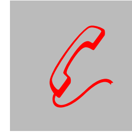
Notice:

These draft slides are provided as a courtesy to my students to assist in taking notes in class. They are subject to change and revision throughout the week as improvements occur to me.

I/O & network programming



The ins and outs of Java

I/O is always complicated

Streams

objects that represent
where data flows to and from

e.g., `InputStream`, `OutputStream`, `PrintStream`, etc.

Streams

For example, the following variable of the class `System` is initialized when a Java program starts to run:

```
public static PrintStream out
```

Streams

A `PrintStream` object has methods `print`, `println`, `write`, `flush`, etc.

e.g., `System.out.println(...`

Readers & Writers

With Java 1.1, direct use of streams is *deprecated*

Instead, 'wrap' the *Stream* object in a *Reader* or *Writer* object:

```
PrintWriter pout =
    new PrintWriter(System.out);
```

Adds international character support

Readers & Writers

In some cases, you can create readers and writers directly:

```
FileWriter fw = new FileWriter("temp");

StringReader sr =
  new StringReader("this is the input");
```

Files

Java supports many varieties of files:

- 'random' access
- sequential access
  - primitive data types
  - straight text

Buffered reading

Ordinary readers only support reading individual characters; i.e., no readline

So, once again, wrap:

```
infile = new BufferedReader(
      new FileReader(filename));
```

Buffered reading

A `BufferedReader` object has a `readLine` method that lets us read in a line of text in a natural way:

```
String message = stdin.readLine();
```

This method is specified as

```
public String readLine()
        throws IOException
```

Example: read a file into a `TextArea`

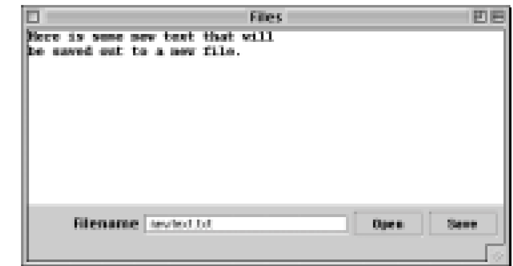
Example: read a file into a `TextArea`

```
private void openFile(String filename) {
  String inline;
  BufferedReader infile = null;

  ta.setText(null);
  try {...
  }
  catch (FileNotFoundException fnfe) {}
  catch (IOException ioe) {}
  finally {...
  }
}
```

```
private void openFile(String filename) {
  String inline;
  BufferedReader infile = null;

  ta.setText(null);
  try {
    infile = new BufferedReader(
        new FileReader(filename));
    inline = infile.readLine();
    while (inline != null) {
      ta.append(inline + "\n");
      inline = infile.readLine();
    }
  }
```

```
  catch (FileNotFoundException fnfe) {}
  catch (IOException ioe) {}
  finally {
    try {
      if (infile != null) infile.close();
    }
    catch (IOException ioe) {}
  }
}
```

Example extended: write the file out



Example extended: write the file out

We need to turn the `TextArea` contents into a series of lines of text

Will use the `StringTokenizer` class

Example extended: write the file out

The `StringTokenizer` class has methods to break up a string into individual tokens of words, numbers, lines, etc.

`countTokens()`: the number of tokens
`nextToken()`: the next token

A constructor parameter specifies the delimiters

```
private void saveFile(String filename) {
  PrintWriter outfile = null;
  File sFile = new File(filename);
  String outline;
  StringTokenizer outtext;

  if (sFile.exists()) {
    System.err.println("File... exists.");
    Toolkit.getDefaultToolkit().beep();
    return;
  }
  outtext = new
    StringTokenizer(ta.getText(),"\n");
  int nTokens = outtext.countTokens();
```

```
try {
  outfile = new PrintWriter(
      new FileWriter(sFile));
  for (int i = 0; i < nTokens; i++) {
    outline = outtext.nextToken();
    outfile.println(outline);
  }
}
catch (IOException ioe) {}
finally {
  if (outfile != null) outfile.close();
}
}
```

The `File` class

Objects represent file or directory paths

`delete()` – *Delete file specified by this object*

`exists()` – *Test if this file exists*

`isFile()` – *Test if object represents a file*

The `File` class

`getName()` – *Get the name of the file represented by this object*

`getPath()` – *Get the pathname of the file*

`getParent()` – *Get the parent part of the pathname, or null*

The `File` class

`list()` – *Returns a list of the files in the directory represented by this object*

`list(FilenameFilter)` – *Returns the files that satisfy the specified filter*

`mkdir()` – *Creates a directory whose pathname is specified by this File object*

The `File` class

`lastModified()` – *Get the time the file was last changed*

`length()` – *Get the length of the file*

…

Parsing input

Each primitive data type in Java has an associated wrapper class
e.g., `Integer`, `Double`, `Boolean`, etc.

Each has object ↔ string conversion methods:
- `toString` – object to string
- `valueOf` – string to object (a static method)

```
Integer n = Integer.valueOf("17");
```

Parsing input

Wrapper classes also have methods to convert from an object to the corresponding primitive data value:

```
int i = n.intValue();
```

Similarly…
`doubleValue`, `booleanValue`, etc.

Parsing input

Putting it together:

```
String s = "-17.83";
double d =
    Double.valueOf(s).doubleValue();
```

Similarly for booleans

Parsing input

Putting it together:

```
String s = "1783";
int i =
    Integer.valueOf(s).intValue();
```

In this case, there's a shorter equivalent:

```
int i = Integer.parseInt(s);
```

So much for primitive types...

What about objects?

For simple cases, just write out the data then...
– read it back in
and
– reconstruct the object

Read in has to know how write out was done

Serializable

Implements the `Serializable` interface:

```
package java.io;
public interface Serializable {
}
```

works as long as
- all instance variables are serializable
- superclass has a zero parameter constructor

Serialization

An object is serialized by being passed as a parameter to a `writeObject` method that is applied to an `ObjectOutputStream` object

```
ObjectOutputStream oos =
  new ObjectOutputStream(
    new FileOutputStream("x.ser"));
oos.writeObject(new Point(10,25));
```

Deserialization

An object is deserialized by being passed as a parameter to a `readObject` method that is applied to an `ObjectInputStream` object

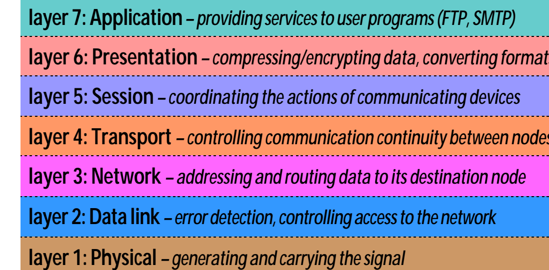
```
ObjectInputStream ois =
  new ObjectInputStream(
    new FileInputStream("x.ser"));
Point p = (Point)ois.readObject();
```

Network programming

To make it work requires enormous amounts of coordination at many different levels

Historically, network programming has been error-prone, difficult, and complex. The programmer... needed to understand the various "layers" of the networking protocol... It was a daunting task. – *Bruce Eckel*

ISO/OSI reference model



Socket-based communications

Networking treated as if it were file I/O

Sockets are a software abstraction of what a stream 'plugs into'

Two forms:
- stream sockets
- datagram sockets

Stream sockets

Connection-based (TCP)

Logically, a continuous stream from process to process

Functionally, packets of data are sent individually, (along different routes as traffic demands) are resent if necessary, and re-assembled in the proper order

Datagram sockets

Packet-oriented (UDP)

Packets sent with no guarantee of delivery or order

Faster but less reliable and more arduous

Addresses,...

Machines are identified by IP addresses
e.g. 128.255.155.57

For convenience, also have names:
Domain Naming System

Subdomains are registered with name servers
e.g., current top-level domains: com, edu, jp
Thus bull.cs.uiowa.edu

... ports, and URLs

Universal Resource Locators,
e.g., http://www.cs.uiowa.edu/...

Ports – like sockets, are not physical things

Allow a program to know
when a packet is meant for it
e.g., 13 or 80 or 8001

A very simple example

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

public class URLgrab {

  public static void main(String args[])
            throws IOException {
    BufferedReader in = null;
    String inStr = "";
    URL fURL =
        new URL("http://128.255.155.57
                  /test.html");
```

```
    try {
      in = new BufferedReader (new
          InputStreamReader(
              fURL.openStream()));
      while (!inStr.equals("</html>")) {
        System.out.println(
            inStr = in.readLine());
      }
    } catch (IOException ioe) {}
    finally { in.close(); }
  }
```

Using sockets directly

Constructors

`Socket(InetAddress, int)`

Creates a stream socket and connects it to the specified port number at the specified IP address

`Socket(String, int)`

Creates a stream socket and connects it to the specified port number on the named host

Using sockets directly

Methods

`close()`

Closes this socket

`getInputStream()`

Returns an input stream for this socket

`getOutputStream()`

Returns an output stream for this socket

adapted from Eckel

```
public class JabberClient {
  static final int PORT = 8080;

  public static void main(String[] args)
          throws IOException {
    InetAddress addr =
      InetAddress.getByName("localhost");
    Socket sk = new Socket(addr, PORT);
    try {
    ...
    } finally {socket.close();}
  }
}
```

```
    try {
      BufferedReader in = new BufferedReader(
          new InputStreamReader(
              sk.getInputStream()));
      PrintWriter out = new PrintWriter(
          new BufferedWriter(
            new OutputStreamWriter(
              sk.getOutputStream())),true);
      for(int i = 0; i < 10; i ++) {
        out.println("howdy " + i);
        String str = in.readLine();
        System.out.println(str);
      }
    }
```

Waiting to connect (a server's job)

Constructors

`ServerSocket(int)`

Creates a server socket on a specified port

A server socket waits for connection requests
to come in over the network

Waiting to connect

Methods

`accept()`
Waits for a connection to be made to this server socket and accepts it; a stream socket representing the connection is returned

`close()`
Closes this server socket

Waiting to connect

Methods

`setSoTimeout(int)`
An `accept` call on this server socket blocks for only the specified time (in milliseconds); if this time expires, a n `InterruptedIOException` is thrown

'Blocking' is a concept that pertains to concurrent programming; for now, it basically means *waiting*

adapted from Eckel

```
public class JabberServer {
   static final int PORT = 8080;

   public static void main(String[] args)
            throws IOException {
      ServerSocket ssk = new ServerSocket(PORT);
      try {
         Socket sk = ssk.accept();    // waits
         try {
         ...
         } finally { sk.close(); }
      } finally { ssk.close(); }
   }
}
```

```
try {
   BufferedReader in = new BufferedReader(
         new InputStreamReader(
            sk.getInputStream()));
   PrintWriter out = new PrintWriter(
         new BufferedWriter(
            new OutputStreamWriter(
               sk.getOutputStream())),true);
   while (true) {
      String str = in.readLine();
      if (str.equals("END")) break;
      System.out.println("Echoing: " + str);
      out.println(str);
   }
}
```