();
     }
   } //while true
   } finally {
    s.close();
   }
   }

} ///:~
```

Reference:
http://www.cs.odu.edu/~cs476/SocketProgramming/java/java.htm



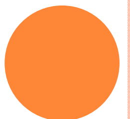
DATAGRAM

```
import java.io.*;
import java.net.*;
public class Sender {
  public static void main(String[] args) throws IOException {
    InetAddress addr = InetAddress.getByName(args[0]);
    byte[]      buf  = args[1].getBytes();
    DatagramPacket p =
                new DatagramPacket(buf, buf.length, addr, 1115);
    DatagramSocket socket = new DatagramSocket();
    socket.send(p);
  }
}
```



DATAGRAM RECEIVER

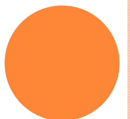
```
import java.io.*;
import java.net.*;
public class Receiver {
  public static void main(String[] args) throws IOException {
    DatagramSocket socket = new DatagramSocket(1115);
    byte[] buf = new byte[256];
    DatagramPacket p = new DatagramPacket(buf, buf.length);
    socket.receive(p);
    String s = new String(p.getData(), 0, p.getLength());
    System.out.println(p.getAddress().getHostName() + ": " + s);
  }
}
```



THREADS AND SOCKETS

```
while (true) {
 try {
  Socket s = ss.accept();
  MyThread mt = new MyThread(s);
   mt.start();
 }
catch (IOException e) {}
```

Store all incoming connections in a queue and process them



REFERENCES

- Concurrent Programming and Threads:
  - http://research.microsoft.com/en-us/um/people/costa/slides/concurrentprogramming.pdf
- Java Sockets:
  - http://www-compsci.swan.ac.uk/~csneal/InternetComputing/JavaSockets.pdf

Lot of sites on the Internet have tons of information on socket programming and Threads.