Networks and Internet Programming

Basic Network Concepts

Eng. Asma Abdel Karim
Computer Engineering Department

Outline

- What is a Network?
- The Layers of a Network.
- IP, TCP and UDP.
- IP Addresses and Domain Names.
- Ports.
- The Internet.
- Firewalls.
- Proxy Servers.

What is a Network?

- A ***network*** is a collection of computers and other devices that can send data to and receive data from one another.
- Connectivity:
  - Wires - electromagnetic waves.
  - Wireless - radio waves.
  - Fiber-optic cables - light waves.
- Such connections carry data between one point in the network and another. This data is represented as bits of information (ON/OFF, 0/1).

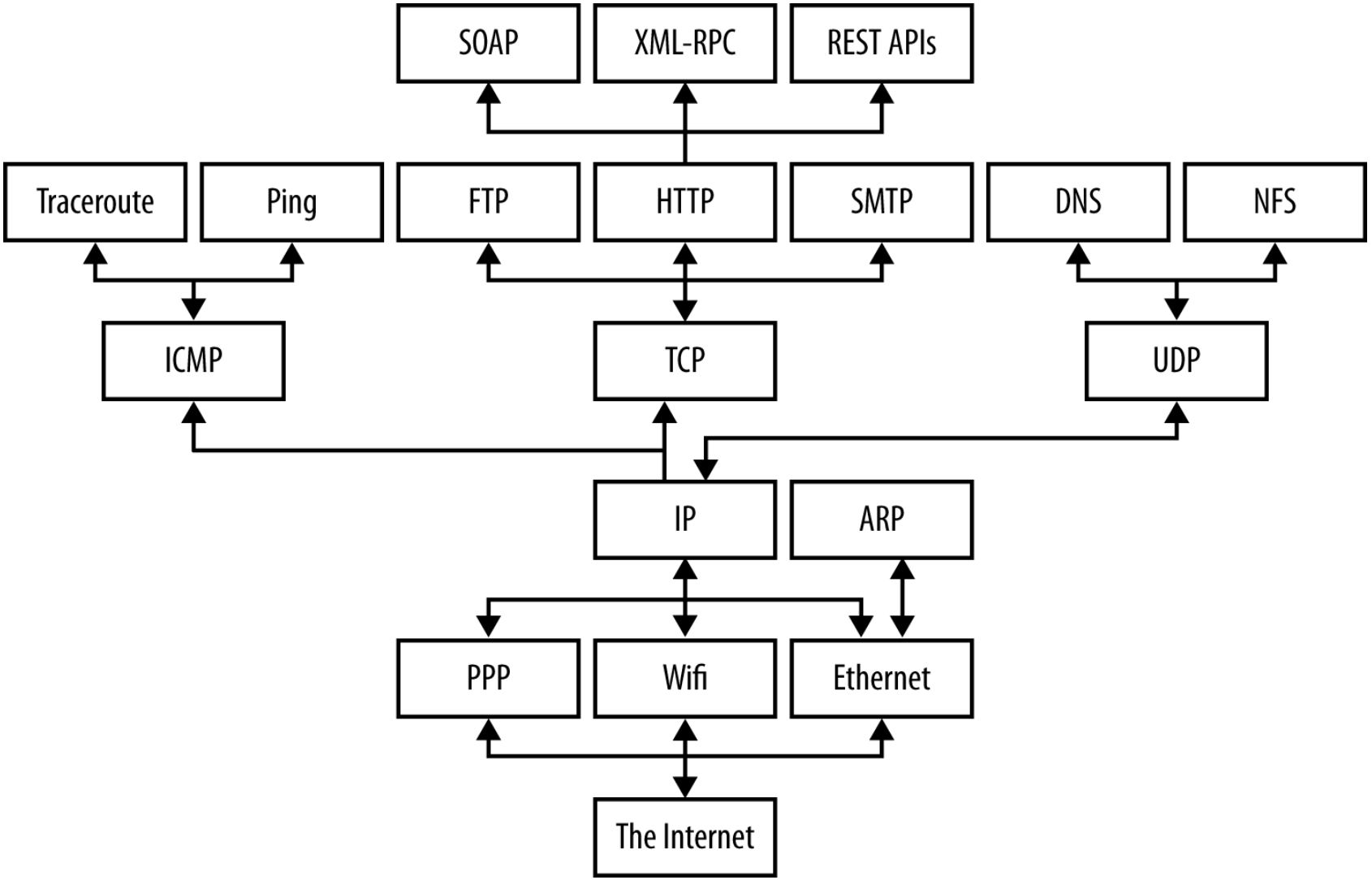
What is a Network? (Cont.)

- Each machine on a network is called a ***node***.
  - Nodes are computers, printers, routers, bridges, gateways, etc…
  - Nodes that are fully functional computers are also called ***hosts***.
- Every network node has an ***address***, a sequence of bytes that uniquely identifies it.
  - Physical address.
  - Logical address.

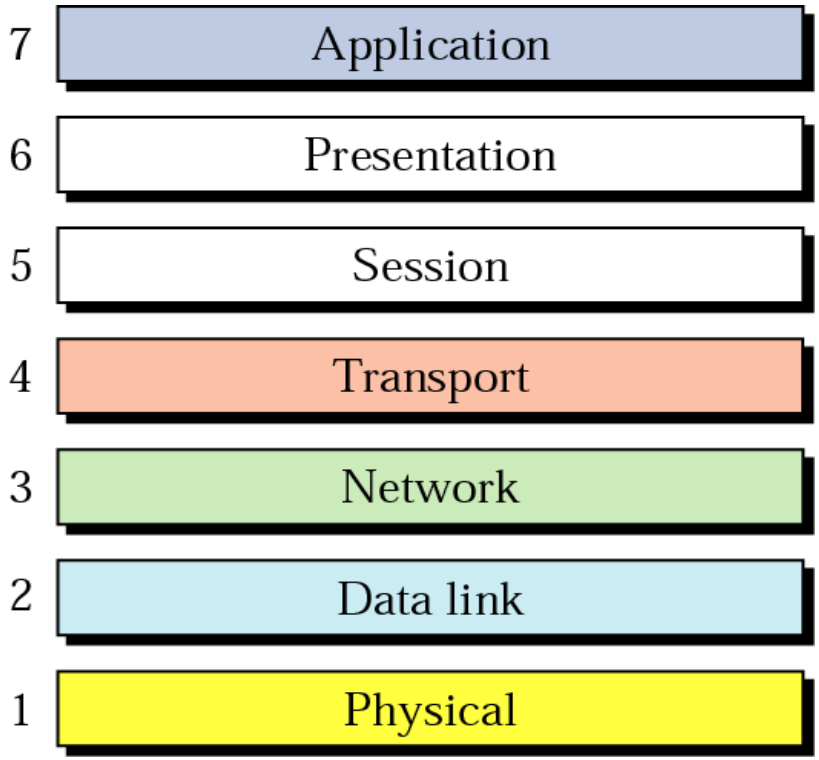
What is a Network? (Cont.)

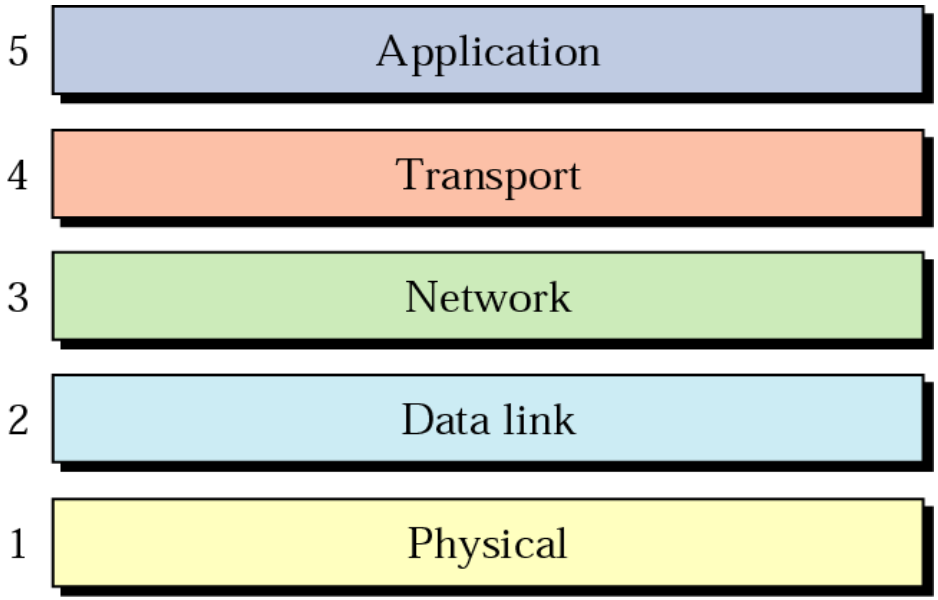
- All modern computer networks are ***packet-switched*** networks.
  - Data traveling on the network is broken into chunks called ***packets*** and each packet is handled separately.
  - Each packet contains information about who sent it and where it's going.
- A ***protocol*** is a precise set of rules defining how computers communicate.
  - The format of addresses, how data is split into packets, and so on.
  - HTTP, IP, TCP, UDP, IEEE 802.3, etc…

The Layers of a Network

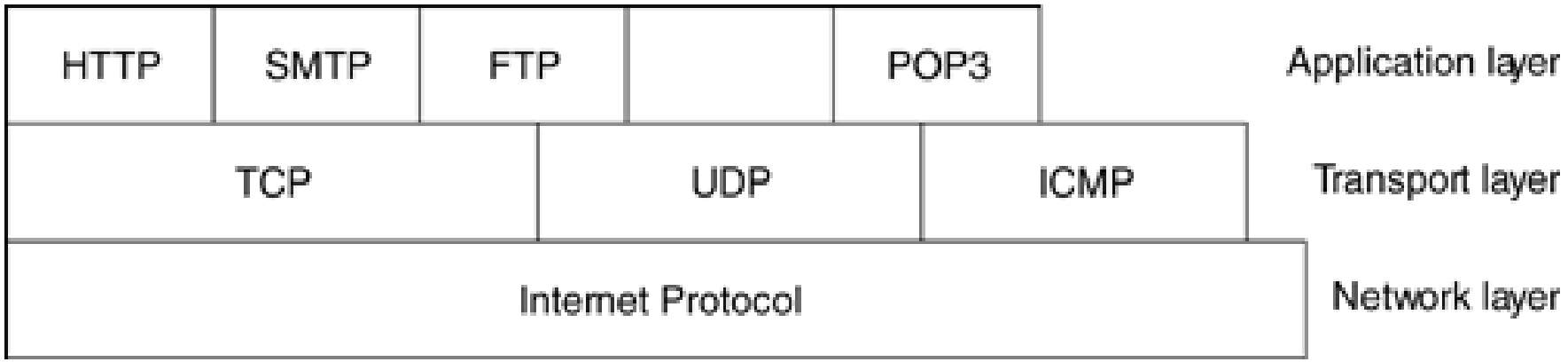


The Layers of a Network
OSI Model vs. TCP/IP Model

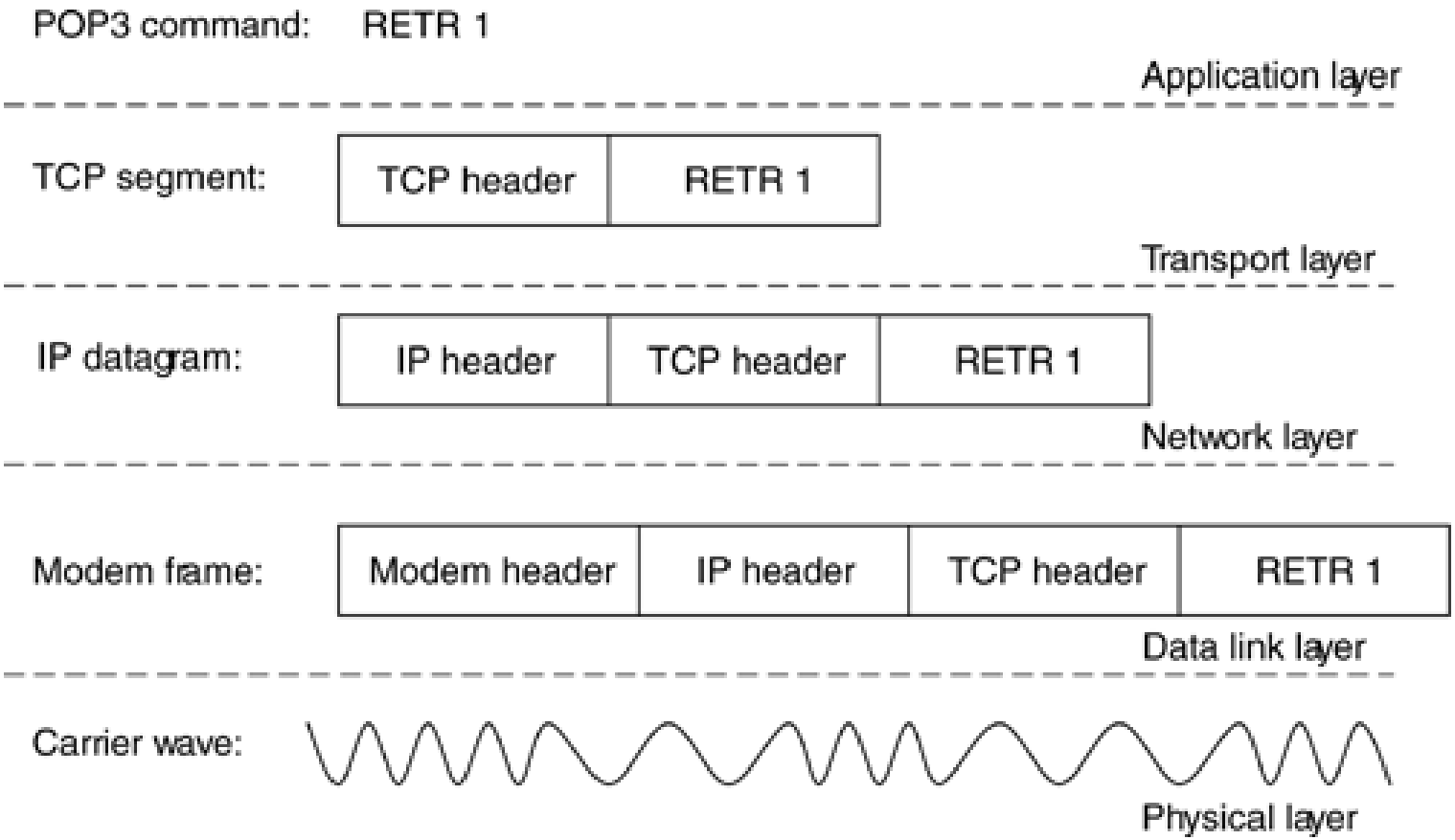




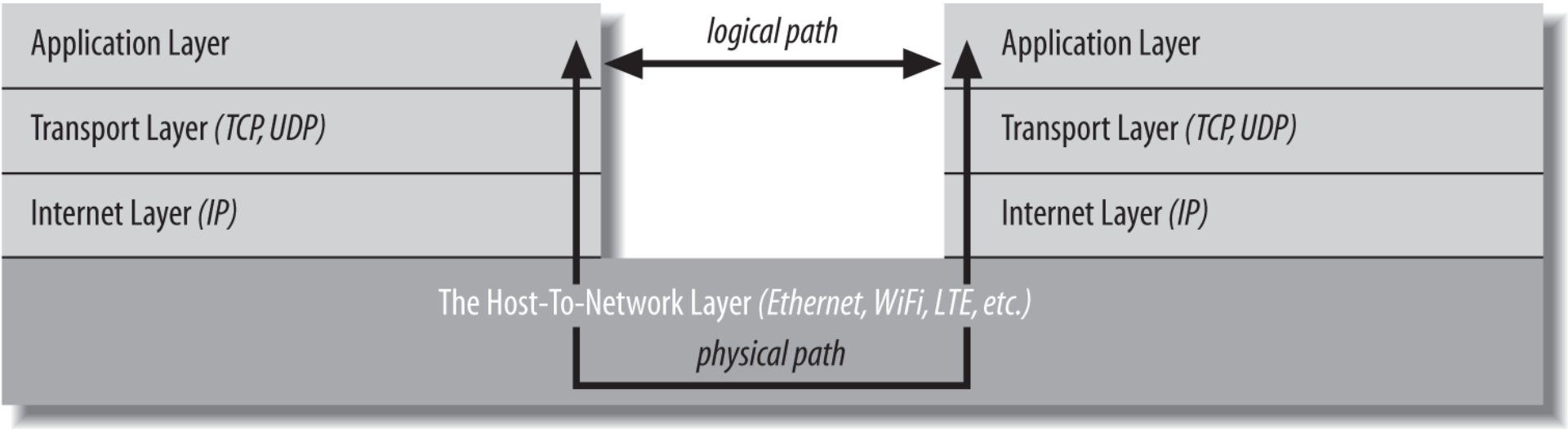
TCP/IP Protocol Suite Layers



TCP/IP Protocol Suite Layers (Example)



The Layers of a Network (Cont.)



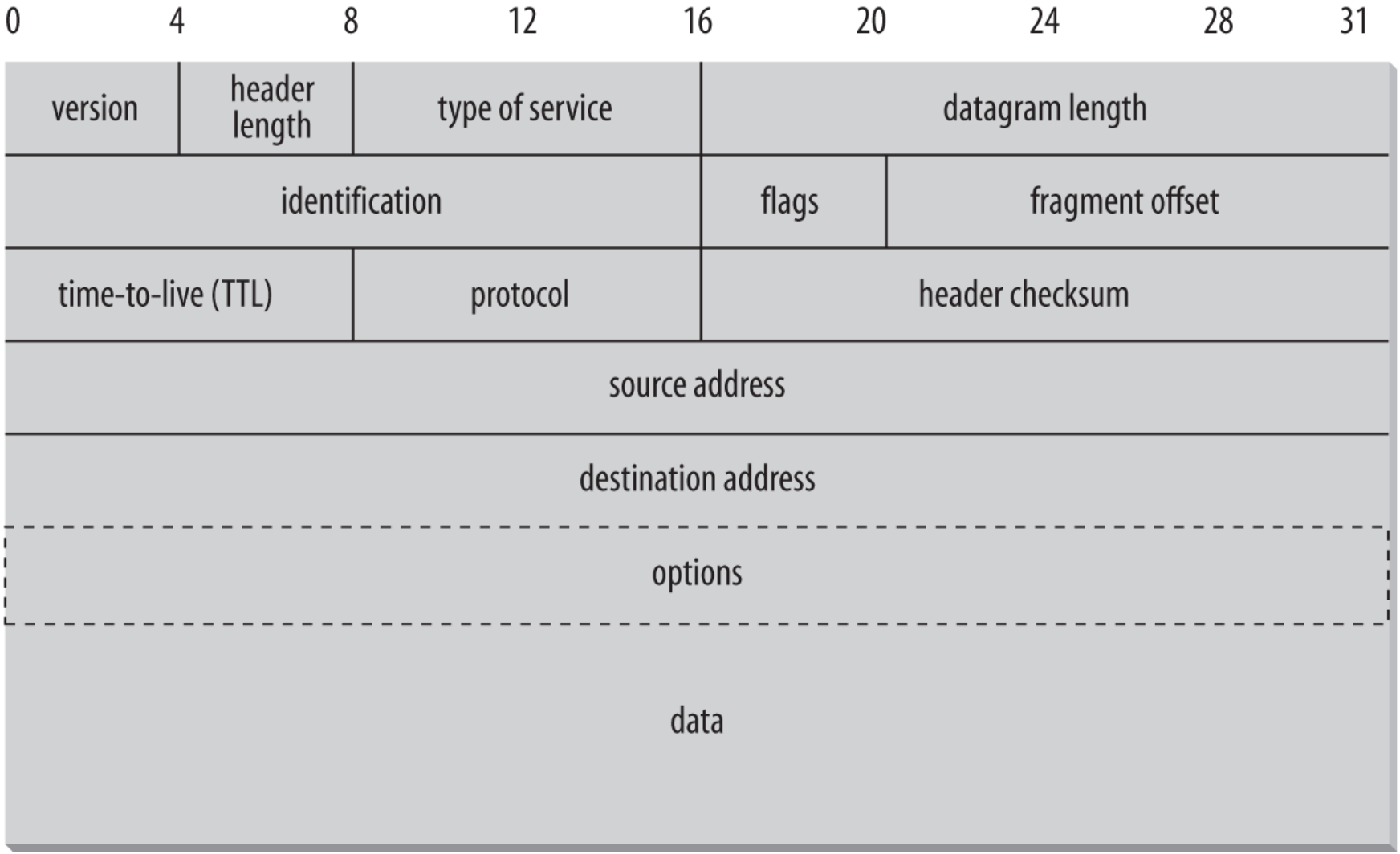
The Host-To-Network Layer

- The ***link*** layer, ***data link*** layer, or ***network interface*** layer.
  - Defines how a particular network interface—such as an Ethernet card or a WiFi antenna—sends IP datagrams over its physical connection to the local network and the world.
- The part of the host-to-network layer made up of the hardware that connects different computers (wires, fiber-optic cables, radio waves, or smoke signals) is called the ***physical*** layer of the network.
- The primary reason you'll need to think about the host-to-network layer and the physical layer, if you need to think about them at all, is performance.
  - However, whichever physical links you encounter, the APIs you use to communicate across those networks are the same.

The Internet Layer

- A network layer protocol defines:
  - how bits and bytes of data are organized into the larger groups called packets, and
  - the addressing scheme by which different machines find one another.
- The ***Internet Protocol*** (IP) is the most widely used network layer protocol in the world and ***the only network layer protocol Java understands***.
- In fact, it's two protocols:
  - IPv4, which uses 32-bit addresses, and
  - IPv6, which uses 128-bit addresses and adds a few other technical features to assist with routing.
- In both IPv4 and IPv6, data is sent across the internet layer in packets called ***datagrams***.

The Internet Layer
IPv4 Datagram Format

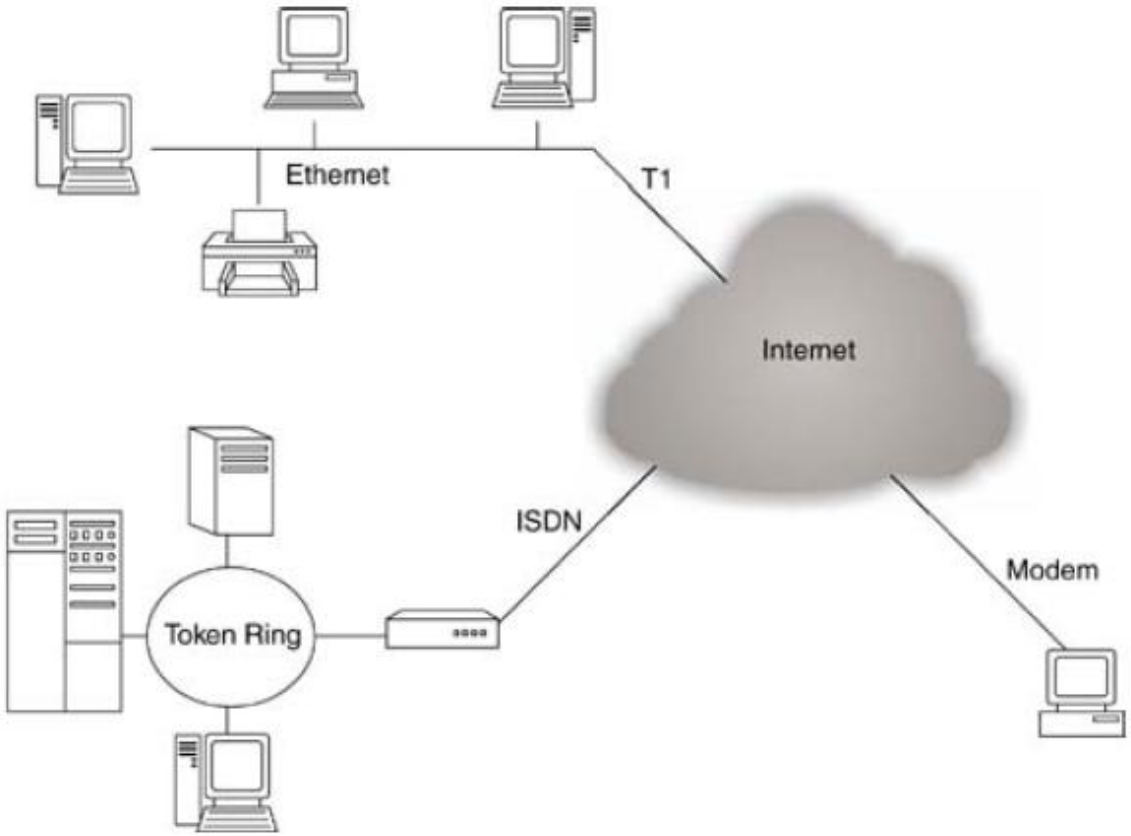


The Internet Layer
Data Transmission Using Packets

- Packets may take different routes to reach the destination depending on the routing approach and congestion level of the network.
- Mechanism to ensure that no packets are lost is available depending on the protocol used to send the data.

The Internet Layer (Cont.)

- The internet layer is responsible for connecting heterogenous networks to each other using homogeneous protocols.



The Transport Layer

- There are two primary protocols at this level:
- The Transmission Control Protocol (TCP):
  - A reliable protocol.
  - A high-overhead protocol that allows for retransmission of lost or corrupted data and delivery of bytes in the order they were sent.
- The User Datagram Protocol (UDP):
  - An unreliable protocol.
  - Does not guarantee that packets are delivered in the correct order (or at all).

The Application Layer

- The layer that delivers data to the user.
- The three lower layers all work together to define how data is transferred from one computer to another.
- The application layer decides what to do with the data after it's transferred.
  - For example, an application protocol like HTTP (for the World Wide Web) makes sure that your web browser displays a graphic image as a picture, not a long stream of numbers.

The Application Layer Protocols

Protocol	Purpose
HTTP	Web
SMTP, POP, IMAP	Email
FTP, FSP, TFTP	File Transfer
NFS	File Access
Gnutella, BitTorrent	File Sharing
SIP and Skype	Voice Communication

IP, TCP, and UDP

- ***IP*** was designed:
  - To allow multiple routes between any two points and to route packets of data around damaged routers.
  - To be open and platform-independent.
- Packets that make up a particular data stream may not all take the same route.
- Furthermore, they may not arrive in the order they were sent, if they even arrive at all.
- To improve on the basic scheme, ***TCP*** was layered on top of IP to:
  - Give each end of a connection the ability to acknowledge receipt of IP packets and request retransmission of lost or corrupted packets.
  - Allow the packets to be put back together on the receiving end in the same order they were sent.

IP, TCP, and UDP (Cont.)

- TCP, however, carries a fair amount of overhead.
- UDP is an unreliable protocol that does not guarantee that packets will arrive at their destination or that they will arrive in the same order they were sent.
- Although this would be a problem for uses such as file transfer, it is perfectly acceptable for applications where the loss of some data would go unnoticed by the end user.
  - For example, losing a few bits from a video or audio signal won't cause much degradation; it would be a bigger problem if you had to wait for a protocol like TCP to request a retransmission of missing data.
  - Furthermore, error-correcting codes can be built into UDP data streams at the application level to account for missing data.

IP, TCP, and UDP (Cont.)

- A number of other protocols can run on top of IP.
- The most commonly requested is ***ICMP***, the Internet Control Message Protocol, which uses raw IP datagrams to relay error messages between hosts.
  - The best-known use of this protocol is in the ping program.
  - Java does not support ICMP, nor does it allow the sending of raw IP datagrams.
- The only protocols Java supports are TCP and UDP, and application layer protocols built on top of these.
- All other transport layer, internet layer, and lower layer protocols such as ICMP, IGMP, ARP, RARP, RSVP, and others can only be implemented in Java programs by linking to native code.

IP Addresses and Domain Names

- As a Java programmer, you don't need to worry about the inner workings of IP, but you do need to know about ***addressing***.
- Every computer on an IPv4 network is identified by a unique four-byte address.
  - This is normally written in a *dotted quad format like 199.1.32.90,* where each of the four numbers is one unsigned byte ranging in value from 0 to 255.
- When data is transmitted across the network, the packet's header includes the address of the machine for which the packet is intended (the destination address) and the address of the machine that sent the packet (the source address).

IP Addresses and Domain Names (Cont.)

- Routers along the way choose the best route on which to send the packet by inspecting the destination address. The source address is included so the recipient will know who to reply to.
- A slow transition is under way to ***IPv6***, which will use 16-byte addresses.
  - This provides enough IP addresses to identify every person, every computer, and indeed every device on the planet.
  - IPv6 addresses are customarily written in eight blocks of four hexadecimal digits separated by colons, such as *FEDC:BA98:7654:3210:FEDC:BA98:7654:3210*

IP Addresses and Domain Names (Cont.)

- Although computers are very comfortable with numbers, human beings aren't very good at remembering them.
- Therefore, the ***Domain Name System (DNS)*** was developed to translate hostnames that humans can remember, such as "www.oreilly.com," into numeric Internet addresses such as 208.201.239.101.
- Some computers, especially servers, have fixed addresses.
- Others, especially clients on local area networks and wireless connections, receive a different address every time they boot up, often provided by a ***DHCP*** server.
- Mostly you just need to remember that IP addresses may change over time, and not write any code that relies on a system having the same IP address.

IP Addresses and Domain Names (Cont.)

- Some IPv4 addresses can be used on internal networks, but no host using addresses in these blocks is allowed onto the global Internet.
  - Addresses that begin with 10., 172.16. through 172.31. and 192.168.
  - These ***non-routable*** *addresses are useful for building private networks that can't be seen* on the Internet.
- IPv4 addresses beginning with 127 (most commonly 127.0.0.1) always mean the ***local loopback*** *address.*
  - The hostname for this address is often ***localhost****.*
  - *In IPv6, 0:0:0:0:0:0:0:1 (a.k.a. ::1) is the loopback address.*
- The IPv4 address that uses the same number for each of the four bytes (i.e.,255.255.255.255), is a broadcast address.
  - Packets sent to this address are received by all nodes on the local network, though they are not routed beyond the local network.

Ports

- Different types of traffic on a computer are sorted out using ***ports***.
- Each port is identified by a number between 1 and 65535.
- Port numbers between 1 and 1023 are reserved for well-known services like FTP, HTTP, and IMAP.

The Internet

- The world's largest IP-based network.
  - An amorphous group of computers in many different countries on all seven continents (Antarctica included) that talk to one another using IP protocols.
- Each computer on the Internet has at least one IP address by which it can be identified.
  - Many of them also have at least one name that maps to that IP address.

Internet Address Blocks

- Blocks of IPv4 addresses are assigned to ***Internet service providers (ISPs)*** by their regional Internet registry.
- When a company or an organization wants to set up an IP-based network connected to the Internet, their ISP assigns them a block of addresses.
- Each block has a fixed prefix.
  - For instance if the prefix is 216.254.85, then the local network can use addresses from 216.254.85.0 to 216.254.85.255.
  - Because this block fixes the first 24 bits, it's called a /24.
- Keep in mind that you have two fewer available addresses than you might first expect:
  - The lowest address in all block used to identify the network itself, and
  - The largest address is a broadcast address for the network.

Network Address Translation

- In NAT-based networks most nodes only have local, non-routable addresses selected from either :
  - 10.x.x.x,
  - 172.16.x.x to 172.31.x.x, or
  - 192.168.x.x.
- The routers that connect the local networks to the ISP translate these local addresses to a much smaller set of routable addresses.

Firewalls

- The hardware and software that sit between the Internet and the local network, checking all the data that comes in or out to make sure it's safe.
- The firewall can be:
  - part of the router that connects the local network to the broader Internet and may perform other tasks, such as network address translation.
  - a separate machine.
- Modern operating systems like Mac OS X and Red Hat Linux often have built-in personal firewalls that monitor just the traffic sent to that one machine.
- Either way, the firewall is responsible for inspecting each packet that passes into or out of its network interface and accepting it or rejecting it according to a set of rules.

Firewalls (Cont.)

- Filtering is usually based on network addresses and ports.
  - All traffic coming from the Class C network 193.28.25.x may be rejected because you had bad experiences with hackers from that network in the past.
  - Outgoing SSH connections may be allowed, but incoming SSH connections may not.
- More intelligent firewalls look at the contents of the packets to determine whether to accept or reject them.
- The exact configuration of a firewall—which packets of data are and to pass through and which are not—depends on the security needs of an individual site.

Firewalls (Cont.)

- The firewall is an excellent tool for network administrators but not for network developers.
  - Most corporate firewalls block direct UDP and TCP access.
  - Hence, developers must make a choice – either use standard Internet protocols and ignore users who work behind firewalls, or adapt software to proxy requests using protocols such a HTTP.

Firewalls (Cont.)

- The firewall is an excellent tool for network administrators but not for network developers.
  - Most corporate firewalls block direct UDP and TCP access.
  - Hence, developers must make a choice – either use standard Internet protocols and ignore users who work behind firewalls, or adapt software to proxy requests using protocols such a HTTP.

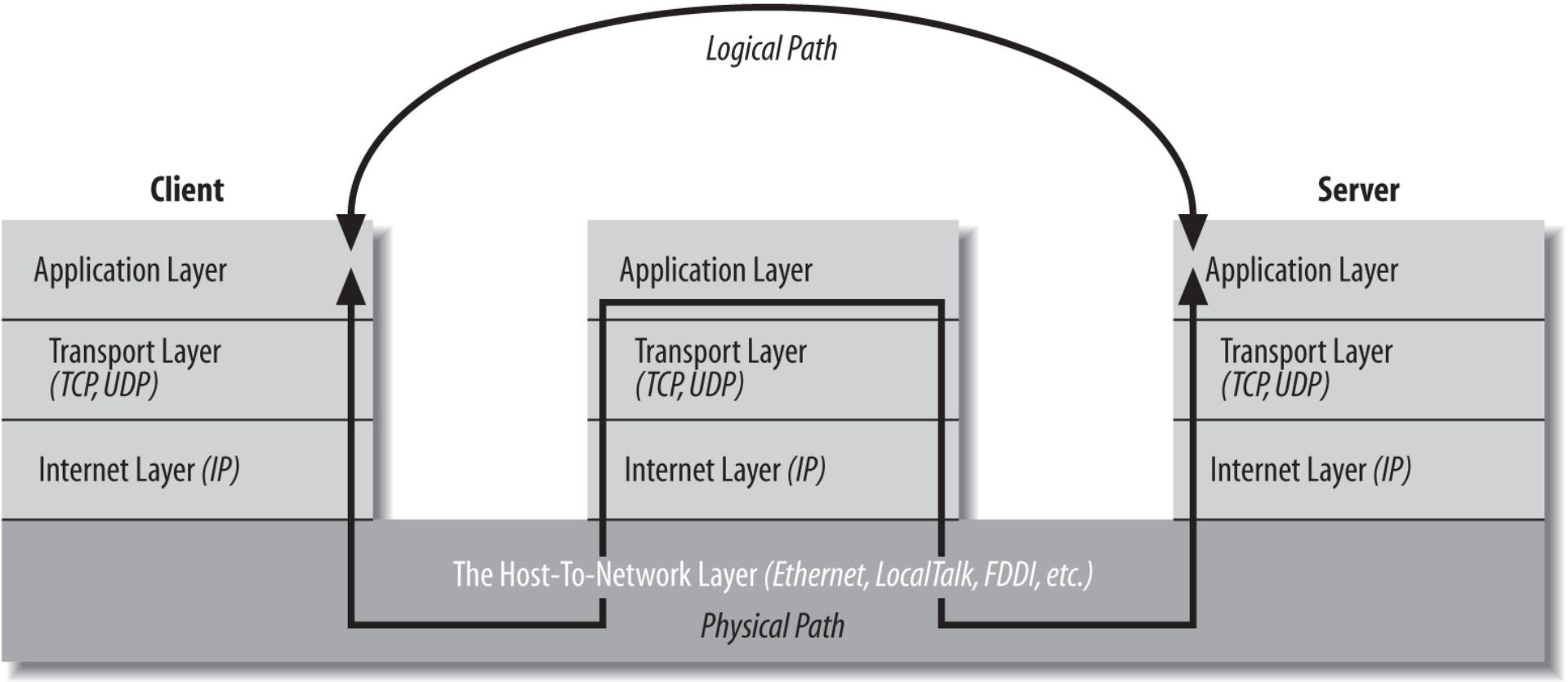
Proxy Servers

- If a firewall prevents hosts on a network from making direct connections to the outside world, a proxy server can act as a go-between.
  - Thus, a machine that is prevented from connecting to the external network by a firewall would make a request for a web page from the local proxy server instead of requesting the web page directly from the remote web server.
  - The proxy server would then request the page from the web server and forward the response back to the original requester.
- One of the security advantages of using a proxy server is that external hosts only find out about the proxy server.
  - They do not learn the names and IP addresses of the internal machines, making it more difficult to hack into internal systems.

Proxy Servers (Cont.)

- Whereas firewalls generally operate at the level of the transport or internet layer, proxy servers normally operate at the application layer.
- A proxy server has a detailed understanding of some application-level protocols, such as HTTP and FTP.
- Packets that pass through the proxy server can be examined to ensure that they contain data appropriate for their type.
  - For instance, FTP packets that seem to contain Telnet data can be rejected.

Proxy Servers (Cont.)



Proxy Servers (Cont.)

- Proxy servers can also be used to implement local caching.
- When a file is requested from a web server, the proxy server first checks to see if the file is in its cache.
  - If the file is in the cache, the proxy serves the file from the cache rather than from the Internet.
  - If the file is not in the cache, the proxy server retrieves the file, forwards it to the requester, and stores it in the cache for the next time it is requested

References

Chapter 1 of *Java™ Network Programming and Distributed Computing*, David Reilly and Michael Reilly.

Chapter 1 of *Java Network Programming*, Elliotte Rusty Harold, O'Reilly, Fourth Edition, 2013.

Networks and Internet Programming

Internet Addressing

Eng. Asma Abdel Karim
Computer Engineering Department

Outline

- Local Area Network Addresses.
- Internet Protocol Addresses.
- The Domain Name System.
- Internet Addressing with Java.

Local Area Network Addresses

- Devices connected to a LAN have their own unique physical or hardware address.
  - This address is useful only in the context of a LAN.
- Java network programmers do not need to be concerned with the details of how data is routed within a LAN.
  - Java does not provide access to the lower-level data link protocols used by LANs.
  - No matter what type of LAN is used, software can be written for it in Java providing it supports TCP/IP.

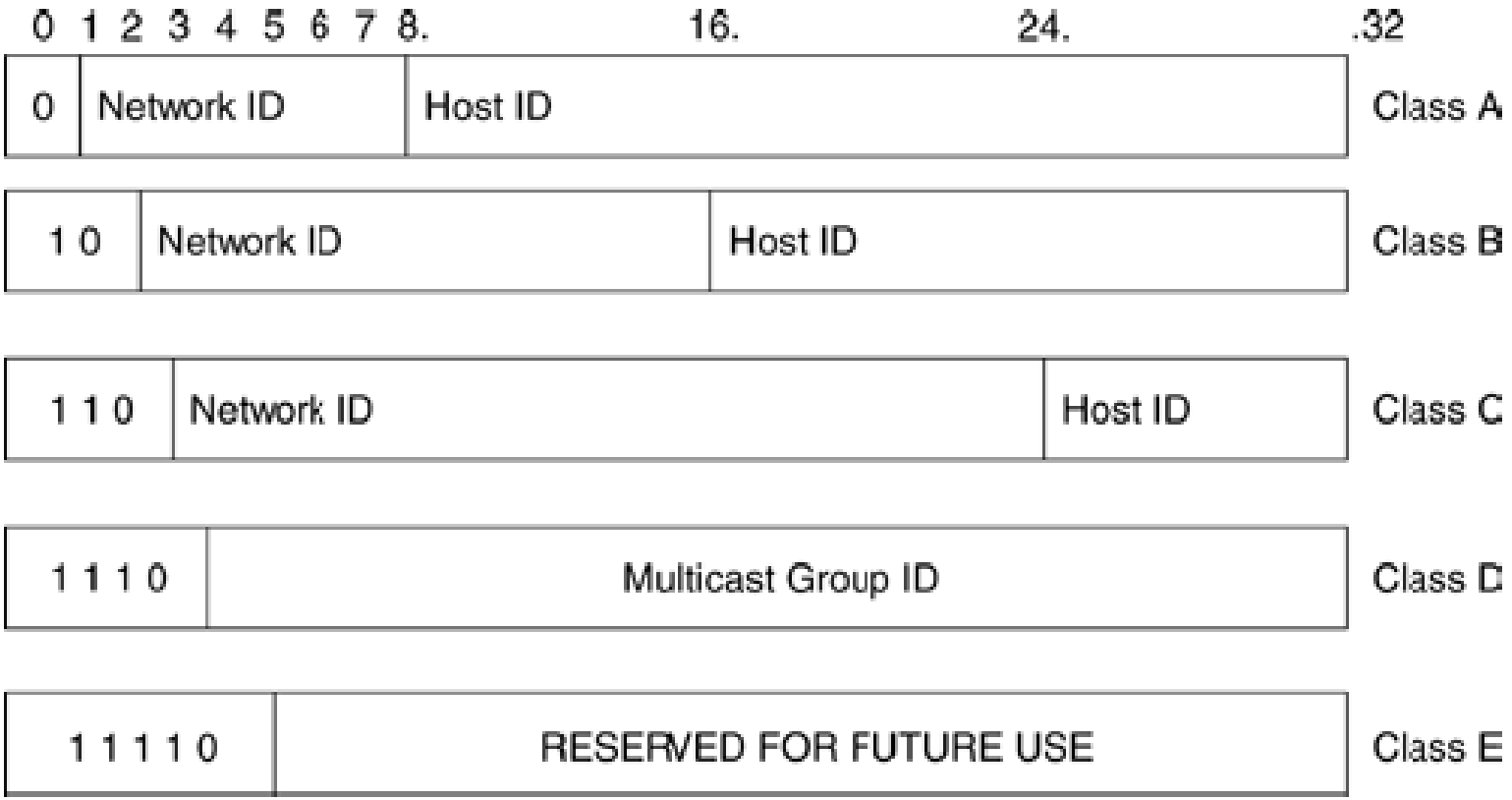
Internet Protocol Addresses

- Devices having a direct internet connection are allocated a ***unique*** IP address.
  - This address is used by the internet protocol to route IP datagrams to the correct location.
- IP addresses may be allocated:
  - ***Statically***: IP address is bounded permanently to certain machine.
  - ***Dynamically***: IP address is leased to a particular machine for a certain period.
    - For example in the case of ISP that offers a pool of modems for dial-up connections.
    - Used when many devices require Internet access for limited periods of time.
    - The Dynamic Host Control Protocol (DHCP) provides addresses on demand from a pool of addresses.

Structure of the IP address

- Under IPv4, the IP address is a 32-bit number made up of four octets (bytes).
- IP addresses are written in dotted decimal notation (e.g. 127.0.0.1).
  - Each byte is an unsigned integer between 0 and 255.
- Each IP address consists of two components:
  - The ***network*** address: a unique identifier of a specific network.
  - The ***host*** address: a unique address of the host in the network it belongs to.

Structure of the IP address (Cont.)



Special IP Addresses

- 127.0.0.1 is a special reserved address known as the ***loopback*** or ***localhost*** address.
- The loopback address is very useful when programming and debugging network software.
  - Programmers often want to connect to the local machine for testing purposes regardless of whether a connection to the internet exists or not.

Special IP Addresses (Cont.)

- The Internet Assigned Number Authority (IANA) has reserved three sets of addresses for use within a local ***intranet*** environment.
- On the internet, routers will never forward data using these addresses, so they can be safely used locally.

Type	Address Range
Class A	10.0.0.0 – 10.255.255.255
Class B	172.16.0.0 – 172.31.255.255
Class C	192.168.0.0 – 192.168.255.255

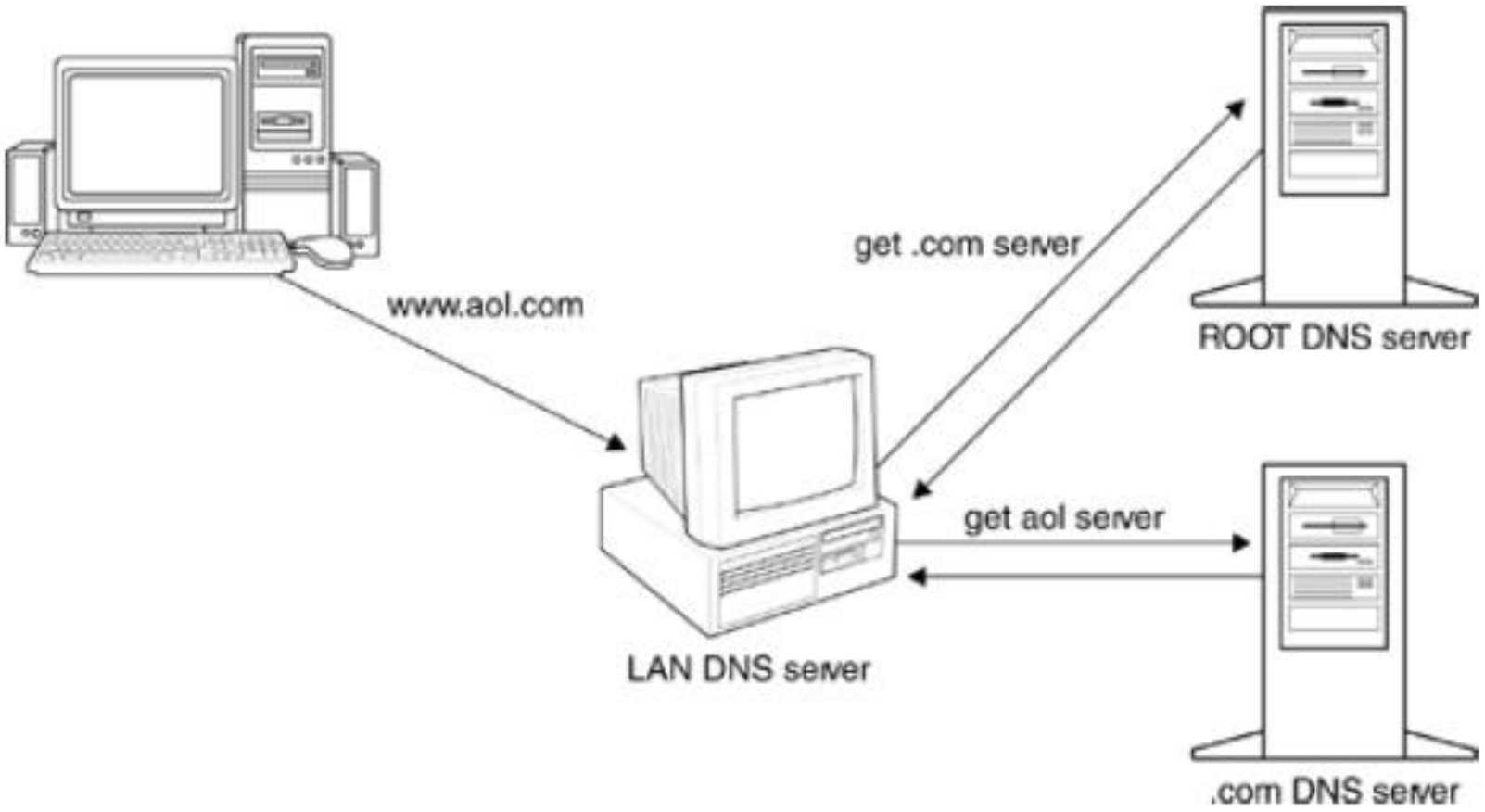
The Domain Name System

- Memorizing IP addresses is an impossible task.
- The ***Domain Name System (DNS)*** makes the internet user-friendly, by associating a textual name with an IP address.
- An entity can apply for a domain name, which can be used by people to locate that entity on the internet.

The Domain Name System (Cont.)

- Given the vast number of machines connected to the internet, the number of domain-name-to-IP-address mappings is too great for any one system to handle.
- The DNS is a more sophisticated and robust system.
  - It can be thought of as a distributed database.
  - Consists of a hierarchical structure which is broken up by the type of address (.net, .com, .gov, .edu, ….) or by the country (.au, .uk, …..).

The Domain Name System (Cont.)



Internet Addressing with Java

- A host on the internet can be represented by either:
  - A dotted decimal format as an IP address, or
  - A hostname such as www.aol.com.
- Under Java, such addresses are represented by the ***java.net.InetAddress*** class.
  - There are no public constructors for this class. Arbitrary addresses may not be created.
  - Instead, there are static methods that return ***InetAddress*** instances.

Methods to Create InetAddress Objects (1)

public static InetAddress ***getByName*** (String host) throws UnknownHostException, SecurityException

public static InetAddress[ ] ***getAllByName***(String host) throws UnknownHostException, SecurityException

public static InetAddress ***getLocalHost()*** throws UnknownHostException, SecurityException

Methods to Create InetAddress Objects (2)

Java 1.4 adds two more factory methods that do not check their addresses with the local DNS server.

- The first creates an InetAddress object with an IP address and no hostname.

  public static InetAddress ***getByAddress*** (byte[ ] address) throws UnknownHostException

- The second creates an InetAddress object with an IP address and a hostname.

  public static InetAddress ***getByAddress*** (String hostName, byte[] address) throws UnknownHostException

Getter Methods

- The following method returns the hotsname of the InetAddress object:

  public String ***getHostName*** () throws SecurityManager
- The following method returns the IP address of the InetAddress object in byte format. The bytes are returned in network byte order, with the highest byte as byteArray[0]:

  public byte [] ***getAddress*** ()
- The following method returns the IP address of the InetAddress in dotted decimal format:

  public String ***getHostAddress*** ()

Object Class Inherited Methods

- public boolean ***equals*** (Object o)
- public int ***hashCode*** ()
- public String ***toString*** ()

Example1: Using InetAddress to Determine Localhost Address

```
import java.net.*;

public class LocalHostDemo {

   public static void main(String[] args) {
      System.out.println("Looking up local host!");
      try {
         InetAddress localAddress = InetAddress.getLocalHost();
         System.out.println("IP address: "+localAddress.getHostAddress());
      }
      catch (UnknownHostException uhe){
        System.out.println("Error - unable to resolve localhost");
      }
   }
}
```

Example2: Using InetAddress to Find Out About Other Addresses

```
import java.net.*;

public class NetworkResolverDemo {

   public static void main(String[] args) {
      if (args.length!=1){
         System.err.println("Syntax - NetworkResolverDemo host");
         System.exit(0);
      }
      System.out.println("Resolving "+args[0]);

      try{
         InetAddress addr = InetAddress.getByName(args[0]);
         System.out.println(addr);
      }
      catch (UnknownHostException uhe){
         System.out.println("Error - unable to resolve host name");
      }
   }

}
```

Address Types Methods

- public boolean *isAnyLocalAddress*()
- public boolean *isLoopbackAddress*()
- public boolean *isLinkLocalAddress*()
- public boolean *isSiteLocalAddress*()
- public boolean *isMulticastAddress*()
- public boolean *isMCGlobal*()
- public boolean *isMCNodeLocal*()
- public boolean *isMCLinkLocal*()
- public boolean *isMCSiteLocal*()
- public boolean *isMCOrgLocal*()

Testing Reachability

- public boolean ***isReachable*** (int timeout) throws IOException
- public boolean ***isReachable*** (NetworkInterface interface, int ttl, int timeout) throws IOException

Inet4Address and Inet6Address

- Public final class ***Inet4Address*** extends InetAddress

- Public final class ***Inet6Address*** extends InetAddress

The NetworkInterface Class

- The NetworkInterface class represents a local IP address. This can be:
  - A physical interface such as an additional Ethernet card, or
  - A virtual interface bound to the same physical hardware.
- The NetworkInterface class provides methods to enumerate all the local addresses, regardless of interface, and to create InetAddress objects from them.

Methods to Create NetworkInterface Objects

- By name:

public static NetworkInterface ***getByName*** (String name) throws SocketException

- By IP address:

public static NetworkInterface ***getByInetAddress*** (InetAddress address) throws SocketException

- By enumeration:

public static Enumeration ***getNetworkInterfaces***() throws SocketException

Getter Methods

- The following method returns a ***java.util.Enumeration*** containing an InetAddress object for each IP address the interface is bound to:

  public Enumeration ***getInetAddresses***()
- The following method returns the name of a particular NetworkInterface object, such as eth0 or lo:

  public String ***getName***()
- The following method returns a more human-friendly name for the particular NetworkInterface — something like “Ethernet Card 0.”:

  public String ***getDisplayName***()

References

Chapter 3 of *Java™ Network Programming and Distributed Computing*, David Reilly and Michael Reilly.

Chapter 4 of Java Network Programming, Elliotte Rusty Harold, O'Reilly, Fourth Edition, 2013.

Networks and Internet Programming

Data Streams

Part-1

Eng. Asma Abdel Karim
Computer Engineering Department

Outline

- Overview.
- How Streams Work?
- Low-level Streams.
- Filter Streams.

Overview

- Communication over networks, with files, and even between applications, is represented in Java by ***Streams***.
- Stream-based communication is central to almost any type of Java application.
- Almost all network communication (except UDP communication) is conducted over streams.

What Exactly are Streams?

- Byte-level communication is represented in Java by data streams.
- Data streams are conduits through which information is sent and received.

What Exactly are Streams? (Cont.)

- When designing a system, the correct stream must be selected.
  - The type of stream used is not important, as a consistent interface is provided.
- Streams may be chained together, to provide an easier and more manageable interface.
  - If for example, data needed to be processed in a particular way, a second stream could connect to an existing stream, to provide for processing of the data.

What Exactly are Streams? (Cont.)

- Streams are divided into two categories:
  - ***Input streams*** that may be read from.
  - ***Output streams*** that may be written to.
- Although streams are usually one-way, multiple streams can be used together for two-way communication.

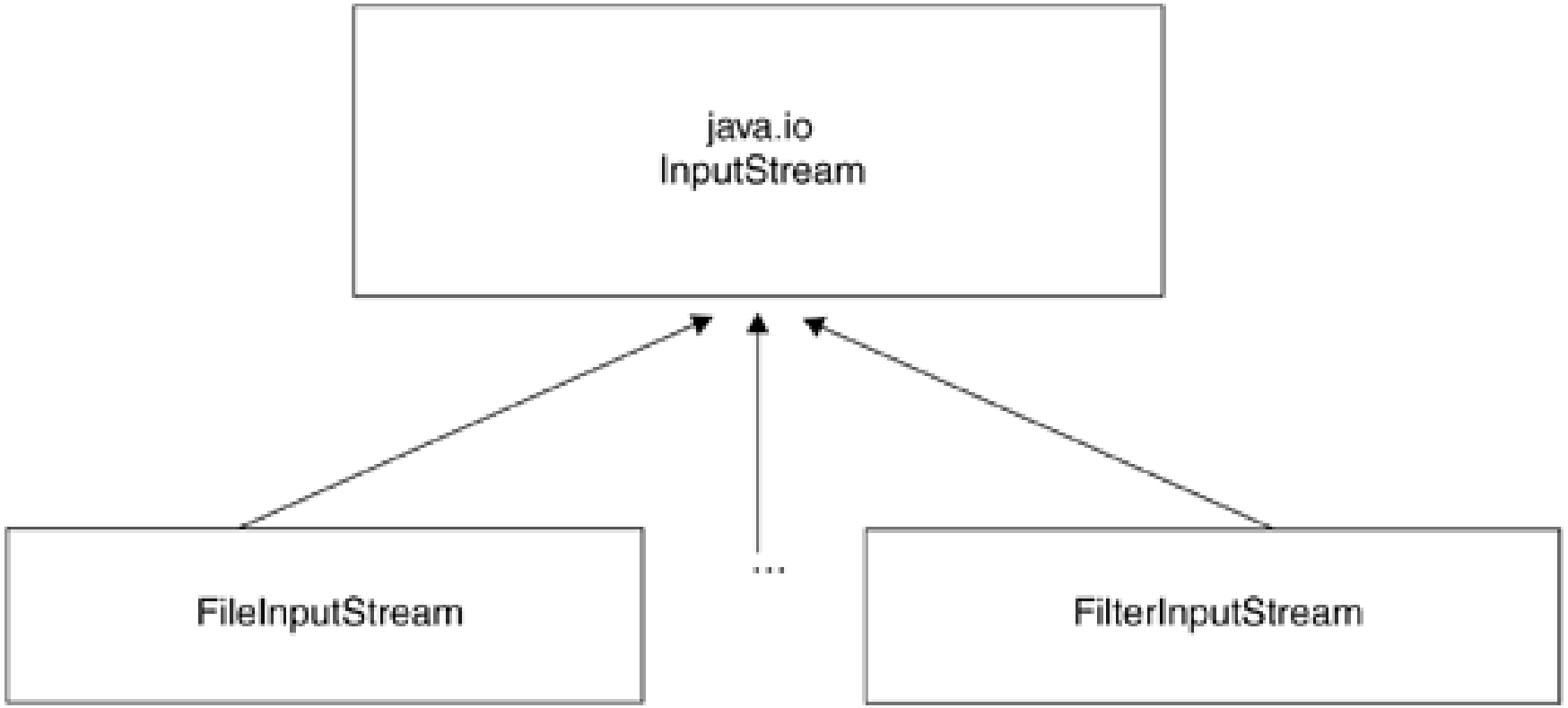
What Exactly are Streams? (Cont.)

- In Java, streams take a flexible, one-size-fits-all approach.
  - They are fairly interchangeable, and can be applied on top of another stream, or even several other streams.
- You can attach any filter stream to any low level stream (i.e. file or network stream).
  - This can be safely done, as long as you don't try to write to an input stream or read from an output stream.
  - A filter stream is a stream that filters data in some fashion.

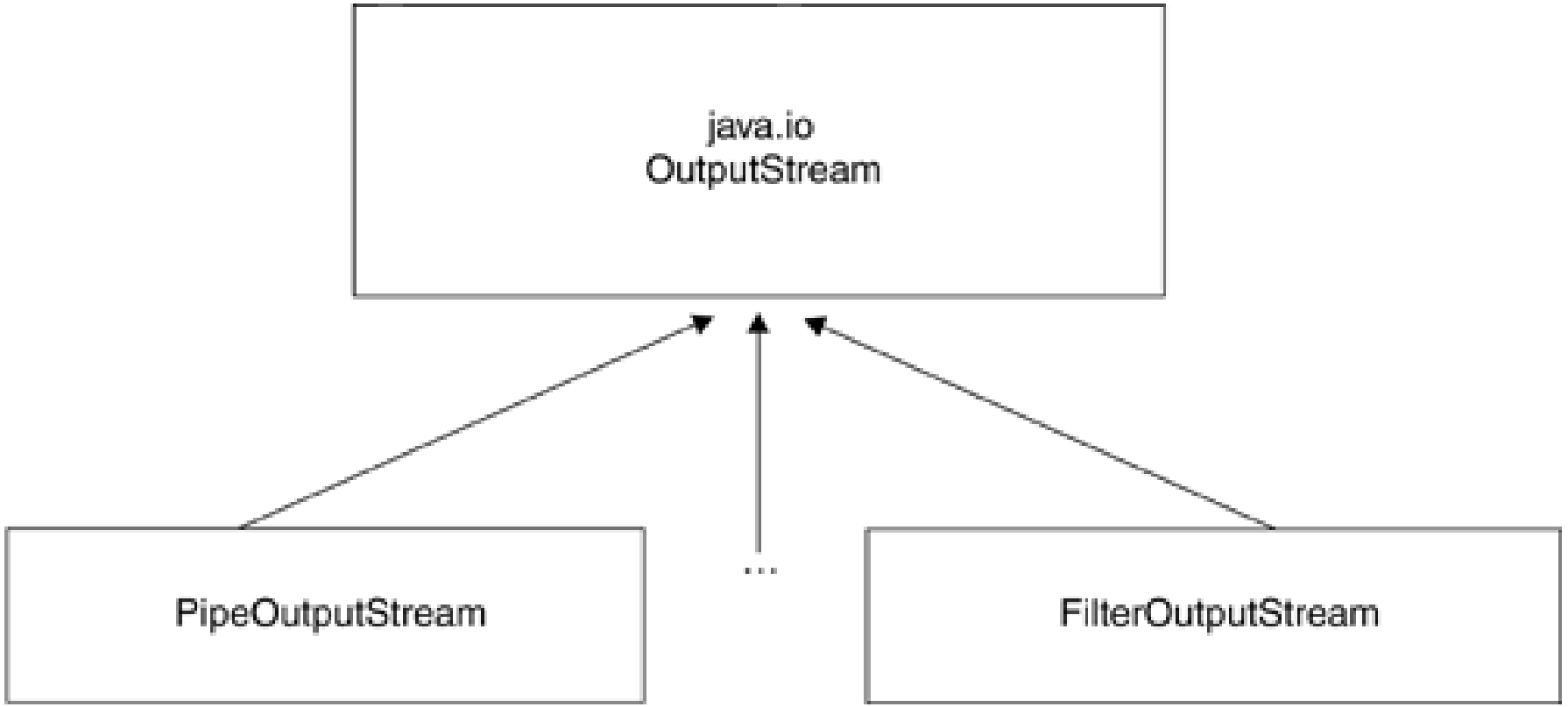
How Streams Work?

- Streams for reading inherit from a common superclass, the ***java.io.InputStream*** class.
- Streams for writing inherit from a common superclass, the ***java.io.OutputStream*** class.
- These are abstract classes; they cannot be instantiated.
  - Instead, an appropriate subclass for the task in hand is instantiated.

How Streams Work? (Cont.)



How Streams Work? (Cont.)



Reading from an Input Stream

- Choosing the right low-level input stream is a fairly straightforward task.
  - The name of the stream matches the data source it will read from.
- There are six low-level input streams to choose from, each of which performs an entirely different task.
- There are other low-level streams that are not directly instantiated by developers
  - These are returned by invoking a method of a networking object.

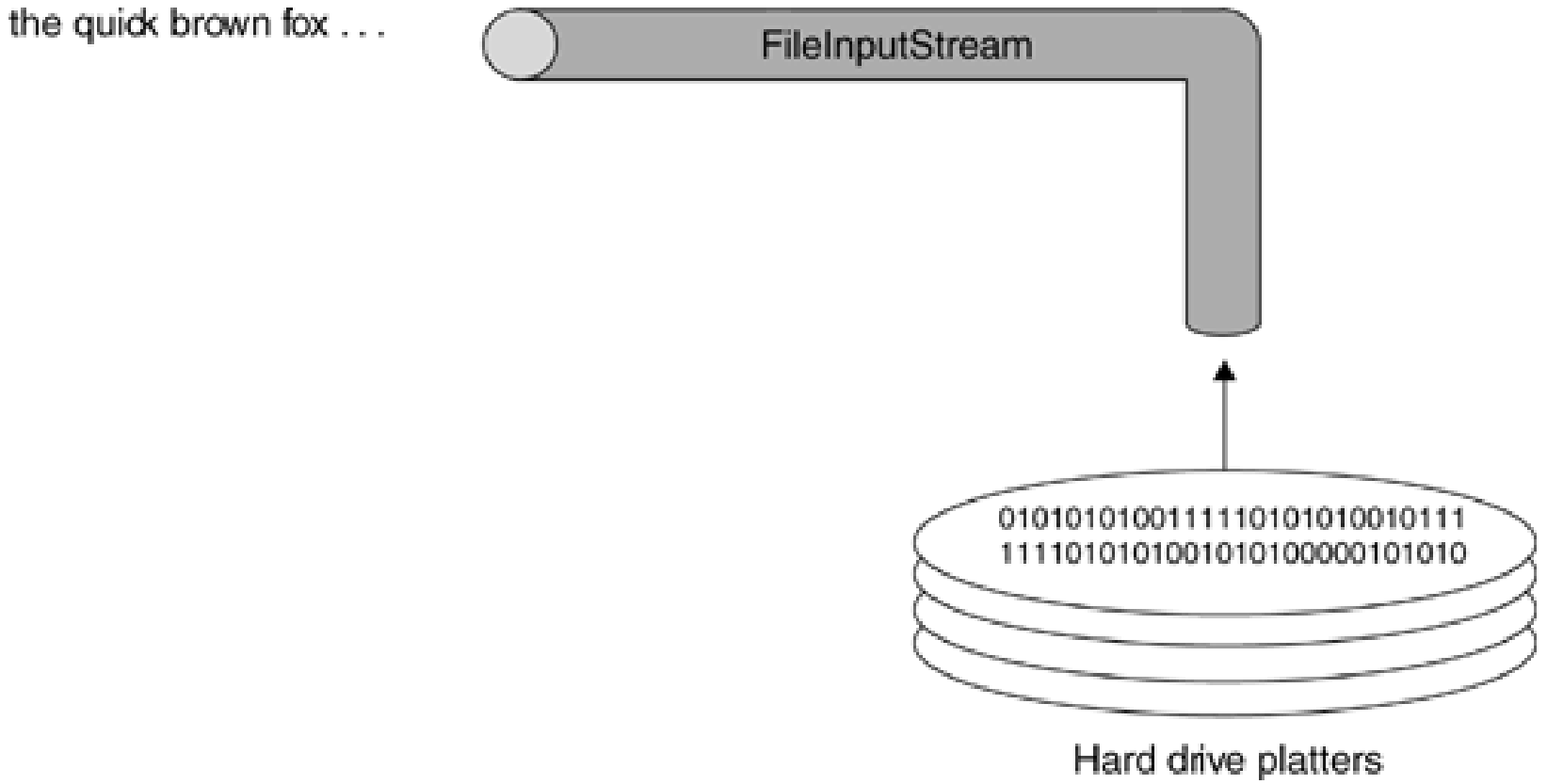
Reading from an Input Streams (Cont.)

Low-level Input Stream	Purpose of Stream
ByteArrayInputStream	Reads bytes of data from an in-memory array.
FileInputStream	Reads bytes of data from a file on the local file system.
PipedInputStream	Reads bytes of data from a thread pipe.
StringBufferInputStream	Reads bytes of data from a string.
SequenceInputStream	Reads bytes of data from two or more low-level streams, switching from one stream to the next when the end of the stream is reached.
System.in	Reads bytes of data from the user console.

Reading from an Input Stream (Cont.)

- When a low-level input stream is created, it will read from a source of information that supplies it with data.
- Inputs streams act as consumers of information.
  - Bytes are read from the source sequentially.
  - Once bytes have been read, you can't go back and read them again.
  - Bytes haven't been erased, the stream has simply moved on to the next byte of information.

Reading from an Input Stream (Cont.)



Reading from an Input Stream (Cont.)

- Input streams use blocking I/O.
- ***Blocking I/O*** is a term applied to any form of input or output that does not immediately return from an operation.
- Blocking I/O may cause performance problems.
  - This can be alleviated by using ***data buffering***.

The java.io.InputStream Class

- The abstract InputStream class defines methods common to all input streams and all of them are public:
  - int **available** ( ) throws java.io.IOException

  Returns the number of bytes currently available for reading. More bytes may be available in the future, but reading more than the number of available bytes will result in a read that will block indefinitely.

  - void **close** ( ) throws java.io.IOException

  Closes the input stream and frees any resources (such as file handles or file locks) associated with the input stream.

The java.io.InputStream Class (Cont.)

- The abstract InputStream class defines methods common to all input streams and all of them are public:
  - void **mark** (int readLimit)
  1. *Records the current position in the input stream, to allow an input stream to revisit the same sequence of bytes at a later point in the future, by invoking the InputStream.reset() method.*
  2. *Not every input stream will support this functionality.*
  - boolean **markSupported** ( )
  1. *Returns "true" if an input stream supports the mark() and reset() methods, "false" if it does not.*
  2. *Unless over ridden by a subclass of InputStream, the default value returned is false.*

The java.io.InputStream Class (Cont.)

- The abstract InputStream class defines methods common to all input streams and all of them are public:
  - int **read**( ) throws java.io.IOException
  1. *Returns the next byte of data from the stream.*
  2. *Subclasses of InputStream usually override this method to provide custom functionality (such as reading from a file or a string).*
  3. *As mentioned earlier, input streams use blocking I/O, and will block indefinitely if no further bytes are yet available.*
  4. *When the end of the stream is reached, a value of –1 is returned.*

The java.io.InputStream Class (Cont.)

- The abstract InputStream class defines methods common to all input streams and all of them are public:
  - int **read** (byte[ ] byteArray) throws java.io.IOException
  1. *Reads a sequence of bytes and places them in the specified byte array, by calling the read() method repeatedly until the array is filled or no more data can be obtained.*
  2. *This method returns the number of bytes successfully read, or –1 if the end of the stream has been reached.*

The java.io.InputStream Class (Cont.)

- The abstract InputStream class defines methods common to all input streams and all of them are public:
  - int **read** (byte [ ] byteArray, int offset, int length) throws java.io.IOException, java.lang.IndexOutOfBoundsException
    1. *Reads a sequence of bytes, placing them in the specified array.*
    2. *Unlike the previous method, read(byte[] byteArray), this method begins stuffing bytes into the array at the specified offset, and for the specified length, if possible. This allows developers to fill up only part of an array.*
    3. *Developers should be mindful that at runtime, out-of-bounds exceptions may be thrown if the array size, offset, and length exceed array capacity.*

The java.io.InputStream Class (Cont.)

- The abstract InputStream class defines methods common to all input streams and all of them are public:
  - void **reset**( ) throws java.io.IOException
  1. *Moves the position of the input stream back to a preset mark, determined by the point in time when the mark() method was invoked.*
  2. *Few input streams support this functionality, and may cause an IOException to be thrown if called.*

The java.io.InputStream Class (Cont.)

- The abstract InputStream class defines methods common to all input streams and all of them are public:
  - long **skip** (long amount) throws java.io.IOException
  1. *Reads, but ignores, the specified amount of bytes.*
  2. *These bytes are discarded, and the position of the input stream is updated.*
  3. *Though unlikely, it is entirely possible that the specified number of bytes could not be skipped (for example, as stated in the Java API, if the end of the stream is reached).*
  4. *The skip method returns the number of bytes skipped over, which may be less than the requested amount.*

Example Using a Low-level Input Stream

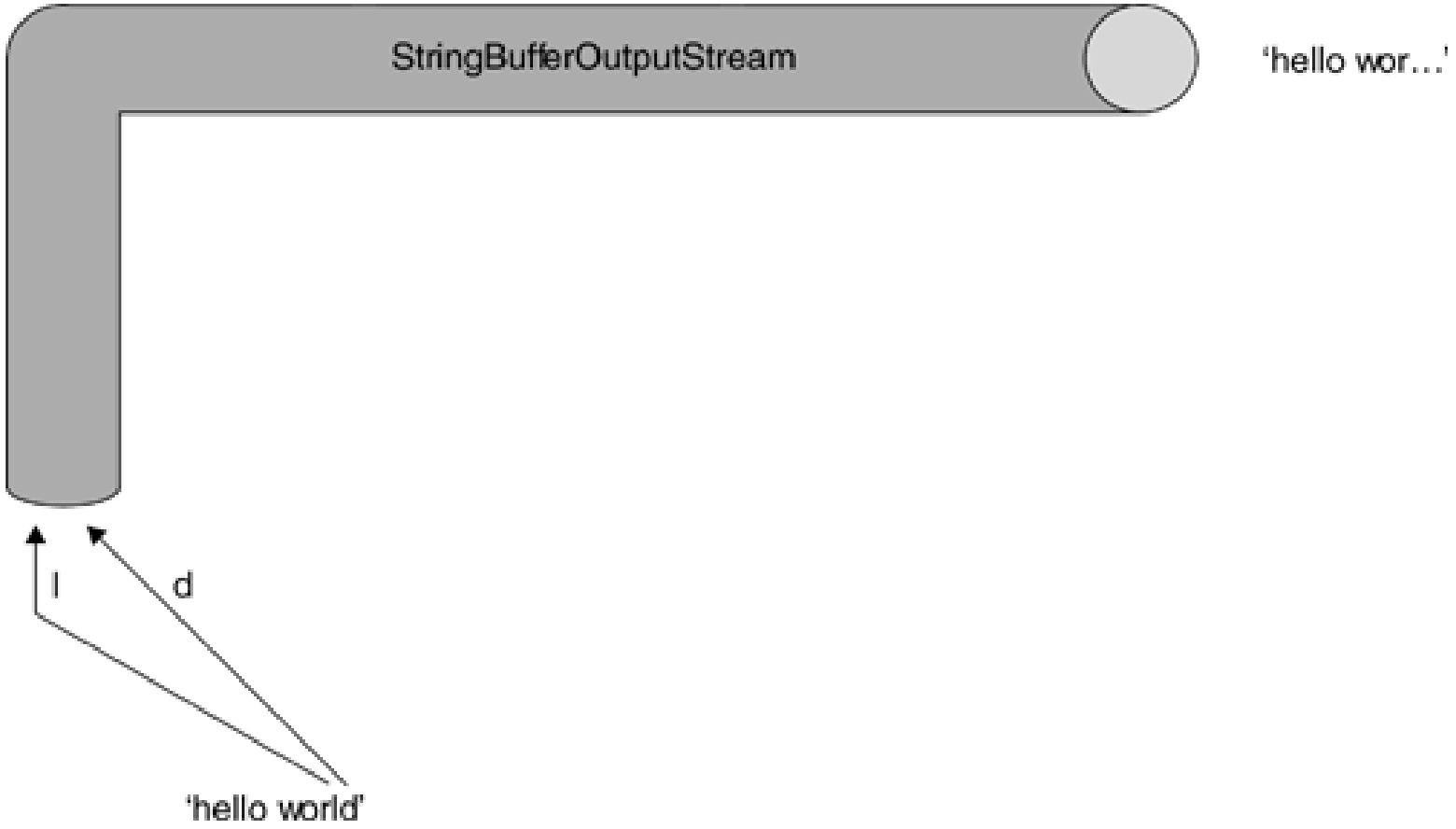
```
import java.io.*;

public class FileInputStreamDemo {
   public static void main(String[] args) {
      if (args.length != 1){
         System.err.println ("Syntax - FileInputStreamDemofile");
         return;
      }
      try{
         InputStream fileInput = new FileInputStream( args[0] );
         int data = fileInput.read();
         while (data != -1){
            System.out.write ( data );
            data = fileInput.read();
         }
         fileInput.close();
      }
      catch (IOException ioe){
         System.err.println ("I/O error - " + ioe);
      }
   }
}
```

Writing to an Output Stream

- While an input stream is a data consumer, an output stream is a data producer.
  - It literally creates bytes of information and transmits them to something else (such as a file or data structure or network connection).
- Like input streams, data is communicated sequentially; that is, the first byte in will be the first byte out.
  - This approach is analogous to a FIFO queue.
- Unlike some specialized filter input streams, which allow you to "go back *n*" bytes within a sequence, once data is sent to an output stream it cannot be undone.

Writing to an Output Stream (Cont.)



Writing to an Output Stream (Cont.)

- A number of output streams are available in the java.io package for a variety of tasks.
  - Such as writing to data structures including strings and arrays, or to files or communication pipes.
- There are six important low-level output streams that may be written to.
  - In addition to filter streams that may be connected to these low-level streams
- As mentioned earlier, there are other streams which may be written to that developers cannot create and instantiate directly.

Writing to an Output Stream (Cont.)

Low-level Output Stream	Purpose of Stream
ByteArrayOutputStream	Writes bytes of data to an array of bytes.
FileOutputStream	Writes bytes of data to a local file.
PipedOutputStream	Writes bytes of data to a communications pipe, which will be connected to a *java.io.PipedInputStream.*
StringBufferOutputStream	Writes bytes to a string buffer (a substitute data structure for the fixed-length string).
System.err	Writes bytes of data to the error stream of the user console, also known as standard error. In addition, this stream is cast to a *PrintStream*.
System.out	Writes bytes of data to the user console, also known as standard output. In addition, this stream is cast to a *PrintStream*.

Writing to an Output Stream (Cont.)

- Bytes may be sent one at a time or as part of an array.
  - However, when bytes are read one at a time, individual byte writes may affect system performance.
- Reading information can block indefinitely, but writing information may also block for small amounts of time.
  - This is not normally as significant an issue as the case of blocking read operations, as the bytes are ready to send.

The java.io.OutputStream Class

The abstract class java.io.OutputStream defines the following public methods:

- void **close**() throws java.io.IOException

1. *Closes the output stream, notifying the other side that the stream has ended.*
2. *Pending data that has not yet been sent will be sent, but no more data will be delivered.*

The java.io.OutputStream Class (Cont.)

The abstract class java.io.OutputStream defines the following public methods:

- void **flush**() throws java.io.IOException

1. *Performs a "flush" of any unsent data and sends it to the recipient of the output stream.*
2. *To improve performance, streams will often be buffered, so data remains unsent. This is useful at times, but obstructive at others.*
3. *The method is particularly important for OutputStream subclasses that represent network operations, as flushing should always occur after a request or response is sent so that the remote side isn't left waiting for data.*

The java.io.OutputStream Class (Cont.)

The abstract class java.io.OutputStream defines the following public methods:

- void **write** (int byte) throws java.io.IOException

1. *Writes the specified byte.*
2. *This is an abstract method, overridden by OutputStream subclasses.*

- void **write** (byte[] byteArray) throws java.io.IOException

1. *Writes the contents of the byte array to the output stream.*
2. *The entire contents of the array (barring any error) will be written.*

The java.io.OutputStream Class (Cont.)

The abstract class java.io.OutputStream defines the following public methods:

- void **write** (byte[] byteArray, int offset, int length) throws java.io.IOException

1. *Writes the contents of a subset of the byte array to the output stream.*
2. *This method allows developers to specify just how much of an array is sent, and which part, as opposed to the OutputStream.write(byte[] byteArray) method, which sends the entire contents of an array.*

Example Using a Low-level Output Stream

```
import java.io.*;

public class FileOutputStreamDemo{
   public static void main(String args[]){
      if (args.length != 2){
         System.err.println("Syntax - FileOutputStreamDemo src dest");
         return;
      }
      String source = args[0];
      String destination = args[1];
      try {
         InputStream input = new FileInputStream( source );
         System.out.println ("Opened " +source + " for reading.");
         OutputStream output = new FileOutputStream( destination );
         System.out.println ("Opened " +destination + " for writing.");
```

Example Using a Low-level Output Stream (Cont.)

```
            int data = input.read();
            while ( data != -1){
                output.write (data);
                data=input.read();
            }
            input.close();
            output.close();
            System.out.println ("I/O streams closed");
        }
        catch (IOException ioe){
            System.err.println ("I/O error - " + ioe);
        }
    }
}
```

Filter Streams

- While the basic low-level streams provide a simple mechanism to read and write bytes of information, their flexibility is limited.
- After all, reading bytes is complex.
  - There's more to the world that just bytes of data.
  - Text, for example, is a sequence of characters, and other forms of data like numbers take up more than a single byte.
- Byte-level communication can also be inefficient.
  - Data buffering can improve performance.
- To overcome these limitations, filter streams are used.

Filter Streams (Cont.)

- Filter streams add additional functionality to an existing stream.
  - By processing data in some form, such as buffering for performance.
  - By offering additional methods that allow data to be accessed in a different manner. For example, reading a line of text rather than a sequence of bytes.
- Filters make life easier for programmers.
  - As they can work with familiar constructs such as strings, lines of text, and numbers, rather than individual bytes.
  - Instead of the programmer writing a string one character at a time and converting each character to an int value for the OutputStream.write(int) method, the filter stream does this for them.

Connecting a Filter Stream to an Existing Stream

- Filter streams can be connected to any other stream.
  - To a low-level stream or even another filter stream.
- Filter streams are extended from the *java.io. FilterInputStream* and *java.io.FilterOutputStream* classes.
- Each filter stream supports one or more constructors That accept:
  - Either an InputStream, in the case of an input filter, or
  - An OutputStream, in the case of an output filter.
- Connecting a filter stream is as simple as:
  - Creating a new instance of a filter passing an instance of an existing stream, and
  - Using the filter from then on to read or write.

Connecting a Filter Stream to an Existing Stream (Cont.)

- The following code connects a *PrintStream* (used to print text to an *OutputStream* subclass) to a stream that wrote to a file and uses the filter stream to write a message on the file.

```
FileOutputStream fout = new FileOutputStream ( somefile );
PrintStream pout = new PrintStream (fout);
pout.println ("hello world");
```

Connecting a Filter Stream to an Existing Stream (Cont.)

- The process is fairly simple as long as the programmer remembers two things:

1. Read and write operations must take place on the new filter stream.
2. Read and write operations on the underlying stream can still take place, but not at the same time as an operation on the filter stream.

Useful Filter Input Streams *BufferedInputStream* Class

- The purpose of I/O buffering is to improve system performance.
- Rather than reading a byte at a time, a large number of bytes are read together the first time the read() method is invoked.
- When an attempt is made to read subsequent bytes, they are taken from the buffer, not the underlying input stream.
  - This improves data access time and can reduce the number of times an application blocks for input.

Useful Filter Input Streams *BufferedInputStream* Class (Cont.)

- Constructors:
  - BufferedInputStream (InputStream input)

  Creates a buffered stream that will read from the specified InputStream object.
  - BufferedInputStream (InputStream input, int bufferSize) throws java.lang.IllegalArgumentException

  1. *Creates a buffered stream, of the specified size, which reads from the InputStream object passed as a parameter.*
  2. *This allows developers to specify a size, which can improve efficiency if large amounts of data are going to be read. The buffer size specified must be greater than or equal to one.*

Useful Filter Input Streams *BufferedInputStream* Class (Cont.)

- Methods:
  - No additional methods are provided by the BufferedInputStream class.
  - However, it does override the markSupported() method, indicating that it supports the mark(int) and reset() methods.

Useful Filter Input Streams
DataInputStream Class

- A frequent task in any programming language is reading and writing primitive data types such as numbers and characters.
  - These information types are not easily represented as bytes (for example, some data types take up more than one byte of information).
- Developers should not be concerned with the way in which representation occurs.
  - Instead, the data types can be read simply, by invoking methods of the DataInputStream class, which handles the translation automatically.
- This class implements the java.io. DataInput interface.

Useful Filter Input Streams *DataInputStream* Class (Cont.)

- Constructors:
  - DataInputStream (InputStream input)

  Creates a data input stream, reading from the specified input stream.

Useful Filter Input Streams *DataInputStream* Class (Cont.)

- Methods:

Many methods are added to the DataInputStream class, in order to facilitate access to new data types.

- Boolean **readBoolean**( ) throws java.io.EOFException java.io
- Byte readByte( ) throws java.io.EOFException java.io.IOException
- char readChar( ) throws java.io.EOFException java.io.IOException
- double readDouble( ) throws java.io.EOFException java.io.IOException
- float readFloat( ) throws java.io.EOFException java.io.IOException
- void readFully(byte[ ] byteArray) throws java.io.EOFException java.io.IOException
- void readFully(byte[] byteArray, int offset, int length) throws java.io.EOFException java.io.IOException

Useful Filter Input Streams *DataInputStream* Class (Cont.)

- Methods:
- *float readInt( )* throws java.io.EOFException java.io.IOException
- *string readLine( )* throws java.io.IOException
- *long readLong( )* throws java.io.EOFException java.io.IOException
- *short readShort( )* throws java.io.EOFException java.io.IOException
- *int readUnsignedByte( )* throws java.io.EOFException java.io.IOException
- *int readUnsignedShort( )* throws java.io.EOFException java.io.IOException
- *String readUTF( )* throws java.io.EOFException java.io.IOException
- *Static String readUTF(DataInputStream input)* throws java.io.EOFException java.io.IOException
- *int skipBytes(int number)* throws java.io.IOException

Useful Filter Input Streams
LineNumberInputStream Class

- This class provides helpful functionality by tracking the number of lines read from an input stream.
- It is deprecated as of JDK1.1, however, since the preferred way to process text data is to use a reader class.
- Also, line numbers are not very serviceable in terms of a stream of bytes.
- Nonetheless, if writing for JDK1.02 systems, it may be useful.

Useful Filter Input Streams *LineNumberInputStream* Class (Cont.)

- Constructors:
  - LineNumberInputStream(InputStream input)

  Creates a line number stream, reading from the specified input stream.
- Methods:
  - int **getLineNumber**()

  Returns the number of lines that have been read by this input stream.
  - void **setLineNumber**(int number)

  Modifies the line number counter to the specified value.

Useful Filter Input Streams *PushBackInputStream* Class

- The PushBackInputStream class allows a single byte to be read and then "pushed back" into the stream for later reading.
- An internal buffer is maintained that allows data to be pushed back into the front of the input stream buffer, or added if the data had never been read from it.
- This is useful when the programmer needs to take a "sneak peek" at what's coming.
  - For example in a text parser or to determine what the next command in a communications protocol is going to be.

Useful Filter Input Streams *PushBackInputStream* Class (Cont.)

- Constructors:
  - PushBackInputStream(InputStream input)

  Creates a PushBackInputStream that will read from the specified input stream.
  - PushBackInputStream (InputStream input int bufferSize) throws java.lang.IllegalArgumentException

  1. *Creates a PushBackInputStream that will read from an input stream and use a buffer of the specified size.*
  2. *If a value of less than one is specified for the buffer size, an exception will be thrown.*

Useful Filter Input Streams *PushBackInputStream* Class (Cont.)

- Methods:
  - void **unread** (byte[] byteArray) throws java.io.IOException

  Pushes back the contents of the specified array. If a buffer overrun occurs, an exception is thrown.
  - void **unread** (byte[] byteArray, int offset, int length) throws java.io.IOException

  Pushes back a subset of the contents of the specified array, starting at the specified offset and lasting for the specified duration. If a buffer overrun occurs, an exception is thrown.
  - void **unread** (int byte) throws java.io.IOException

  Pushes back the specified byte into the front of the buffer. If a buffer overrun occurs, an exception is thrown.

Useful Filter Output Streams
BufferedOutputStream Class

- The BufferedOutputStream provides data buffering similar to the BufferedInputStream.
- As suggested by the name of the class, however, it buffers writes, not reads.
- An internal buffer is maintained, and when the buffer is complete or if a request to flush the buffer is made, the buffer contents are dumped to the output stream to which the buffered stream is connected.

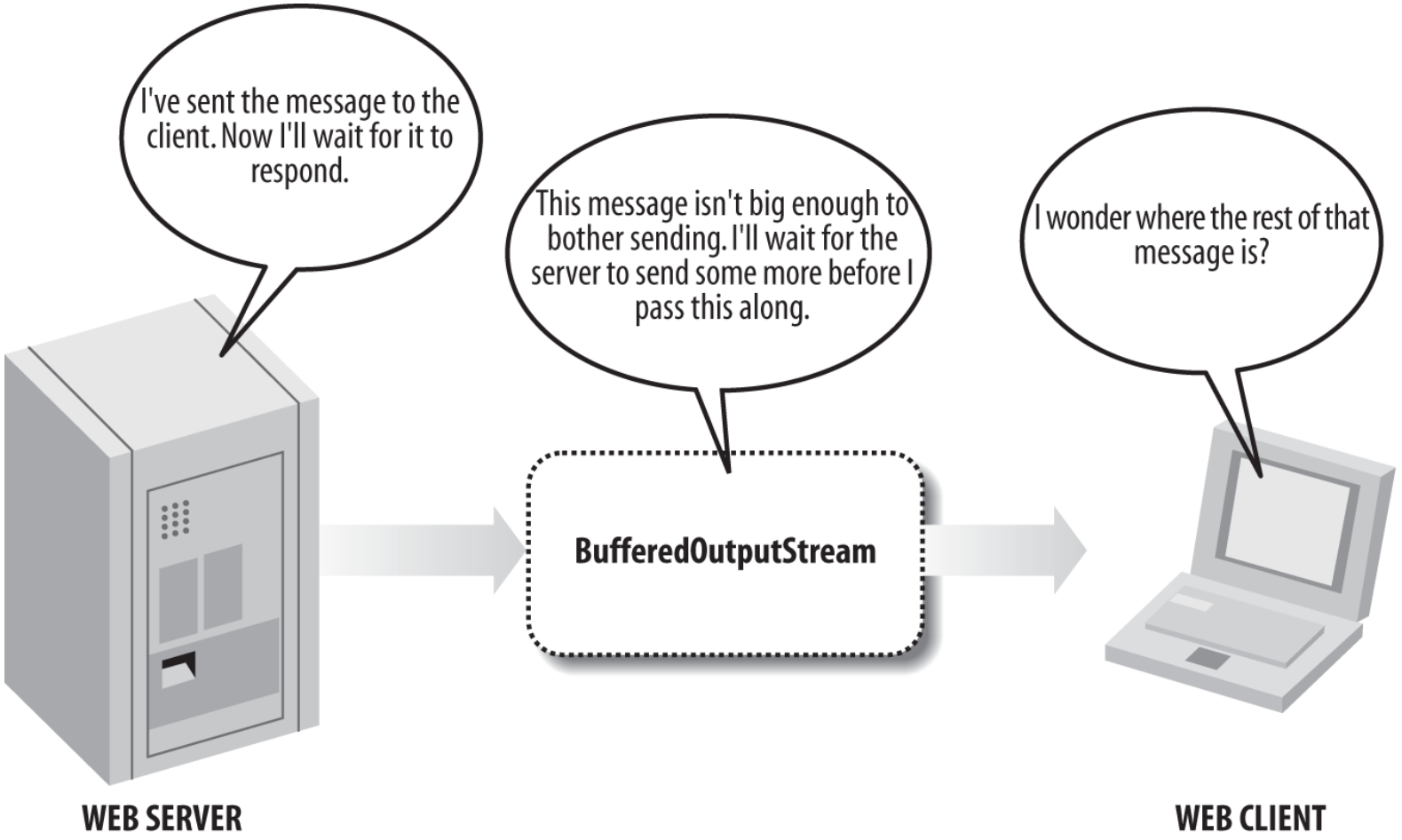
Useful Filter Output Streams *BufferedOutputStream* Class (Cont.)

- Constructors:
  - BufferedOutputStream (OutputStream output)
  1. *Creates a buffer for writing to the specified output stream.*
  2. *The default size of this buffer is 512 bytes in length.*
  - BufferedOutputStream (OutputStream output int bufferSize) throws java.lang.IllegalArgumentException
  1. Creates a buffer for writing to the specified output stream, overriding the default buffer sizing.
  2. The buffer is set to the specified buffer size, which must be greater than zero or an exception is thrown.

Useful Filter Output Streams *BufferedOutputStream* Class (Cont.)

- Methods:
  - No extra methods have been added to this class.
  - However, the flush() method has been overridden.
    - It will flush the contents of a buffer, sending it immediately to the output stream it is connected to.
    - This is particularly important in networking, as a protocol request can't be sent if it is still stuck in the buffer, and the remote program may be waiting for a response.

Useful Filter Output Streams *BufferedOutputStream* Class (Cont.)



Useful Filter Output Streams
DataOutputStream Class

- Like the DataInputStreamclass, the DataOutputStream class is designed to deal with primitive datatypes, such as numbers or bytes.
- Most of the read methods of DataInputStream have a corresponding write method mirrored in DataOutputStream.
  - This allows developers to write datatypes to a file or other type of stream, and to have them read back by another Java application without any compatibility issues over how primitive datatypes are represented by different hardware and software platforms.
- It implements the java.io.DataOutput interface, which provides additional methods for writing primitive datatypes.

Useful Filter Output Streams *DataOutputStream* Class (Cont.)

- Constructors:
  - DataOutputStream (OutputStream output)

  Creates a data output stream, which will write to the specified stream.

Useful Filter Output Streams *DataOutputStream* Class (Cont.)

- Methods:
  - int **size**( )

  Returns the number of bytes written to the data output stream.
  - void **writeBoolean** (boolean value) throws java.io.IOException

  Writes the specified boolean value, represented as a one-byte value.
  - void **writeByte** (int byte) throws java.io.IOException

  Writes the specified byte to the output stream.
  - void **writeBytes** (String string) throws java.io.IOException

  Writes the entire contents of a string to the output stream a byte at a time.
  - void **writeChar** (int char) throws java.io.IOException

  Writes the character to the output stream as a two-byte value.
  - void **writeChars** (String string) throws java.io.IOException

  Writes the entire contents of a string to the output stream, represented as two-byte valuesMethods

Useful Filter Output Streams *DataOutputStream* Class (Cont.)

- Methods:
  - void **writeDouble** (double doubleValue) throws java.io.IOException

  Converts the specified double value to a long value and then converts it to an eight-byte value.
  - void **writeFloat** (float floatValue) throws java.io.IOException

  Converts the specified float value to an int and then writes it as a four-byte value.
  - void **writeInt** (int intValue) throws java.io.IOException

  Writes an int value as a four-byte value.
  - void **writeLong** (int intValue) throws java.io.IOException

  Writes a long value as eight bytes.
  - void **writeShort** (int intValue) throws java.io.IOException

  Writes a short value as two bytes.
  - void **writeUTF** (String string) throws java.io.IOException

  Writes a string using UTF-8 encoding.

Useful Filter Output Streams
PrintStream Class

- The PrintStream is the most unusual of all filter output streams.
  - It is atypical in that it overrides methods inherited from FilterOutputStream without throwing the expected java.io.IOException class.
- The PrintStream adds additional methods as well.
  - None of which may throw an IOException.
  - No errors are overtly reported, and instead the presence of an error is determined by invoking the checkError() method—although no further details may be obtained as to the cause of the error.
- Despite its idiosyncrasies, the PrintStream is an extremely useful class.
  - It provides a convenient way to print primitive datatypes as text using the print(..) method, and to print these with line separators using the println(..) method.

Useful Filter Output Streams *PrintStream* Class (Cont.)

- Constructors*:*
  - PrintStream (OutputStream output)

  Creates a print stream for printing of datatypes as text.
  - PrintStream (OutputStream output, boolean flush)

  1. *Creates a print stream for printing of datatypes as text.*
  2. *If the specified boolean flag is set to "true," whenever a byte array, println method, or newline character is sent, the underlying buffer will be automatically flushed.*

Useful Filter Output Streams *PrintStream* Class (Cont.)

- Methods:
  - boolean **checkError**()

  Automatically flushes the output stream and checks to see if an error has occurred. Instead of throwing an IOException, an internal flag is maintained that checks for errors.
  - void **print** (boolean value)

  Prints a boolean value.
  - void **print** (char character)

  Prints a character value.
  - void **print** (char[] charArray)

  Prints an array of characters.
  - void **print** (double doubleValue)

  Prints a double value.
  - void **print** (float floatValue)

  Prints a float value.

Useful Filter Output Streams *PrintStream* Class (Cont.)

- Methods:
  - void **print** (int intValue)

  Prints an int value.
  - void **print** (long longValue)

  Prints a long value.
  - void **print** (Object obj)

  Prints the value of the specified object's toString() method.
  - void **print** (String string)

  Prints a string's contents.
  - void **println**()

  Sends a line separator (such as '\n'). This value is system dependent and determined by the value of the system property "line.separator."
  - void **println** (char character)

  Prints a character value, followed by a println().
  - void **println** (char[] charArray)

  Prints an array of characters, followed by a println().

Useful Filter Output Streams *PrintStream* Class (Cont.)

- Methods:
  - void **println** (double doubleValue)

  Prints a double value, followed by a println().
  - void **println** (float floatValue)

  Prints a float value, followed by a println().
  - void **println** (int intValue)

  Prints an int value, followed by a println().
  - void **println** (long longValue)

  Prints a long value, followed by a println().
  - void **println** (Object obj)

  Prints the specified object's toString() method, followed by a println().
  - void **println** (String string)

  Prints a string followed by a line separator.
  - protected void **setError**()

  Modifies the error flag to a value of "true."

References

Chapter 4 of *Java™ Network Programming and Distributed Computing*, David Reilly and Michael Reilly.

Networks and Internet Programming

Data Streams

Part-II

Eng. Asma Abdel Karim
Computer Engineering Department

Outline

- Readers and Writers.
- Object Persistence and Object Serialization.

Overview

- While input streams and output streams may be used to read and write text as well as bytes of information and primitive data types, a better alternative is to use ***readers*** and ***writers***.
- Readers and writers were introduced in JDK1.1 to better support Unicode character streams.

What Are Unicode Characters?

- Most people think of characters as being composed of 8 bits of data, offering a range of 256 possible characters.
  - Low ASCII (0–127) characters are followed by high ASCII characters (128–255).
  - The high ASCII characters represent characters and symbols such as those used in foreign languages or punctuation.
- However, people quickly realized that even 256 characters were not enough to handle the many characters used in languages around the world. This is where Unicode came in.

What Are Unicode Characters? (Cont.)

- Unicode characters are represented by 16 bits.
  - Allowing for a maximum of 65,536 possible characters.
- Unicode characters are supported by Java.
- Java also supports a modified form called UTF-8.
  - This is a variable-width encoding format; some characters are a single byte and others multiple bytes.

The Importance of Readers and Writers

- For those dealing solely with primitive data types, use of input streams and output streams may by all means be continued.
- However, if applications are processing text information only, use of a reader and/or a writer, to better support Unicode characters, should be considered.

From Input Streams to Readers

- The *java.io.InputStream* class has a character-based equivalent in the form of the *java.io.Reader* class.
- The reader class has similar method signatures to that of the InputStream class.
  - Existing code may be quickly converted to use it.
- However:
  - Some slight changes are made to the method signatures, to support character, and not byte, reading.
  - The available() method has been removed, and replaced by the ready() method.

The java.io.Reader Class

- Constructors:

No public constructors are available for this class. Instead, a reader subclass should be instantiated.

The java.io.Reader Class (Cont.)

- Methods:

The class includes the following methods, all of which are public:

– void close() throws java.io.IOException

Closes the reader.

– void mark(int amount) throws java.io.IOException

1. *Marks the current position within the reader, and uses the specified amount of characters as a buffer.*
2. *Not every reader will support the mark(int) and reset() methods.*

– boolean markSupported()

Returns "true" if the reader supports mark and reset operations.

The java.io.Reader Class (Cont.)

- Methods:

The class includes the following methods, all of which are public:

- int read() throws java.io.IOException

  Reads and returns a character, blocking if no character is yet available. If the end of the reader's stream has been reached, a value of –1 is returned.

- int read(char[] characterArray) throws java.io.IOException

  Populates an array of characters with data. This method returns an int value, representing the number of bytes that were read. If the end of the reader's stream is reached, a value of –1 is returned and the array is not modified.

- int read(char[] characterArray, int offset, int length) throws java.io.IOException

  Populates a subset of the array with data, starting at the specified offset and lasting for the specified duration. This method returns an int value, representing the number of bytes read, or –1 if no bytes could be obtained.

The java.io.Reader Class (Cont.)

- Methods:

The class includes the following methods, all of which are public:

- boolean ready() throws java.io.IOException

  Returns "true" if there is data available, or "false" if not. This is similar to the InputStream.available() method, except that the number of bytes/characters is not available.

- void reset() throws java.io.IOException

  Attempts to reset the reader's stream, by moving back to an earlier position. Not every reader supports either mark or reset, and an exception could be thrown or the request ignored.

- long skip(long amount) throws java.io.IOException

  Reads and discards the specified number of characters, unless the end of the input stream is reached or another error occurs. The skip method returns the number of characters successfully skipped.

The java.io.Reader Class (Cont.)

- Like input streams:
  - There are a variety of low-level readers (which connect to a data source, such as a file or a data structure), and
  - There are filter readers for high-level communication tasks.

Low-Level Readers
CharArrayReader Class

- The CharArrayReader class is a reader that obtains data by reading characters from an ***array***.
- Constructors:
  - CharArrayReader(char[] charArray)

  Creates a character array reader that will operate on the specified array.
  - CharArrayReader(char[] charArray, int offset, int length)

  Creates a character array reader that will operate only on a subset of the specified array, starting at the specified offset and lasting for the specified length.
- Methods:

The CharArrayReader adds no new methods.

Low-Level Readers
FileReader Class

- This reader obtains its data directly from a local file, similar to the FileInputStream class.
- Care must be taken, as with the FileInputStream class, when creating an instance of it, as an exception will be thrown:
  - If the file could not be located, or
  - If security access permissions restrict it from being read.

Low-Level Readers
FileReader Class (Cont.)

- Constructors:
  - FileReader(File file) throws java.io.FileNotFoundException

  Creates a reader that will access the contents of the specified file object, if the file it represents exists.
  - FileReader(String filename) throws java.io.FileNotFoundException

  Creates a reader that will access the contents of the specified filename, if it exists.
  - FileReader(FileDescriptor descriptor)

  Creates a reader that will access the contents of the specified descriptor handle.
- Methods: The FileReader class adds no new methods.

Low-Level Readers
PipedReader Class

- Constructors:
  - PipedReader()

  Creates an unconnected pipe reader.
  - PipedReader(PipedWriter writer)

  Creates a connected pipe that will read the output of the specified writer.
- Methods:

A single (public) method is added by this class.

  - void connect(PipedWriter writer) throws java.io.IOException

  Connects the reader to the specified writer. Any output that is sent by the piped writer may then be read by the piped reader.

Low-Level Readers
StringReader Class

- While it is sometimes useful to work with a character array, most programmers prefer to deal with strings.
- The StringReader class offers a substitute to the CharArrayReader, accepting a string as an input source.

Low-Level Readers
StringReader Class (Cont.)

- Constructors:
  - StringReader(String stringToBeRead)

  Reads from the beginning of the specified string until the end.
- Methods:

No additional methods are added.

Low-Level Readers
InputStreamReader Class

- While readers are quite common, there is still a need for backward compatibility with older input streams.
  - Particularly those written by third parties for which there is no equivalent reader class.
  - For example, the System.in member variable is an InputStream instance that can read input from a user. There is no comparable reader class for this.
  - The solution is to connect an InputStreamReader to an InputStream instance, which will perform the necessary translation.

Low-Level Readers
InputStreamReader Class (Cont.)

- Constructors:
  - InputStreamReader(InputStream input)

  Connects an input stream to the reader.
  - InputStreamReader(InputStream input, String encoding) throws java.io.UnsupportedEncodingException

  Connects an input stream to the reader using the specified encoding form. If the encoding form isn't supported, an exception is thrown.
- Methods:

The InputStreamReader class adds the following public method:

  - String getEncoding()

  Returns the name of the character encoding used by this stream.

InputStreamToReader Demo

```
import java.io.*;
public class InputStreamToReaderDemo {
        public static void main(String args[]){
           try{
                System.out.print ("Please enter your name : ");
                InputStream input = System.in;
                InputStreamReader reader = new InputStreamReader ( input );
                BufferedReader bufReader = new BufferedReader ( reader );
                String name = bufReader.readLine();
                System.out.println ("Pleased to meet you, " + name);
            }
            catch (IOException ioe){
                System.err.println ("I/O error : " + ioe);
            }
        }
}
```

Filter Readers

- Filter readers, just like filter input streams, provide additional functionality in the form of new methods, or process data in a different way (such as buffering).
- Always connect to another reader.

Filter Readers
BufferedReader Class

- One of the most frustrating problems with reading data from a reader, as with an input stream, is that blocking I/O is used.
- When this happens frequently, the performance and responsiveness of software suffers.
- An alternative is to buffer data so that reads are grouped together for better performance.
- Just as the BufferedInputStream buffers bytes of data, the BufferedReader buffers characters.
- Also, although one would not guess it from the name, the BufferedReader is a partial substitute for the DataInputStream class.
  - It provides a readLine() method that is not deprecated.

Filter Readers
BufferedReader Class (Cont.)

- Constructors:
  - BufferedReader (Reader reader)

  Reads data from the specified reader into a buffer.
  - BufferedReader (Reader reader, int bufferSize) throws java.lang.IllegalArgumentException

  Reads data from the specified reader into a buffer, which is allocated to the specified size. The buffer size must be greater than zero.

Filter Readers
BufferedReader Class (Cont.)

- Methods:
  - The following public method is added by BufferedReader, as a replacement for the deprecated DataInputStream.readLine() method.
    - String readLine() throws java.io.IOException

    Reads a line of text from the underlying stream. The l line is terminated by a line separator sequence, such as a carriage return/linefeed.
  - In addition, the reader overrides the markSupported() method, to indicate that it supports the mark and reset operations.

FilterReaders
FilterReader Class

- Rather than perform a practical action, this class acts as a template on which other filters can be constructed.
  - If a custom filter needs to be written, the class should be extended, and methods overridden or new ones added.
- The FilterReader class has been designed so that it cannot be instantiated by making its constructor protected; the class should instead be extended.
- The FilterReader class defines no new methods, but subclasses are free to add additional methods or override existing ones.

Filter Readers
PushBackReader Class

- This class allows characters to be "pushed back" into the head of a reader's input queue, so that it may be read again.
- This allows programs to peek ahead at the next character and then push it back into the queue.
- Constructors:
  - PushBackReader(Reader reader)

  Creates a push-back reader with a single character buffer.
  - PushBackReader(Reader reader, int bufferSize) throws java.lang.IllegalArgumentException

  Creates a push-back reader with a larger buffer, of the specified size. The buffer size must be greater than zero, or an exception is thrown.

Filter Readers
PushBackReader Class (Cont.)

- Methods:
  - void unread(int character) throws java.io.IOException

  Pushes the character back to the beginning of the queue. If the queue is full, an exception is thrown.
  - void unread(char[] charArray) throws java.io.IOException

  Pushes every character in the specified array into the queue. If full, an exception is thrown.
  - void unread(char[] charArray, int offset, int length) throws java.io.IOException

  Pushes a subset of the characters in the specified array into the queue, starting at the specified offset and lasting for the specified length. If full, an exception is thrown.

Filter Readers
LineNumberReader Class

- The LineNumberReader class provides a useful line counter, which measures how many lines have been read.
- It is the reader equivalent of the LineNumberInputStream.
- As it extends the BufferedReader class, it also supports the mark/reset operations.

Filter Readers
LineNumberReader Class (Cont.)

- Constructors:
  - LineNumberReader (Reader reader)

  Creates a new line-number reader.
  - LineNumberReader (Reader reader, int size)

  Creates a new line-number reader and allocates a buffer of the specified size.
- Methods:

Two new methods, to determine and to modify the line number counter, are offered.

- int getLineNumber()

  Returns the current line number.
- void setLineNumber(int lineNumber)

  Modifies the line-number counter.

Filter Readers
LineNumberReader Class (Cont.)

- Constructors:
  - LineNumberReader (Reader reader)

  Creates a new line-number reader.
  - LineNumberReader (Reader reader int size)

  Creates a new line-number reader and allocates a buffer of the specified size.
- Methods:

Two new methods, to determine and to modify the line number counter, are offered.

- int getLineNumber()

  Returns the current line number.
- void setLineNumber(int lineNumber)

  Modifies the line-number counter.

The java.io.Writer Class

- Constructors:
  - There are no public constructors for this class. Instead, a writer subclass should be instantiated.
- Methods:
  - void close() throws java.io.IOException

  Invokes the flush() method to send any buffered data, and then closes the writer.
  - void flush() throws java.io.IOException

  Flushes any unsent data, sending it immediately. A buffered writer might not yet have enough data to send, and may be storing it for later.

The java.io.Writer Class

- Methods:
  - void write(int character) throws java.io.IOException

  Writes the specified character.
  - void write(char[] charArray) throws java.io.IOException

  Reads the entire contents of the specified character array and writes it.
  - void write(char[] charArray ,int offset, int length) throws java.io.IOException

  Reads a subset of the character array, starting at the specified offset and lasting for the specified length, and writes it.
  - void write(String string) throws java.io.IOException

  Writes the specified string.
  - void write(String string, int offset, int length) throws java.io.IOException

  Writes a subset of the string, starting from the specified offset and lasting for the specified length.

Low-Level Writers
The CharArrayWriter Class

- The CharArrayWriter maintains an internal buffer that is added to each time a write request is made, and may be converted to a character array.
- Constructors:
  - CharArrayWriter()

  Creates a character array writer that can be converted to a character array.
  - CharArrayWriter(int bufferSize) throws java.lang.IllegalArgumentException

  Creates a character array writer using the specified initial buffer size (which must not be negative).
- Methods:
  - char[] toCharArray

  Returns a character array, containing all characters written thus far.
  - String toString()

  Returns a string containing all characters written thus far.

Low-Level Writers
The FileWriter Class

- The FileWriter class extends the OutputStreamWriter class, and provides a convenient way to write characters to a local file.
- This class is equivalent to the FileOutputStream class discussed earlier.
- Constructors:
  - FileWriter (File file) throws java.io.IOException

  Creates a writer connected to the resource represented by the specified file object, if not prevented by security permissions.
  - FileWriter (FileDescriptor descriptor) throws java.io.IOException

  Creates a writer connected to the specified descriptor handle, if allowable.
  - FileWriter(String filename) throws java.io.IOException

  Writes to the specified file location, creating a file if one does not already exist and overwriting an existing one. If not permitted by security access restrictions, an exception will be thrown.
  - FileWriter(String filename, boolean appendFlag) throws java.io.IOException

  Writes to the specified file location. If the appendFlag is set to "true," the file will be opened in append mode and data will be written to the end of the file.

Low-Level Writers
The PipedWriter Class

- The purpose of the PipedWriter class is to write data that will be read by a PipedReader.
- These two classes are reader/writer equivalents of the PipedInputStream and PipedOutputStream classes, but may not be interchanged.
- Constrcutors*:*
  - PipedWriter()

  Creates an unconnected pipe writer.
  - PipedWriter(PipedReader reader) throws java.io.IOException

  Creates a piped writer connected to the specified reader. The reader may later read any data written to this writer.
- Methods:
  - void connect (PipedReader reader) throws java.io.IOException

  Attempts to connect to the specified pipe, so that any data written may be read by the reader. If the pipe is already connected to another pipe, an IOException will be thrown.

Low-Level Writers
The StringWriter Class

- Judging by its name, you might expect that this class allowed for writing to a string.
- A string is of fixed length and is immutable (the contents of a string may not be modified).
- Writing to a string is accomplished by using a StringBuffer.
  - The StringBuffer class is similar to a string, but may be modified. When the modifications are complete, the StringBuffer can be converted back to a string.
- This is how the StringWriter class works.
  - It maintains a string buffer, and provides a method to access the buffer contents or to convert to a string.

Low-Level Writers
The StringWriter Class

- Constructors:
  - StringWriter()

  Creates a new string writer, using the default-sized buffer.
  - StringWriter(int startingSize)

  Creates a new string writer and allocates a StringBuffer of the specified size.
- Methods:
  - StringBuffer getBuffer()

  Returns the buffer used to store data sent to the writer.
  - String toString()

  Converts the internal buffer into a string.

Low-Level Writers
The OutputStreamWriter Class

- While there are many writer classes equivalent to output stream classes in the Java API, there is still a need to maintain compatibility with older output stream classes.
  - As most of the networking API and some third-party libraries provide only stream interfaces.
- The OutputStreamWriter class handles the translation between a Writer and an OuputStream, allowing new writer classes to interact with older output streams.

Low-Level Writers
The OutputStreamWriter Class

- Constructors:
  - OutputStreamWriter(OutputStream output)

  Creates a writer that will translate between characters and bytes, using the default character encoding.
  - OutputStreamWriter(OutputStream output, String encoding) throws java.io.UnsupportedEncodingException

  Creates a writer that translates between characters and bytes, using the specified character encoding. If the specified encoding form is not supported, an exception is thrown.
- Methods:
  - String getEncoding()

  Returns the character encoding used by the writer.

OutputStreamToWriter Demo

```
import java.io.*;
public class OutputStreamToWriterDemo
{
    public static void main(String args[])
    {
        try
        {
            //Get the output stream representing standard output
            OutputStream output = System.out;
            // Create an OutputStreamWriter
            OutputStreamWriter writer = new OutputStreamWriter (output);
            // Write to standard output using a writer
            writer.write ("Hello world");
            // Flush and writer, to ensure it is written
            writer.flush(); writer.close();
        }
        catch (IOException ioe)
        {
            System.err.println ("I/O error : " + ioe);
        }
    }
}
```

Filter Writers
The BufferedWriter Class

- Used to improve system performance by buffering write request together.
- Constructors:
  - BufferedWriter(Writer writer)

  Creates a buffered writer, connected to the specified writer. Write requests will be buffered, to improve efficiency. To send all queued data, the flush() method should be invoked.
  - BufferedWriter(Writer writer, int bufferSize) throws java.lang.IllegalArgumentException

  Creates a buffered writer, with a buffer of the specified size. The size must be greater than or equal to 1.

Filter Writers
The FilterWriter Class

- Developers creating custom filter classes should extend this class, rather than extending the java.io.Writer class.
- It provides no additional functionality, but may be used as a template on which filters can be constructed.
- Constructors
  - *protected FilterWriter(Writer writer)*
- Methods:
  - The FilterWriter class defines no new methods, but subclasses are free to add additional methods or override existing ones.

Filter Writers
The PrintWriter Class

- PrintWriter is the sister class of PrintStream, and provides the same methods for writing datatypes as text.
- Like PrintStream, none of the methods may throw an IOException—rather, the error state is determined by invoking the checkError() method, which returns a boolean value.
- Constructors:
  - PrintWriter(Writer writer)

  Creates a print writer, writing to the specified writer.
  - PrintWriter(Writer writer, boolean flushFlag)

  Creates a print writer, the output of which may or may not be automatically flushed whenever a println() method or a line separator is sent, based on the state of the specified boolean flag. A value of "true" will flush when a println method is executed.
- Methods:
  - The PrintWriter class implements new methods to match the signatures of the PrintStream class.

Object Persistence & Object Serialization

- Data that can be read or written ranges from individual bytes to primitive datatypes and strings.
- But what if you wanted to read and write an entire object, composed of a series of member variables?
- To do this would require that each field of the object be written individually; then at a later time, each field would be read back and assigned to an object.
  - This is a complicated process.
  - The solution is to use object persistence.

What is Object Persistence

- Object persistence is the ability of an object to persist over time (and, if moved to a different computer or JVM, over space).
- Most objects in a Java Virtual Machine are fairly short-lived.
  - When there are no references to an object, the memory space allocated to it is reclaimed by the automatic garbage collector thread.
  - If an object is frequently used, and does not lose references to it, it will still die at some point in time the JVM will terminate eventually and the object will be destroyed.
- Object persistence allows an object to outlive the JVM that hosts it.

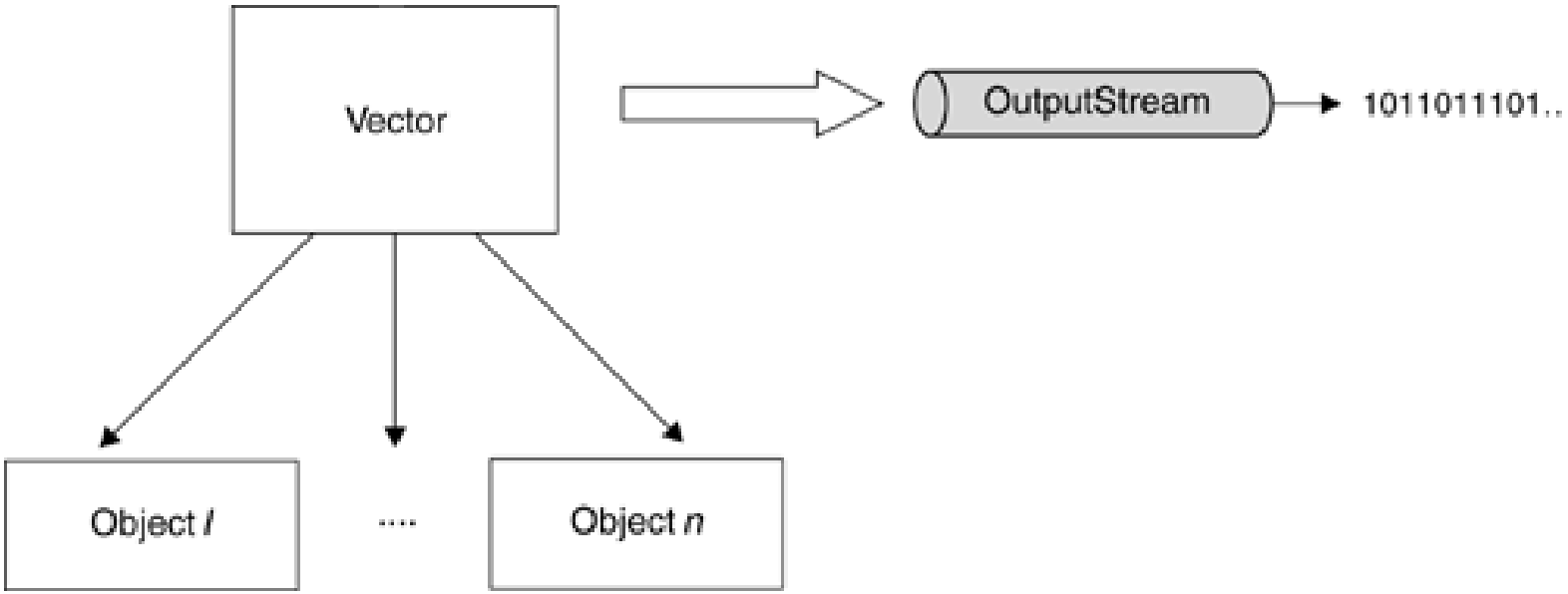
What is Object Serialization

- Object serialization controls how the data that comprises an object's state information (the individual member variables, whether public, private, or protected) is written as a sequence of bytes.
- The serialized object might be sent over the network or saved to a disk so that it can be accessed at some point in the future.
- This allows objects to move from one JVM to another, whether located on the same machine or a remote one.

What is Object Serialization (Cont.)

- Serialization works by examining the variables of an object and writing primitive datatypes like numbers and characters to a byte stream.
- If an object contains an object as a member variable (as opposed to a primitive datatype), the object member variable must be serialized as well.
- This must be done recursively, so that if an object has a reference to an object, which has a reference to another object (and so on), they are all saved together.
- The set of all objects referenced is called a graph of objects, and object serialization converts entire graphs to byte form.

Graphs of Objects



How Serialization Works?

- Support for serialization was introduced in JDK1.1.
- Any object that implements the ***java.io.Serializable*** interface may be serialized with only a few lines of code (along with any other object referenced by a serialized object).
- The interface serves only as an indication that the developer endorses serialization—no methods need to be implemented to support serialization.
- Implementing the java.io.Serializable interface requires no additional effort on the part of developers, other than:
  - Adding the appropriate "implements" statement during the class declaration and,
  - Declaring a no-argument constructor (also referred to as the default constructor).
    - The constructor is required so that the class maybe instantiated later by the JVM, and then deserialized by assigning new values to member variables.

Example

```
Public class SomeClass extends SomeOtherClass
   implements java.io.Serializable {
        public SomeClass()
        {
        }
                .....
}
```

Serialization Issues

- There are some legitimate reasons, too, for not supporting serialization.
  - For example, if an object contained very sensitive information, it might be unwise to serialize it and save it to disk or send it over a network.
  - Developers should be aware that no special care is taken to protect the contents of a serialized object from scrutiny or modification, and that any class in any JVM may choose to deserialize an object at a later time.

Serialization Issues (Cont.)

- To prevent individual member variables being serialized, they can be marked with the ***transient*** keyword, which indicates that the object or primitive datatype must not be serialized.
- Other uses for the *transient* keyword are for fields that are being continuously updated by some means, such as a timer, and hence do not make sense to serialize.

Example

```
Public class UserAccount implements
   java.io.Serializable {
        protected String username;
        protected transient String password;

        public UserAccount( )
        {
                ....
        }
}
```

Reading and Writing Objects to Streams

- The main point of serialization is to write an object out to a stream and to read it back.
- This is accomplished by using the *java.io.ObjectOutputStream* and *java.io.ObjectInputStream* classes, which can write serializable objects out to an output stream and read them back from an input stream.

The ObjectInputStream Class

- The ***ObjectInputStream*** class is used to read a serialized object from a byte stream, to allow an object to be reconstituted back to its original form, providing the object's class can be loaded by the JVM's class loader.
- The ***ObjectInputStream*** class implements the ***ObjectInput*** interface, which extends the ***DataInput*** interface.
- This means that the ***ObjectInputStream*** class provides many methods with the same signature as ***DataInputStream***, in addition to extra methods responsible for reading objects.

The ObjectInputStream Class (Cont.)

- Constructors:
  - protected ObjectInputStream() throws java.io.IOException java.lang.SecurityException

  Provides a default constructor for ObjectInputStream subclasses.
  - ObjectInputStream(InputStream input) throws java.io.IOException

  Creates an object input stream connected to the specified input stream, which is capable of restoring serialized objects.

The ObjectInputStream Class (Cont.)

- Methods:

Many of the methods of ***ObjectInputStream*** were covered in the discussion of the ***DataInputStream*** class.

ObjectInputStream can read primitive datatypes just like the DataInputStream class.

- public final Object readObject() throws java.io.OptionalDataException, java.io.IOException, java.lang.ClassNotFoundException

Reads a serialized object from the stream and reconstructs it to its original state. If the object contains references to other objects, these objects are also reconstructed. If an object cannot be read, the application will be notified by the method throwing an exception. An Object instance is returned. If required, this object can be cast to a specific class type before it is used.

The ObjectOutputStream Class

- The ObjectOutputStream class serializes an object to a byte stream, for the purpose of object persistence.
- It may be connected to any existing output stream, such as a file or a networking stream, for transmission over the Internet.
- Objects written to an ObjectOutputStream have all their member variables (such as primitive data types and objects) written.
- If the object contains references to other objects, they too will be written, so an ObjectOutputStream can write entire object graphs.
- A sequence of objects can be written or wrapped in a collection (such as an array or a vector) whose entire contents could be serialized with one call to the ObjectOutputStream.writeObject method.

The ObjectOutputStream Class (Cont.)

- Constructors:
  - protected ObjectOutputStream () throws java.io.IOException java.lang.SecurityException

  Default constructor, provided for the benefit of subclasses of the ObjectOutputStream.

  - ObjectOutputStream (OutputStream output) throws java.io.IOException

  Creates an object output stream capable of serializing objects to the specified output stream.

The ObjectOutputStream Class (Cont.)

- Methods:
- The ObjectOutputStream class also provides method implementations for the DataOutput interface.
  - void writeObject (Object object) throws java.io.IOException, java.io.InvalidClassException,java.io.NotSerializableException

  Writes the specified object to the output stream, through object serialization. All variables that are not marked as transient or static will be written, providing the specified class is an instance of the java.io.Serializable interface.

References

Chapter 4 of *Java™ Network Programming and Distributed Computing*, David Reilly and Michael Reilly.

Networks and Internet Programming

User Datagram Protocol

Eng. Asma Abdel Karim
Computer Engineering Department

Outline

- Overview.
- DatagramPacket Class.
- DatagramSocket Class.
- Listening for UDP Packets.
- Sending UDP Packets.
- Additional Information on UDP.

Eng. Asma Abdel Karim
Computer Engineering Department

Overview

- The User Datagram Protocol (UDP) is a commonly used transport protocol employed by many types of applications.
- UDP is a ***connectionless*** transport protocol, meaning that it doesn't guarantee either packet delivery or that packets arrive in sequential order.
- Rather than reading from, and writing to, an ordered sequence of bytes using I/O streams, bytes of data are grouped together in discrete packets, which are sent over the network.

Overview

- The packets may travel along different paths, as selected by the various network routers that distribute traffic flow, depending on factors such as network congestion, priority of routes, and cost of transmission.
  - This means that a packet can arrive out of sequence, if it encounters a faster route than the previous packet (or if the previous packet encounters some other form of delay).
  - No two packets are guaranteed the same route, and if a particular route is heavily congested, the packet may be discarded entirely. Each packet has a time-to-live (TTL) counter, which is updated when the packet is routed along to the next point in the network. When the timer expires, it will be discarded, and the recipient of the packet will not be notified.
  - If a packet does arrive, however, it will always arrive intact. Packets that are corrupt or only partially delivered are discarded.

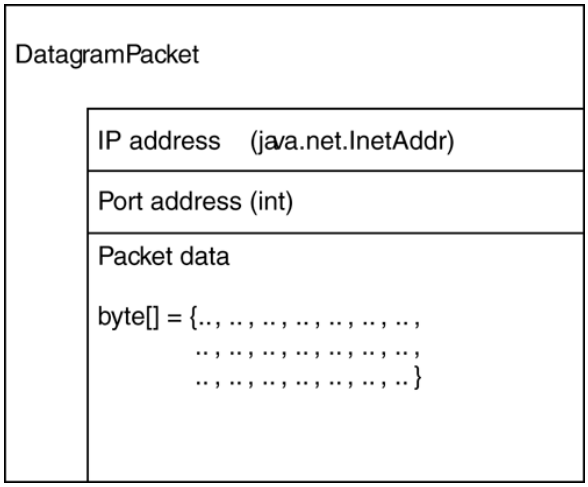
Advantages of UDP

- UDP communication can be more efficient than guaranteed-delivery data streams. If the amount of data is small and the data is sent frequently.
- Unlike TCP streams, which establish a connection, UDP causes fewer overheads.
  - If the amount of data being sent is small and the data is sent infrequently, the overhead of establishing a connection might not be worth it.
  - If data is being sent from a large number of machines to one central one, in which case the sum total of all these connections might cause significant overload.
- Real-time applications that demand up-to-the-second or better performance may be candidates for UDP, as there are fewer delays due to the error checking and flow control of TCP.
- UDP sockets can receive data from more than one host machine. If several machines must be communicated with, then UDP may be more convenient than other mechanisms such as TCP.
- Some network protocols specify UDP as the transport mechanism, requiring its use.

java.net.DatagramPacket Class

- The ***DatagramPacket*** class represents a data packet intended for transmission using the User Datagram Protocol.
- Packets are containers for a small sequence of bytes, and include addressing information such as an IP address and a port.

java.net.DatagramPacket Class



java.net.DatagramPacket Class

- The meaning of the data stored in a ***DatagramPacket*** is determined by its context.
- When a ***DatagramPacket*** has been read from a UDP socket, the IP address of the packet represents the address of the sender (likewise with the port number).
- However, when a ***DatagramPacket*** is used to send a UDP packet, the IP address stored in ***DatagramPacket*** represents the address of the recipient (likewise with the port number).

Creating a DatagramPacket

There are two reasons to create a new ***DatagramPacket***:

1. To send data to a remote machine using UDP.

2. To receive data sent by a remote machine using UDP.

Creating a DatagramPacket

- Constructors:
  - The choice of which DatagramPacket constructor to use is determined by its intended purpose.
  - Either constructor requires the specification of a byte array, which will be used to store the UDP packet contents, and the length of the data packet.
  - To create a DatagramPacket for receiving incoming UDP packets, the following constructor should be used:

    DatagramPacket(byte[] buffer, int length).

    For example:

    DatagramPacket packet = new DatagramPacket(new byte[256], 256);
  - To send a DatagramPacket to a remote machine, it is preferable to use the following constructor:

    DatagramPacket(byte[] buffer, int length, InetAddress dest_addr, int dest_port).

    For example:

    InetAddress addr = InetAddress.getByName("192.168.0.1");

    DatagramPacket packet = new DatagramPacket (new byte[128],128, addr, 2000);

Creating a DatagramPacket

- Methods:
  - The ***DatagramPacket*** class provides some important methods that allow the remote address, remote port, data (as a byte array), and length of the packet to be retrieved.
  - As of JDK1.1, there are also methods to modify these, via a corresponding set method. This means that a received packet can be reused.
    - For example, a packet's contents can be replaced and then sent back to the sender. This saves having to reset addressing information—the address and port of the packet are already set to those of the sender.

Creating a DatagramPacket

- Methods:
  - InetAddress getAddress()

  Returns the IP address from which a DatagramPacket was sent, or (if the packet is going to be sent to a remote machine), the destination IP address.
  - byte[] getData()

  Returns the contents of the DatagramPacket, represented as an array of bytes.
  - int getLength()

  Returns the length of the data stored in a DatagramPacket. This can be less than the actual size of the data buffer.
  - int getPort()

  Returns the port number from which a DatagramPacket was sent, or (if the packet is going to be sent to a remote machine), the destination port number.

Creating a DatagramPacket

- Methods:
  - void setAddress(InetAddress addr)

  Assigns a new destination address to a DatagramPacket.
  - void setData(byte[] buffer)

  Assigns a new data buffer to the DatagramPacket. Remember to make the buffer long enough, to prevent data loss.
  - void setLength(int length)

  Assigns a new length to the DatagramPacket. Remember that the length must be less than or equal to the maximum size of the data buffer, or an IllegalArgumentException will be thrown. When sending a smaller amount of data, you can adjust the length to fit—you do not need to resize the data buffer.
  - void setPort(int port)

  Assigns a new destination port to a DatagramPacket.

java.net.DatagramSocket Class

- The ***DatagramSocket*** class provides access to a UDP socket, which allows UDP packets to be sent and received.
- A ***DatagramPacket*** is used to represent a UDP packet, and must be created prior to receiving any packets.
- The same ***DatagramSocket*** can be used to receive packets as well as to send them.
- Read operations are blocking, meaning that the application will continue to wait until a packet arrives.
  - Since UDP packets do not guarantee delivery, this can cause an application to stall if the sender does not resubmit packets.
  - You can use multiple threads of execution, or as of JDK1.1, you can use nonblocking I/O to avoid this problem.

Creating a DatagramSocket

- A ***DatagramSocket*** can be used to both send and receive packets.
- Each ***DatagramSocket*** binds to a port on the local machine, which is used for addressing packets.
- The port number need not match the port number of the remote machine, but if the application is a UDP server, it will usually choose a specific port number.
- If the ***DatagramSocket*** is intended to be a client, and doesn't need to bind to a specific port number, a blank constructor can be specified.

Creating a DatagramSocket (Cont.)

- To create a client DatagramSocket, the following constructor is used:

DatagramSocket() throws java.net.SocketException

- To create a server Datagram Socket, the following constructor is used, which takes as a parameter the port to which the UDP service will be bound:

DatagramSocket(int port) throws java.net.SocketException

Creating a DatagramSocket (Cont.)

- Although rarely used, there is a third constructor for ***DatagramSocket***, introduced in JDK1.1.
- If a machine is known by several IP addresses, you can specify the IP address and port to which a UDP service should be bound.
- It takes as parameters the port to which the UDP service will be bound, as well as the InetAddress of the service.
- This constructor is:

DatagramSocket (int port, InetAddress addr) throws java.net.SocketException

Using a DatagramSocket

- ***DatagramSocket*** is used to receive incoming UDP packets and to send outgoing UDP packets.
- It provides methods to:
  - Send and receive packets,
  - Specify a timeout value when nonblocking I/O is being used,
  - Inspect and modify maximum UDP packet sizes, and
  - Close the socket.

Using a DatagramSocket (Cont.)

- void close()

Closes a socket, and unbinds it from the local port.

- void connect(InetAddress remote_addr, int remote_port)

Restricts access to the specified remote address and port. The designation is a misnomer, as UDP doesn't actually create a "connection" between one machine and another. However, if this method is used, it causes exceptions to be thrown if an attempt is made to send packets to, or read packets from, any other host and port than those specified.

- void disconnect()

Disconnects the DatagramSocket and removes any restrictions imposed on it by an earlier connect operation.

Using a DatagramSocket (Cont.)

- InetAddress getInetAddress()

Returns the remote address to which the socket is connected, or null if no such connection exists.

- int getPort()

Returns the remote port to which the socket is connected, or –1 if no such connection exists.

- InetAddress getLocalAddress()

Returns the local address to which the socket is bound.

- int getLocalPort()

Returns the local port to which the socket is bound.

- int getReceiveBufferSize() throws java.net.SocketException

Returns the maximum buffer size used for incoming UDP packets.

- int getSendBufferSize() throws java.net.SocketException

Returns the maximum buffer size used for outgoing UDP packets.

Using a DatagramSocket (Cont.)

- int getSoTimeout() throws java.net.SocketException

Returns the value of the timeout socket option. This value is used to determine the number of milliseconds a read operation will block before throwing a java.io.InterruptedIOException. By default, this value will be zero, indicating that blocking I/O will be used.

- void receive(DatagramPacket packet) throws java.io.IOException

Reads a UDP packet and stores the contents in the specified packet. The address and port fields of the packet will be overwritten with the sender address and port fields, and the length field of the packet will contain the length of the original packet, which can be less than the size of the packet's byte-array. If a timeout value hasn't been specified by using DatagramSocket.setSoTimeout(int duration), this method will block indefinitely. If a timeout value has been specified, a java.io.InterruptedIOException will be thrown if the time is exceeded.

Using a DatagramSocket (Cont.)

- void send(DatagramPacket packet) throws java.io.IOException

Sends a UDP packet, represented by the specified packet parameter.

- void setReceiveBufferSize(int length) throws java.net. SocketException

Sets the maximum buffer size used for incoming UDP packets. Whether the specified length will be adhered to is dependent on the operating system.

- void setSendBufferSize(int length) throws java.net.SocketException

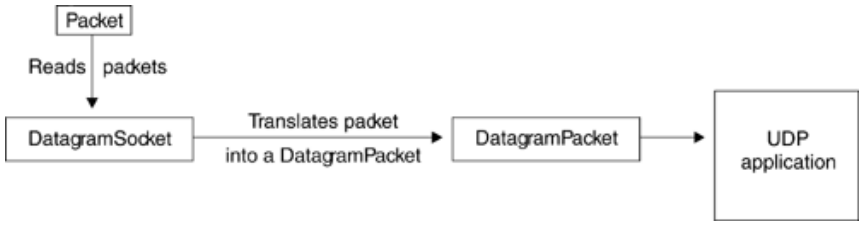
Sets the maximum buffer size used for outgoing UDP packets. Whether the specified length will be adhered to is dependent on the operating system.

- void setSoTimeout(int duration) throws java.net.SocketException

Sets the value of the timeout socket option. This value is the number of milliseconds a read operation will block before throwing a java.io.InterruptedIOException.

Listening for UDP Packets

- Before an application can read UDP packets sent to it by remote machines, it must:
  - Bind a socket to a local UDP port using ***DatagramSocket***, and
  - Create a ***DatagramPacket*** that will act as a container for the UDP packet's data.



Listening for UDP Packets (Cont.)

- When an application wishes to read UDP packets, it calls the ***DatagramSocket.receive*** method, which copies a UDP packet into the specified ***DatagramPacket***. The contents of the ***DatagramPacket*** are processed, and the process is repeated as needed.

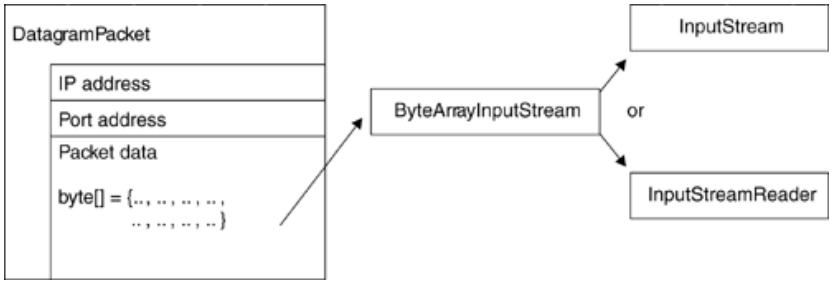
```
DatagramPacket packet = new DatagramPacket (new byte[256], 256);
DatagramSocket socket = new DatagramSocket(2000);
boolean finished = false;
while (! finished )
{
        socket.receive (packet);
        // process the packet
}
socket.close();
```

Listening for UDP Packets (Cont.)

- When processing the packet, the application must work directly with an array of bytes.
- If, however, your application is better suited to reading text, you can use classes from the Java I/O package to convert between a byte array and another type of stream or reader.
  - By hooking a ***ByteArrayInputStream*** to the contents of a datagram and then to another type of ***InputStream*** or an ***InputStreamReader***, you can access the contents of UDP packets relatively easily.

```
ByteArrayInputStream bin = new ByteArrayInputStream(
packet.getData() );
DataInputStream din = new DataInputStream (bin);
// Read the contents of the UDP packet
```

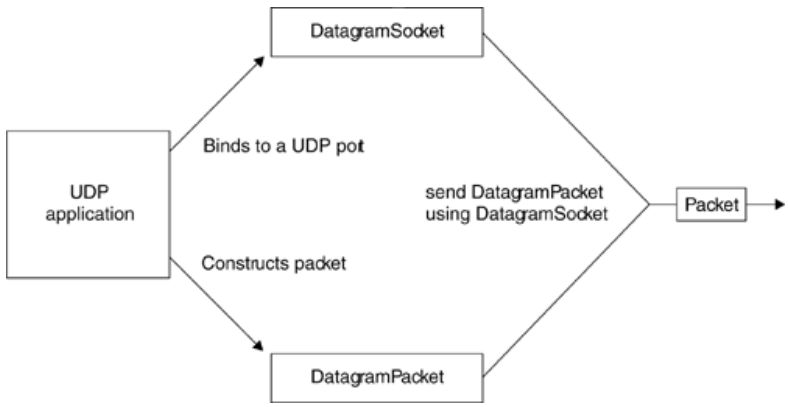
Listening for UDP Packets (Cont.)



Sending UDP Packets

- The same interface (***DatagramSocket***) employed to receive UDP packets is also used to send them.
- When sending a packet, the application must create a ***DatagramPacket***, set the address and port information, and write the data intended for transmission to its byte array.
- If replying to a received packet, the address and port information will already be stored, and only the data need be overwritten.
- Once the packet is ready for transmission, the send method of ***DatagramSocket*** is invoked, and a UDP packet is sent.

Sending UDP Packets (Cont.)



Sending UDP Packets (Cont.)

```
DatagramSocket socket = new DatagramSocket(2000);
DatagramPacket packet = new DatagramPacket (new byte[256], 256);
packet.setAddress ( InetAddress.getByName ( somehost ) );
packet.setPort ( 2000 );
boolean finished = false;
while !finished )
{
        // Write data to packet buffer
        ........
        socket.send (packet);
        // Do something else, like read other packets, or check to
        // see if no more packets to send
        .........
}
socket.close();
```

Additional Information on UDP

- While the UDP is sometimes the best alternative for certain classes of applications, because of its unique properties, it does present some challenges to developers.
  - Lack of guaranteed delivery.
  - Lack of guaranteed packet sequencing.
  - Lack of flow control.

References

Chapter 5 of *Java™ Network Programming and Distributed Computing*, David Reilly and Michael Reilly.

Networks and Internet Programming

Transmission Control Protocol

Eng. Asma Abdel Karim
Computer Engineering Department

Outline

- Overview.
- Advantages of TCP Over UDP.
- Communication between Applications Using Ports.
- Socket Operations.
- TCP and the Client/Server Paradigm.
- TCP Sockets and Java.
- Socket Class.
- Creating a TCP Client.
- ServerSocket Class.
- Creating a TCP Server.
- Exception Handling: Socket-Specific Exceptions.

Eng. Asma Abdel Karim
Computer Engineering Department

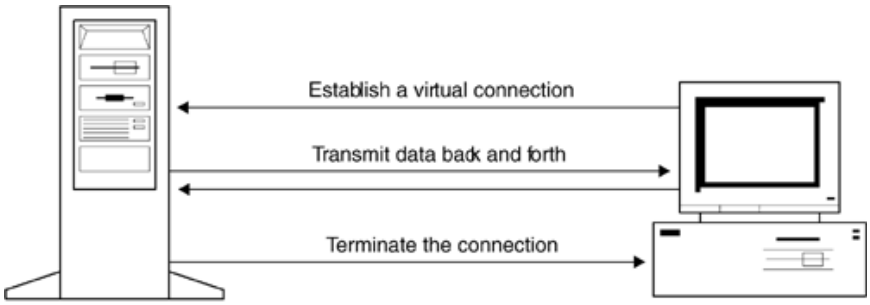
Overview

- The properties of TCP make it highly attractive to network programmers.
  - As it simplifies network communication by removing many of the obstacles of UDP, such as ordering of packets and packet loss.
- UDP is concerned with the transmission of packets of data.
  - TCP focuses instead on establishing a network connection, through which a stream of bytes may be sent and received.

Overview (Cont.)

- Packets may be sent through a network using various paths and may arrive at different times.
- This benefits performance and robustness, as the loss of a single packet doesn't necessarily disrupt the transmission of other packets.
- Nonetheless, such a system creates extra work for programmers who need to guarantee delivery of data.
- TCP eliminates this extra work by guaranteeing delivery and order, providing for a reliable byte communication stream between client and server that supports two-way communication.
- TCP establishes a "virtual connection" between two machines, through which streams of data may be sent.

Overview (Cont.)



Overview (Cont.)

- TCP uses a lower-level communications protocol, the ***Internet Protocol (IP)***, to establish the connection between machines.
  - This connection provides an interface that allows streams of bytes to be sent and received, and transparently converts the data into IP datagram packets.
- TCP provides guaranteed delivery of bytes of data.
  - Of course, it's always possible that network errors will prevent delivery, but TCP handles the implementation issues such as resending packets, and alerts the programmer only in serious cases such as if there is no route to a network host or if a connection is lost.

Overview (Cont.)

- The virtual connection between two machines is represented by a ***socket***.
- There are substantial differences between a UDP socket and a TCP socket.
  - First, TCP sockets are connected to a single machine, whereas UDP sockets may transmit or receive data from multiple machines.
  - Second, UDP sockets only send and receive ***packets*** of data, whereas TCP allows transmission of data through ***byte streams*** (represented as an InputStream and OutputStream). They are converted into datagram packets for transmission over the network, without requiring the programmer to intervene.

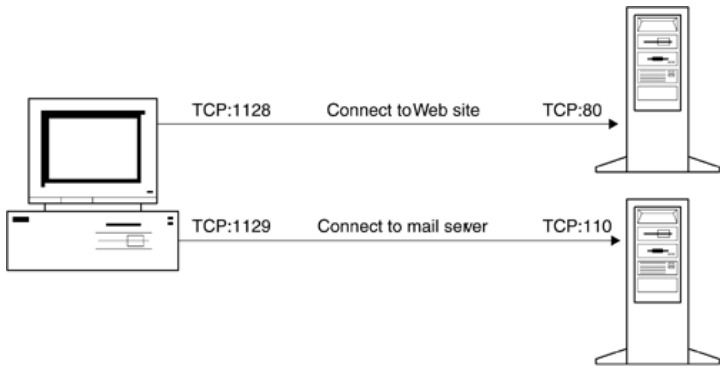
Advantages of TCP over UDP

- Automatic Error Control.
- Reliability.
- Ease of Use.

Communication between Applications Using Ports

- It is clear that there are significant differences between TCP and UDP, but there is also an important similarity between these two protocols. Both share the concept of a communications port, which distinguishes one application from another.
- When a TCP socket establishes a connection to another machine, it requires two very important pieces of information to connect to the remote end—the IP address of the machine and the port number.
- In addition, a local IP address and port number will be bound to it, so that the remote machine can identify which application established the connection.

Communication between Applications Using Ports (Cont.)



Communication between Applications Using Ports (Cont.)

- Ports in TCP are just like ports in UDP—they are represented by a number in the range 1–65535.
- Ports below 1024 are restricted to use by well-known services such as HTTP, FTP, SMTP, POP3, and telnet.

Well-Known Services	Service Port
Telnet	23
Simple Mail Transfer Protocol	25
HyperTexr Transfer Protocol	80
Post Office Protocol 3	110

Socket Operations

- TCP sockets can perform a variety of operations. They can:
  - Establish a connection to a remote host.
  - Send data to a remote host.
  - Receive data from a remote host.
  - Close a connection.
- In addition, there is a special type of socket that provides a service that will bind to a specific port number. This type of socket is normally used only in servers, and can perform the following operations:
  - Bind to a local port.
  - Accept incoming connections from remote hosts.
  - Unbind from a local port.
- These two sockets are grouped into different categories, and are used by either a client or a server (since some clients may also be acting as servers, and some servers as clients). However, it is normal practice for the role of client and server to be separate.

TCP and the Client/Server Paradigm

- The client/server paradigm divides software into two categories, clients and servers.
  - A client is software that initiates a connection and sends requests, whereas
  - A server is software that listens for connections and processes requests.
- In the context of UDP programming, no actual connection is established, and UDP applications may both initiate and receive requests on the same socket.
- In the context of TCP, where connections are established between machines, the client/server paradigm is much more relevant.

TCP and the Client/Server Paradigm (Cont.)

- When software acts as a client, or as a server, it has a rigidly defined role that fits easily into a familiar mental model.
  - Either the software is initiating requests, or it is processing them.
- Switching between these roles makes for a more complex system.
  - Even if switching is permitted, at any given time one software program must be the client and one software program must be the server. If they both try to be clients at the same time, no server exists to process the requests!

Network Clients

- Network clients initiate connections and usually take charge of network transactions.
- The server is there to fulfill the requests of the client—a client does not fulfill the requests of a server.
- Although the client is in control, some power still resides in the server, of course. A client can tell a server to delete all files on the local file system, but the server isn't necessarily compelled to carry out that action.
- The network client speaks to the server using an agreed-upon standard for communication, the network protocol.
  - For example, an HTTP client uses a set of commands different from a mail client, and has a completely different purpose.
  - Connecting an HTTP client to a mail server, or a mail client to an HTTP server, will result not only in an error message but in an error message that the client will not understand.
  - For this reason, as part of the protocol specification, a port number is used so that the client can locate the server.

Network Servers

- The role of the network server is to bind to a specific port (which is used by the client to locate the server), and to listen for new connections.
- While the client is temporary, and runs only when the user chooses, the server must run continually (even if no clients are actually connected) in the hope that someone, at some time, will want its services.
- Some servers can handle only a single connection at a time, while others can handle many connections concurrently, through the use of threads.

TCP Sockets and Java

- Java offers good support for TCP sockets, in the form of two socket classes, ***java.net.Socket*** and ***java.net.ServerSocket***.
- When writing client software that connects to an existing service, the ***Socket*** class should be used.
- When writing server software that binds to a local port in order to provide a service, the ***ServerSocket*** class should be employed.
- This is different from the way a ***DatagramSocket*** works with UDP.
  - The function of connecting to servers, and the function of accepting data from clients, is split into a separate class under TCP.

Socket Class

- The ***Socket*** class represents client sockets, and is a communication channel between two TCP communications ports belonging to one or two machines.
- A socket may connect to a port on the local system, avoiding the need for a second machine, but most network software will usually involve two machines.
- TCP sockets can't communicate with more than two machines, however.
  - If this functionality is required, a client application should establish multiple socket connections, one for each machine.

Socket Class - Constructors

- The easiest way to create a socket is to specify the hostname of the machine and the port of the service. For example, to connect to a Web server on port 80, the following code might be used:

```
try{
        // Connect to the specified host and port
        Socket mySocket = new Socket ( "www.awl.com", 80);
        // ......
}
catch (Exception e){
        System.err.println ("Err – " + e);
}
```

Socket Class – Constructors (Cont.)

- **protected Socket ()**
  - *Creates an unconnected socket using the default implementation provided by the current socket factory. Developers should not normally use this constructor , as it does not allow a hostname or port to be specified.*
- **Socket (InetAddress address, int port) throws java.io.IOException, java.lang.SecurityException**
  - *Creates a socket connected to the specified IP address and port. If a connection cannot be established, or if connecting to that host violates a security restriction (such as when an applet tries to connect to a machine other than the machine from which it was loaded), an exception is thrown.*
- **Socket (InetAddress address, int port, InetAddress localAddress, int localPort) throws java.io.IOException, java.lang.SecurityException**
  - *Creates a socket connected to the specified address and port, and is bound to the specified local address and local port. By default, a free port is used, but this method allows you to specify a specific port number, as well as a specific address, in the case of multihomed hosts (i.e., a machine where the localhost is known by two or more IP addresses).*

Socket Class – Constructors (Cont.)

- **protected Socket (SocketImpl implementation)**
  - Creates an unconnected socket using the specified socket implementation. Developers should not normally use this constructor, as it does not allow a hostname or port to be specified.
- **Socket (String host, int port) throws java.net.UnknownHostException, java.io.IOException, java.lang.SecurityException**
  - Creates a socket connected to the specified host and port. This method allows a string to be specified, rather than an InetAddress. If the hostname could not be resolved, a connection could not be established, or a security restriction is violated, an exception is thrown.
- **Socket (String host, int port, InetAddress localAddress, int localPort) throws java.net.UnknownHostException, java.io. IOException, java.lang.SecurityException**
  - Creates a socket connected to the specified host and port, and bound to the specified local port and address. This allows a hostname to be specified as a string, and not an InetAddress instance, as well as allowing a specific local address and port to be bound to. These local parameters are useful for multihomed hosts (i.e., a machine where the localhost is known by two or more IP addresses). If the hostname can't be resolved, a connection cannot be established, or a security restriction is violated, an exception is thrown.

Creating a Socket

- Under normal circumstances, a socket is connected to a machine and port when it is created.
  - Although there is a blank constructor that does not require a hostname or port, it is protected and can't be called from normal applications.
  - Furthermore, there isn't a connect() method that allows you to specify these details at a later point in time, so under normal circumstances the socket will be connected when created.
- If the network is fine, the call to a socket constructor will return as soon as a connection is established, but if the remote machine is not responding, the constructor method may block for an indefinite amount of time.
  - This varies from system to system, depending on a variety of factors such as the operating system being used and the default network timeout.
  - In mission-critical systems it may be appropriate to place such calls in a second thread, to prevent an application from stalling.

Using a Socket

- **void close() throws java.io.IOException**
  - *Closes the socket connection. Closing a connect may or may not allow remaining data to be sent, depending on the streams before closing a socket connection.*
- **InetAddress getInetAddress()**
  - *Returns the address of the remote machine that is connected to the socket.*
- **InputStream getInputStream() throws java.io.IOException**
  - *Returns an input stream, which reads from the application this socket is connected to.*
- **OutputStream getOutputStream() throws java.io.IOException**
  - *Returns an output stream, which writes to the application that this socket is connected to.*

Using a Socket (Cont.)

- **boolean getKeepAlive() throws java.net.SocketException**
  - *Returns the state of the SO_KEEPALIVE socket option.*
- **InetAddress getLocalAddress()**
  - *Returns the local address associated with the socket (useful in the case of multihomed machines).*
- **int getLocalPort()**
  - *Returns the port number that the socket is bound to on the local machine.*
- **int getPort()**
  - *Returns the port number of the remote service to which the socket is connected.*
- **int getReceiveBufferSize() throws java.net.SocketException**
  - *Returns the receive buffer size used by the socket, determined by the value of the SO_RCVBUF socket option.*
- **int getSendBufferSize() throws java.net.SocketException**
  - *Returns the send buffer size used by the socket, determined by the value of the SO_SNDBUF socket option.*

Using a Socket (Cont.)

- **int getSoLinger() throws java.net.SocketException**
  - *Returns the value of the SO_LINGER socket option, which controls how long unsent data will be queued when a connection is terminated.*
- **int getSoTimeout() throws java.net.SocketException**
  - *Returns the value of the SO_TIMEOUT socket option, which controls how many milliseconds a read operation will block for. If a value of 0 is returned, the timer is disabled and a thread will block indefinitely (until data is available or the stream is terminated).*
- **boolean getTcpNoDelay() throws java.net.SocketException**
  - *Returns "true" if the TCP_NODELAY socket option is set, which controls whether Nagle's algorithm is enabled.*

Using a Socket (Cont.)

- **void setKeepAlive(boolean onFlag) throws java.net.SocketException**
  - *Enables or disables the SO_KEEPALIVE socket option.*
- **void setReceiveBufferSize(int size) throws java.net.SocketException**
  - *Modifies the value of the SO_RCVBUF socket option, which recommends a buffer size for the operating system's network code to use for receiving incoming data. Not every system will support this functionality or allows absolute control over this feature. If you want to buffer incoming data, you're advised to instead use a BufferedInputStream or a BufferedReader.*
- **void setSendBufferSize(int size) throws java.net.SocketException**
  - *Modifies the value of the SO_SNDBUF socket option, which recommends a buffer size for the operating system's network code to use for sending incoming data. Not every system will support this functionality or allows absolute control over this feature. If you want to buffer incoming data, you're advised to instead use a BufferedOutputStream or a BufferedWriter.*

Using a Socket (Cont.)

- **static void setSocketImplFactory (SocketImplFactory factory) throws java.net.SocketException, java.io.IOException java.lang.SecurityException**
  - *Assigns a socket implementation factory for the JVM, which may already exist, or may violate security restrictions, either of which causes an exception to be thrown. Only one factory can be specified, and this factory will be used whenever a socket is created.*
- **void setSoLinger(boolean onFlag, int duration) throws java.net.SocketException java.lang.IllegalArgumentException**
  - *Enables or disables the SO_LINGER socket option (according to the value of the onFlag boolean parameter), and specifies a duration in seconds. If a negative value is specified, an exception is thrown.*

Using a Socket (Cont.)

- **void setSoTimeout(int duration) throws java.net.SocketException**
  - *Modifies the value of the SO_TIMEOUT socket option, which controls how long (in milliseconds) a read operation will block. A value of zero disables timeouts, and blocks indefinitely. If a timeout does occur, a java.io.IOInterruptedException is thrown whenever a read operation occurs on the socket's input stream. This is distinct from the internal TCP timer, which triggers a resend of unacknowledged datagram packets.*
- **void setTcpNoDelay(boolean onFlag) throws java.net.SocketException**
  - *Enables or disables the TCP_NODELAY socket option, which determines whether Nagle's algorithm is used.*

Using a Socket (Cont.)

- **void shutdownInput() throws java.io.IOException**
  - *Closes the input stream associated with this socket and discards any further information that is sent. Further reads to the input stream will encounter the end of the stream marker.*
- **void shutdownOutput() throws java.io.IOException**
  - *Closes the output stream associated with this socket. Any data previously written, but not yet sent, will be flushed, followed by a TCP connection-termination sequence, which notifies the application that no more data will be available (and in the case of a Java application, that the end of the stream has been reached). Further writes to the socket will cause an IOException to be thrown.*

Reading from and Writing to TCP Sockets

- Once a socket is created, it is connected and ready to read/write by using the socket's input and output streams.
- These streams don't need to be created; they are provided by the ***Socket.getInputStream()*** and ***Socket.getOutputStream()*** methods.
- A filter can easily be connected to a socket stream, to make for simpler programming.

Reading from and Writing to TCP Sockets (Example)

```
try{
        // Connect a socket to some host machine and port
        Socket socket = new Socket ( somehost, someport );
        // Connect a buffered reader
        BufferedReader reader = new BufferedReader ( new
        InputStreamReader ( socket.getInputStream() ) );
        // Connect a print stream
        PrintStream pstream = new PrintStream(
                        socket.getOutputStream() );
}
catch (Exception e){
        System.err.println ("Error – " + e);
}
```

Socket Options

- Socket options are settings that modify how sockets work, and they can affect (both positively and negatively) the performance of applications.
- Generally, socket options should not be changed unless there is a good reason for doing so, as changes may negatively affect application and network performance.
- The one exception to this caveat is the SO_TIMEOUT option.
  - Virtually every TCP application should handle timeouts gracefully rather than stalling if the application the socket is connected to fails to transmit data when required.

SO_KEEPALIVE Socket Option

- By default, no data is sent between two connected sockets unless an application has data to send.
  - This means that an idle socket may not have data submitted for minutes, hours, or even days in the case of long-lived processes.
- Suppose, however, that a client crashes, and the end-of-connection sequence is not sent to a TCP server.
  - Valuable resources (CPU time and memory) might be wasted on a client that will never respond.
- When the keepalive socket option is enabled, the other end of the socket is probed to verify it is still active.
  - However, the application doesn't have any control over how often keepalive probes are sent.

SO_KEEPALIVE Socket Option (Cont.)

- To enable ***keepalive***, the ***Socket.setSoKeepAlive(boolean)*** method is called with a value of "true" (a value of "false" will disable it).
- For example, to enable ***keepalive*** on a socket, the following code would be used.

```
// Enable SO_KEEPALIVE
someSocket.setSoKeepAlive(true);
```

- It should also be kept in mind that keepalive doesn't allow you to specify a value for probing socket endpoints.
  - A better solution than keepalive, and one that developers are advised to use, is to instead modify the timeout socket option.

SO_RCVBUF Socket Option

- The receive buffer socket option controls the buffer used for receiving data.
- Changes can be made to the size by calling the ***Socket.setReceiveBufferSize(int)*** method.
- For example, to increase the receive buffer size to 4,096 bytes, the following code would be used.

```
// Modify receive buffer size
someSocket.setReceiveBufferSize(4096);
```

SO_RCVBUF Socket Option (Cont.)

- Note that a request to modify the size of the receive buffer does not guarantee that it will change.
- For example, some operating systems may not allow this socket option to be modified, and will ignore any changes to the value.
- The current buffer size can be determined by invoking the ***Socket.getReceiveBufferSize()*** method.
- A better choice for buffering is to use a ***BufferedInputStream***/***BufferedReader***.

SO_SNDBUF Socket Option

- The send buffer socket option controls the size of the buffer used for sending data.
- By calling the ***Socket.setSendBufferSize(int)*** method, you can attempt to change the buffer size, but requests to change the size may be rejected by the operating system.

```
//Set the send buffer size to 4096 bytes
someSocket.setSendBufferSize(4096);
```

- To determine the size of the current send buffer, you can call the ***Socket.getSendBufferSize()*** method, which returns an int value.

```
// Get the default size
int size = someSocket.getSendBufferSize();
```

SO_LINGER Socket Option

- When a TCP socket connection is closed, it is possible that data may be queued for delivery and not yet sent (particularly if an IP datagram becomes lost in transit and must be resent).
- The linger socket option controls the amount of time during which unsent data may be sent, after which it is discarded completely.
- It is possible to enable/disable the linger option entirely, or to modify the duration of a linger, by using the ***Socket.setSoLinger(boolean onFlag, int duration)*** method:

```
// Enable linger, for fifty seconds
someSocket.setSoLinger( true, 50 );
```

TCP_NODELAY Socket Option

- This socket option is a flag, the state of which controls whether Nagle's algorithm (RFC 896) is enabled or not.
- Because TCP data is sent over the network using IP datagrams, a fair bit of overhead exists for each packet, such as IP and TCP header information.
- If only a few bytes at a time are sent in each packet, the size of the header information will far exceed that of the data.
- On a local area network, the extra amount of data sent probably won't amount to much, but on the Internet, where hundreds, thousands, or even millions of clients may be sending such packets through individual routers, this adds up to a significant amount of bandwidth consumption.

TCP_NODELAY Socket Option (Cont.)

- The solution is Nagle's algorithm, which states that TCP may send only one datagram at a time.
- When an acknowledgment comes back for each IP datagram, a new packet is sent containing any data that has been queued up.
- This limits the amount of bandwidth being consumed by packet header information, but at a not insignificant cost—network latency.
- Since data is being queued, it isn't dispatched immediately, so systems that require quick response times such as X-Windows or telnet are slowed.
- Disabling Nagle's algorithm may improve performance, but if used by too many clients, network performance is reduced.

TCP_NODELAY Socket Option (Cont.)

- Nagle's algorithm is enabled or disabled by invoking the ***Socket.setTcpNoDelay (boolean state)*** method.
- For example, to deactivate the algorithm, the following code would be used:

```
// Disable Nagle's algorithm for faster response times
someSocket.setTcpNoDelay(false);
```

- To determine the state of Nagle's algorithm and the TCP_NODELAY flag, the ***Socket.getTcpNoDelay()*** method is used:

```
// Get the state of the TCP_NODELAY flag
boolean state = someSocket.getTcpNoDelay();
```

SO_TIMEOUT Socket Option

- This timeout option is the most useful socket option.
- By default, I/O operations (be the file- or network-based) are blocking.
- An attempt to read data from an InputStream will wait indefinitely until input arrives.
- If the input never arrives, the application stalls and in most cases becomes unusable (unless multithreading is used).
- A more robust application will anticipate such problems and take corrective action.

SO_TIMEOUT Socket Option (Cont.)

- When the SO_TIMEOUT option is enabled, any read request to the InputStream of a socket starts a timer.
- When no data arrives in time and the timer expires, a ***java.io.InterruptedIOException*** is thrown, which can be caught to check for a timeout.
- What happens then is up to the application developer— a retry attempt might be made, the user might be notified, or the connection aborted.
- The duration of the timer is controlled by calling the ***Socket.setSoTimeout(int)*** method, which accepts as a parameter the number of milliseconds to wait for data.

SO_TIMEOUT Socket Option (Cont.)

- For example, to set a five-second timeout, the following code would be used:

```
// Set a five second timeout
someSocket.setSoTimeout ( 5 * 1000 );
```

- Once enabled, any attempt to read could potentially throw an ***InterruptedIOException***, which is extended from the ***java.io.IOException*** class.
- Since read attempts can already throw an IOException, no further code is required to handle the exception
  - However, some applications may want to specifically trap timeout-related exceptions, in which case an additional exception handler may be added.

SO_TIMEOUT Socket Option (Cont.)

```
try{
        Socket s = new Socket (...);
        s.setSoTimeout ( 2000 );
        // do some read operation ....
}
catch (InterruptedIOException iioe){
        timeoutFlag = true; // do something special like set a flag
}
catch (IOException ioe){
        System.err.println ("IO error " + ioe);
        System.exit(0);
}
```

SO_TIMEOUT Socket Option (Cont.)

- To determine the length of the TCP timer, the ***Socket.getSoTimeout()*** method, which returns an int, can be used.
- A value of zero indicates that timeouts are disabled, and read operations will block indefinitely.

```
// Check to see if timeout is not zero
if ( someSocket.getSoTimeout() == 0)
        someSocket.setSoTimeout (500);
```

Creating a TCP Client

```
import java.net.*
import java.io.*;
public class DaytimeClient{
    public static final int SERVICE_PORT = 13;
    public static void main(String args[]){
        // Check for hostname parameter
        if (args.length != 1){
            System.out.println ("Syntax - DaytimeClient host");
            return;
        }
        // Get the hostname of server
        String hostname = args[0];
        try{
            // Get a socket to the daytime service
            Socket daytime = new Socket (hostname, SERVICE_PORT);
```

Creating a TCP Client

```
            System.out.println ("Connection established");
            // Set the socket option just in case server stalls
            daytime.setSoTimeout ( 2000 );
            // Read from the server
            BufferedReader reader = new BufferedReader (
                new InputStreamReader(daytime.getInputStream()));
            System.out.println ("Results : " +reader.readLine());
            // Close the connection
            daytime.close();
        }
        catch (IOException ioe){
            System.err.println ("Error " + ioe);
        }
    }
}
```

ServerSocket Class

- A special type of socket, the ***server socket***, is used to provide TCP services.
- Client sockets bind to any free port on the local machine, and connect to a specific server port and host.
- The difference with server sockets is that they bind to a specific port on the local machine, so that remote clients may locate a service.
- Client socket connections will connect to only one machine, whereas server sockets are capable of fulfilling the requests of multiple clients.

ServerSocket Class (Cont.)

- Clients are aware of a service running on a particular port.
- Clients establish a connection, and within the server, the connection is accepted.
  - Multiple connections can be accepted at the same time, or a server may choose to accept only one connection at any given moment.
- Once accepted, the connection is represented as a normal socket, in the form of a ***Socket*** object.
- This ***ServerSocket*** object acts as a factory for client connections, you don't need to create instances of the ***Socket*** class yourself.
  - These connections are modeled as a normal socket, so you can connect input and output filter streams (or even a reader and writer) to the connection.

Creating a ServerSocket

- Once a server socket is created, it will be bound to a local port and ready to accept incoming connections.
- When clients attempt to connect, they are placed into a queue. Once all free space in the queue is exhausted, further clients will be refused.
- The simplest way to create a server socket is to bind to a local address, which is specified as the only parameter, using a constructor.

```
try{
        // Bind to port 80, to provide a TCP service (like HTTP)
        ServerSocket myServer = new ServerSocket ( 80 );
        // ......
}
catch (IOException ioe){
        System.err.println ("I/O error – " + ioe);
}
```

Creating a ServerSocket - Constructors

- **ServerSocket(int port) throws java.io.IOException, java.lang.SecurityException**
  - Binds the server socket to the specified port number, so that remote clients may locate the TCP service.
  - If a value of zero is passed, any free port will be used. However, clients will be unable to access the service unless notified somehow of the port number.
  - By default, the queue size is set to 50, but an alternate constructor is provided that allows modification of this setting.
  - If the port is already bound, or security restrictions (such as security polices or operating system restrictions on well-known ports) prevent access, an exception is thrown.
- **ServerSocket(int port, int numberOfClients) throws java.io.IOException, java.lang.SecurityException**
  - Binds the server socket to the specified port number and allocates sufficient space to the queue to support the specified number of client sockets.
  - If the port is already bound or security restrictions prevent access, an exception is thrown.

Creating a ServerSocket – Constructors (Cont.)

- **ServerSocket(int port, int numberOfClients, InetAddress address) throws java.io.IOException, java.lang.SecurityException**
  - Binds the server socket to the specified port number, and allocates sufficient space to the queue to support the specified number of client sockets.
  - This is an overloaded version of the ServerSocket(int port, int numberOfClients) constructor that allows a server socket to bind to a specific IP address, in the case of a multihomed machine.
  - For example, a machine may have two network cards, or may be configured to represent itself as several machines by using virtual IP addresses.
  - Specifying a null value for the address will cause the server socket to accept requests on all local addresses.
  - If the port is already bound or security restrictions prevent access, an exception is thrown.

Using a ServerSocket

- While the Socket class is fairly versatile, and has many methods, the Server Socket class doesn't really do that much, other than *accept connections* and a*ct as a factory for Socket objects* that model the connection between client and server.
- The most important method is the accept() method, which accepts client connection requests, but there are several others that developers may find useful.

Using a ServerSocket – Methods

- ***Socket accept() throws java.io.IOException, java.lang.SecurityException***
  - Waits for a client to request a connection to the server socket, and accepts it.
  - This is a blocking I/O operation, and will not return until a connection is made (unless the timeout socket option is set).
  - When a connection is established, it will be returned as a Socket object. When accepting connections, each client request will be verified by the default security manager, which makes it possible to accept certain IP addresses and block others, causing an exception to be thrown.
  - However, servers do not need to rely on the security manager to block or terminate connections—the identity of a client can be determined by calling the getInetAddress() method of the client socket.
- ***void close() throws java.io.IOException***
  - Closes the server socket, which unbinds the TCP port and allows other services to use it.

Using a ServerSocket – Methods (Cont.)

- ***InetAddress getInetAddress()***
  - Returns the address of the server socket, which may be different from the local address in the case of a multihomed machine (i.e., a machine whose localhost is known by two or more IP addresses).
- ***int getLocalPort()***
  - Returns the port number to which the server socket is bound.
- ***int getSoTimeout() throws java.io.IOException***
  - Returns the value of the timeout socket option, which determines how many milliseconds an accept() operation can block for. If a value of zero is returned, the accept operation blocks indefinitely.

Using a ServerSocket – Methods (Cont.)

- ***void implAccept(Socket socket) throws java.io.IOException***
  - This method allows ServerSocket subclasses to pass an unconnected socket subclass, and to have that socket object accept an incoming request.
  - Using the implAccept method to accept the connection, an overridden ServerSocket.accept() method can return a connected socket. Few developers will want to subclass the ServerSocket, and using this should be avoided unless required.
- ***static void setSocketFactory (SocketImplFactory factory) throws java.io.IOException, java.net.SocketException, java.lang.Security Exception***
  - Assigns a server socket factory for the JVM. This is a static method, and should be called only once during the lifetime of a JVM. If assigning a new socket factory is prohibited, or one has already been assigned, an exception is thrown.

Using a ServerSocket – Methods (Cont.)

- ***void setSoTimeout(int timeout) throws java.net.SocketException***
  - Assigns a timeout value (specified in milliseconds) for the blocking accept() operation.
  - If a value of zero is specified, timeouts are disabled and the operation will block indefinitely.
  - Providing timeouts are enabled, however, whenever the accept() method is called a timer starts. When the timer expires, a ***java.io.InterruptedIOException*** is thrown, which allows a server to then take further actions.

Accepting and Processing Requests from TCP Clients

- The most important function of a server socket is to accept client sockets.
- Once a client socket is obtained, the server can perform all the "real work" of server programming, which involves reading from and writing to the socket to implement a network protocol.

```
// Perform a blocking read operation, to read the next socket connection
Socket nextSocket = someServerSocket.accept();
// Connect a filter reader and writer to the stream
BufferedReader reader = new BufferedReader (new
        InputStreamReader (nextSocket.getInputStream() ) );
PrintWriter writer = new PrintWriter( new OutputStreamWriter
        (nextSocket.getOutputStream() ) );
```

Creating a TCP Server

```
import java.net.*;
import java.io.*;
public class DaytimeServer{
    public static final int SERVICE_PORT = 13;
    public static void main(String args[]){
        try{
            // Bind to the service port, to grant clients access to the TCP daytime
            //service
            ServerSocket server = new ServerSocket (SERVICE_PORT);
            System.out.println ("Daytime service started");
            // Loop indefinitely, accepting clients
            for (;;) {
                // Get the next TCP client
                Socket nextClient = server.accept();
```

Creating a TCP Server

```
            // Display connection details
            System.out.println ("Received request from " +
                    nextClient.getInetAddress() + ":" +nextClient.getPort() );
            // Don't read, just write the message
            OutputStream out = nextClient.getOutputStream();
            PrintStream pout = new PrintStream (out);
            // Write the current date out to the user
            pout.print( new java.util.Date() );
            // Flush unsent bytes
            out.flush();
            // Close stream
            out.close();
            // Close the connection
            nextClient.close();
        }
    }
```

Creating a TCP Server

```
        catch (BindException be){
            System.err.println ("Service already running on port " + SERVICE_PORT );
        }
        catch (IOException ioe){
            System.err.println ("I/O error - " + ioe);
        }
    }
}
```

Exception Handling: Socket-Specific Exceptions

- All socket-specific exceptions extend from ***SocketException***, so by simply catching that exception, you catch all of the socket-specific ones and write a single generic handler.
- In addition, ***SocketException*** extends from ***java.io.IOException*** if you want to provide a catchall for any I/O exception.
- **SocketException**
  - The ***java.net.SocketException*** represents a generic socket error, which can represent a range of specific error conditions. For finer-grained control, applications should catch its subclasses.

Exception Handling: Socket-Specific Exceptions (Cont.)

- **BindException**
  - The ***java.net.BindException*** represents an inability to bind a socket to a local port. The most common reason for this will be that the local port is already in use.
- **ConnectException**
  - The ***java.net.ConnectException*** occurs when a socket can't connect to a specific remote host and port. There can be several reasons for this, such as that the remote server does not have a service bound to that port, or that it is so swamped by queued connections, it cannot accept any further ones.

Exception Handling: Socket-Specific Exceptions (Cont.)

- **NoRouteToHostException**
  - The ***java.net.NoRouteToHostException*** is thrown when, due to a network error, it is impossible to find a route to the remote host.
  - The cause of this may be local (i.e., the network on which the software application is running), may be a temporary gateway or router problem, or may be the fault of the remote network to which the socket is trying to connect. Another common cause of this is that firewalls and routers are blocking the client software, which is usually a permenant condition.
- **InterruptedIOException**
  - The ***java.net.InterruptedIOException*** occurs when a read operation is blocked for sufficient time to cause a network timeout, as discussed earlier in the chapter. Handling timeouts is a good way to make your code more robust and reliable.

References

Chapter 6 of *Java™ Network Programming and Distributed Computing*, David Reilly and Michael Reilly.

Networks and Internet Programming

Multi-threading and Parallel Programming

Eng. Asma Abdel Karim
Computer Engineering Department

1

Outline

- Overview.
- Multi-threading in Java.
- Controlling Threads.
- Threads Priorities.
- Thread Synchronization.
- Inter-thread Communication.

Eng. Asma Abdel Karim
Computer Engineering Department

2

Overview

- Multi-threaded programming is an important concept in Java networking, as networking clients and servers must often perform several different tasks at a time.
  - For example, listening for incoming requests and responses, processing data, and updating the text or graphical user interface for the user.
- It is important for the developer to understand the differences between ***single-threaded programming***, ***multi-process programming***, and ***multi-threaded programming***.

3

Single-Threaded Programming

- Traditional software written in procedural languages is compiled into a machine-readable format, which is called machine code.
- This code is read by a central processing unit (CPU), which executes programming statements one after another, in a sequential manner.
- The time taken to execute each statement may vary (due to the nature of the operation, such as comparing two bytes for equality or adding two numbers together), but until a statement is completed, no further statements will run. This is ***single-threaded execution***.
- The chief advantage of this type of programming is its simplicity.
  - Developers can easily predict the state of a machine at any given moment in time.
  - It is guaranteed that a variable being accessed in a single-threaded environment will not be accessed or modified by another copy of the program, as only one copy of the program is running.

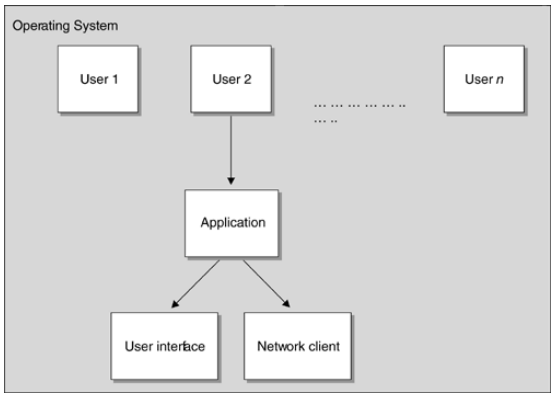
4

Multi-process Programming

- Each application runs as a process, with memory allocated for program code and data storage.
- Multiple processes would run on the same machine.
  - The operating system would allocate CPU time to each process, suspending a process when its time was up and allowing another to take its place.
  - Sometimes, a process will become blocked (waiting on I/O), or may voluntarily choose to yield its CPU time.
  - The operating system creates the illusion that these processes are running concurrently, by frequently switching from one process to another and sharing time between them (though not always equally).
- This type of multitasking is extremely important, as it means that one machine can share its CPU time across many users.

Multi-process Programming (Cont.)

- Programs themselves could create new processes, having one part of the program performing a task while another part does something else.



Multi-process Programming (Cont.)

- Although multi-process programming works well, there are disadvantages to its use.
- ***First***, when a process branches into two, there is overlap between the data storage of one process and another.
  - Because two copies of data are being kept, more memory than is needed is consumed.
- ***Second***, there isn't an easy way for one process to access and modify the data of another.
  - In Unix, Inter-Process Communication (IPC) is used, creating data pipes that allow a process to communicate with another.
  - Nonetheless, it is not as easy to design software that shares data in a multi-process environment as it is in a multi-threaded one.

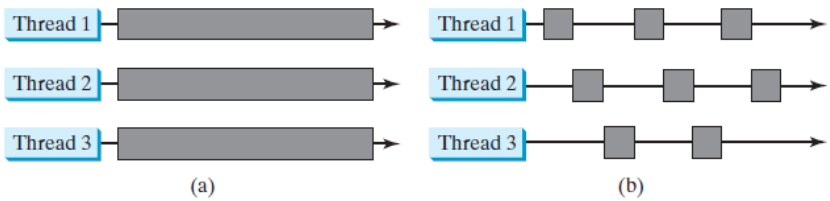
Multi-threaded Programming

- Multi-threaded programming requires a different way of looking at software.
- Rather than executing a series of steps sequentially, tasks are executed concurrently—that is, many tasks are performed at the same time, rather than one task having to finish before another can start.
- Multithreading, also known as multiple threads of execution, allows a program to have multiple instances of itself running, while using the same shared memory space and code.
  - Unlike multi-process programming, which uses separate memory address spaces, making communication between processes difficult.
  - An application can be performing many different tasks concurrently, and threads may access shared data variables to work collaboratively.

Multi-threaded Programming (Cont.)

- Unless you have more than one CPU, only a single thread can be running at any given moment in time.
- The operating system maintains a queue of threads and allocates CPU time to them.
- The process of determining which thread to run is called ***scheduling***.
- Not all operating systems allocate thread time fairly, but to give the operating system a guide, threads are allocated a ***priority level***.
- Since the choice of which thread is executed is up to the operating system and not the application, it becomes impossible to predict the order of execution, or how much CPU time will be given.

Multi-threaded Programming (Cont.)



(a) Threads running on multiple CPUs.

(b) Threads running on a single CPU.

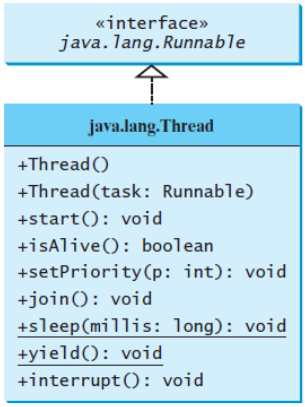
Multi-threaded Programming (Cont.)

- Careful attention must be paid to concurrent access and modification of data, to prevent data from becoming out of sync.
  - With careful design, however, data can be locked, which will prevent read access while write access occurs.
- Multi-threaded programming can be difficult to master, but the rewards that it offers are great.
  - Networking clients do not need to lock up the GUI if a network connection stalls, and servers can process multiple clients concurrently.
- Additionally, threads may use variables independently, and are not forced to share the same data.
  - A thread could, for example, declare its own set of variables that it does not make available to other threads (by marking them as private or protected), thus ensuring that an access conflict does not occur.

Multi-threading in Java

- Java provides exceptionally good support for creating and running threads and for locking resources to prevent conflicts.
- You can create additional threads to run concurrent tasks in the program.
- In Java, each task is an instance of the Runnable interface, also called a runnable object.
- A thread is essentially an object that facilitates the execution of a task.

Multi-threading in Java (Cont.)

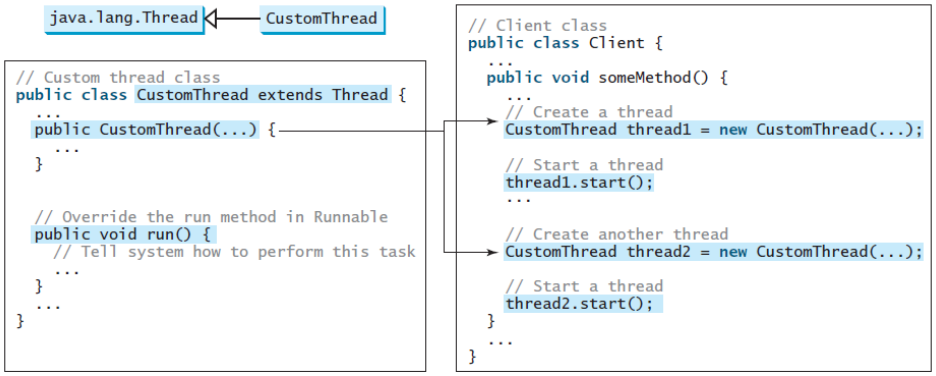


Creates an empty thread.	
Creates a thread for a specified task.	
Starts the thread that causes the `run()` method to be invoked by the JVM.	
Tests whether the thread is currently running.	
Sets priority p (ranging from 1 to 10) for this thread.	
Waits for this thread to finish.	
Puts a thread to sleep for a specified time in milliseconds.	
Causes a thread to pause temporarily and allow other threads to execute.	
Interrupts this thread.	

Creating Multi-threaded Applications with the Thread Class

- The ***java.lang.Thread*** class provides methods to start, suspend, resume, and stop a thread, as well as to control other aspects such as the priority of a thread or the name associated with it.
- The simplest way to use the ***Thread*** class is to extend it and override the ***run()*** method, which is invoked when the thread is first started.
- By overriding the ***run()*** method, a thread can be made to perform useful tasks in the background.
- Keep in mind that threads do not start running automatically at creation time. Instead, the ***Thread.start()*** method must be invoked. If it is not, the thread will not run.

Creating Multi-threaded Applications with the Thread Class (Cont.)



Creating Multi-threaded Applications with the Thread Class (Example)

```
public class ExtendThreadDemo extends java.lang.Thread{
   int threadNumber;
   public ExtendThreadDemo ( int num ){
      // Assign to member variable
      threadNumber = num;
   }
   // Run method is executed when thread first started
   public void run(){
      System.out.println ("I am thread number " + threadNumber);
      try{
         // Sleep for five thousand milliseconds (5 secs), to simulate work being done
         Thread.sleep(5000);
      }
      catch (InterruptedException ie) {}
      System.out.println (threadNumber + " is finished!");
   }
```

Creating Multi-threaded Applications with the Thread Class (Example)

```
// Main method to create and start threads
   public static void main(String args[]){
      System.out.println ("Creating thread 1");
      // Create first thread instance
      Thread t1 = new ExtendThreadDemo(1);
      System.out.println ("Creating thread 2");
      // Create second thread instance
      Thread t2 = new ExtendThreadDemo(2);
      // Start both threads
      t1.start(); t2.start();
   }
}
```

Creating Multi-threaded Applications with the Thread Class (Notes)

- The run() method is not invoked when the thread was created, only when the thread is started by invoking the start() method.
  - You can create threads in advance, and start them only when needed.
- Remember that the thread object only represents a thread—threads are in fact provided by the operating system itself.
  - When the start() method of a thread is called, it sends a request to launch a separate thread, which will call the run() method.
  - The main application does not call the run() method directly. Instead, it calls start() to perform this operation. If your application calls run() directly, it won't be running as a separate thread.

Creating Multi-threaded Applications with the Thread Class (Notes)

- The main method terminates once the two threads are started.
- There is no pause or sleep command issued in the main thread—yet the application doesn't terminate. It keeps on going until the two threads have finished their work and leave their run() method.
- When a normal thread (also referred to as a ***user thread***) is created, it is expected that it will complete its work and not shut down prematurely.
- The Java Virtual Machine (JVM) will not terminate until all user threads have finished, or until a call is made to the ***System.exit()*** method, which terminates the JVM abruptly.
- Sometimes, however, threads are only useful when other threads are running (such as the actual application, which will eventually terminate when the user is finished with it).
- We call these types of threads ***daemon threads***, as opposed to user threads. If only daemon threads are running, the JVM will automatically terminate.

Daemon Threads

- The following is a modification to the previous main method such that t1 and t2 are specified as daemon methods.

```
public static void main(String args[]){
    System.out.println ("Creating thread 1");
    // Create first thread instance
    Thread t1 = new ExtendThreadDemo(1);
    System.out.println ("Creating thread 2");
    // Create second thread instance
    Thread t2 = new ExtendThreadDemo(2);
    // Make both threads daemon threads
    t1.setDaemon(true); t2.setDaemon(true);
    // Start both threads
    t1.start(); t2.start();
    try{
       // Sleep for one second, to allow threads time to display first message
       Thread.sleep(1000);
    }
    catch (InterruptedException ie) {}
}
```

Daemon Threads (Notes)

- The first change makes both t1 and t2 daemon threads, by calling the setDaemon(boolean) method.
- If you need to change the state of a thread to either a daemon or a user thread, this must be done before the thread is started—its state cannot be changed once the thread is running.
- The second change introduces a slight pause, to allow the daemon threads time to display their first message.
  - When you recompile and run this example, you'll notice that the threads do not complete their work and display their final message. This is because there are no more user threads active once the main method finishes.
- The primary thread is always a user thread, never a daemon thread.

Creating Multi-threaded Applications with the Runnable Interface

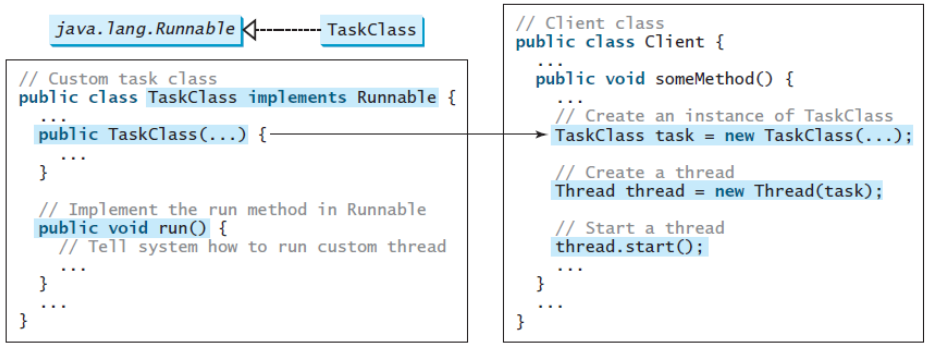
- While extending the ***Thread*** class is one way to create a multi-threaded application, it isn't always the best way.
- Remember, Java supports only single inheritance, unlike languages such as C++, which supports multiple inheritance.
- This means that if a class extends the ***java.lang.Thread*** class, it cannot extend any other class.
- A better way is often to implement the ***java.lang.Runnable*** interface.

Creating Multi-threaded Applications with the Runnable Interface (Cont.)

- The Runnable interface defines a single method, ***run()***, that must be implemented.
- Classes implement this interface to show that they are capable of being run as a separate thread of execution.
- The precise signature for the run method is as follows:

public void run ()

Creating Multi-threaded Applications with the Runnable Interface (Cont.)



Creating Multi-threaded Applications with the Runnable Interface (Cont.)

- The Runnable interface doesn't define any other methods, or provide any thread-specific functionality.
- Its sole purpose is to identify classes capable of running as threads.
- When an object implementing the Runnable interface is passed to the constructor of a thread, and the thread's start() method is invoked, the run() method will be called by the newly created thread.
- When the run() method terminates, the thread stops executing.

Creating Multi-threaded Applications with the Runnable Interface (Example)

```
public class RunnableThreadDemo implements java.lang.Runnable{
  public void run(){
    System.out.println ("I am an instance of the java.lang.Runnable interface");
  }
  public static void main(String args[]){
    System.out.println ("Creating runnable object");
    // Create runnable object
    Runnable run = new RunnableThreadDemo();
    // Create a thread, and pass the runnable object
    System.out.println ("Creating first thread");
    Thread t1 = new Thread (run);
    // Create a second thread, and pass the runnable object
    System.out.println ("Creating second thread");
    Thread t2 = new Thread (run);
    // Start both threads
    System.out.println ("Starting both threads");
    t1.start(); t2.start();
  }
}
```

Creating Multi-threaded Applications with the Runnable Interface (Example-Notes)

- When the example is compiled and run, two threads can be seen printing a message to the console.
- What is very different about this program, and the previous one, is that only one ***Runnable*** object was created, but two different threads ran it.
- Although there was no shared data in this example, in more complex systems, threads must share access to resources, to prevent modification while a resource is being accessed. This is achieved by synchronizing access to resources

Advantages of Using the Runnable Interface over Extending the Thread Class

1. As mentioned previously, an object is free to inherit from a different class.
2. The same Runnable object can be passed to more than one thread, so several concurrent threads can be using the same code and acting on the same data.
   - Though this use is not always advised, it can make sense in certain circumstances, providing that due care is taken to prevent conflicts over data access.
3. Carefully designed applications can minimize overhead, as creating a new Thread instance requires valuable memory and CPU time.
   - A Runnable instance, on the other hand, doesn't incur the same burden of a thread, and can still be passed to a thread at a later point in time to be reused and run again if necessary.

Controlling Threads
Interrupting a Thread

- Observant readers may have noticed that whenever a call to the ***Thread. sleep (int)*** method was made in earlier examples, an exception handler was used.
- This is because the ***sleep*** method puts a thread to sleep for a long period of time, during which it is generally unable to rouse itself.
- However, if a thread must be awakened earlier, interrupting a thread will awaken it; this is achieved by invoking the ***interrupt()*** method.
- Of course, this requires another thread to maintain a reference to the sleeping thread.

Controlling Threads
Interrupting a Thread (Example)

```
public class SleepyHead extends Thread{
   // Run method is executed when thread first started
   public void run(){
      System.out.println ("I feel sleepy. Wake me in eight hours");
      try{
         // Sleep for eight hours
         Thread.sleep( 1000 * 60 * 60 * 8 );
         System.out.println ("That was a nice nap");
      }
      catch (InterruptedException ie){
         System.err.println ("Just five more minutes....");
      }
   }
```

Controlling Threads
Interrupting a Thread (Example-Cont.)

```
// Main method to create and start threads
public static void main(String args[]) throws java.io.IOException{
   // Create a 'sleepy' thread
   Thread sleepy = new SleepyHead();
   // Start thread sleeping
   sleepy.start();
   // Prompt user and wait for input
   System.out.println ("Press enter to interrupt the thread");
   System.in.read();
   // Interrupt the thread
   sleepy.interrupt();
  }
}
```

Controlling Threads
Stopping, Suspending and Resuming a Thread

- The Thread class also contains the stop(), suspend(), and resume() methods.
- As of Java 2, these methods were deprecated (or outdated) because they are known to be inherently unsafe.

Controlling Threads
Yielding CPU Time

- Sometimes a thread might be waiting for an event to occur, or may be entering a section of code where releasing CPU time to another thread will improve either system performance or the user experience.
- For example:
  - After performing a calculation that should be displayed to the user and before starting another one.
  - While waiting for data to become available from an ***InputStream***, a thread might yield CPU time instead of going to sleep.
- In this situation, the static ***yield()*** method can be used instead of the ***sleep()*** method.

Controlling Threads
Yielding CPU Time (Cont.)

- For example, for the currently running thread to yield CPU time, the following method could be invoked:

Thread.yield();

- This is a static method that affects the currently running thread only—an application cannot yield the time of a specific thread.

Controlling Threads
Waiting Until a Thread is Dead

- Sometimes it is necessary to wait until a thread has finished its task.
  - For example, to retrieve the results of the task by invoking a method, or reading a member variable.
- To determine if a thread has died (i.e., if the run() method has finished), the ***isAlive()*** method, which returns a boolean value, can be invoked.
- But continually checking the value returned by this method (known as polling), and then sleeping or yielding, is a very inefficient use of CPU time.

35

Controlling Threads
Waiting Until a Thread is Dead (Cont.)

- A much better way is to use the ***join()*** method, which waits for a thread to die.
- There is also an overloaded version of this method, which takes as a parameter a long value. This version waits for a thread death or the specified number of milliseconds, whichever comes first.

36

Controlling Threads
Waiting Until a Thread is Dead (Example)

```
public class WaitForDeath extends Thread{
   // Run method is executed when thread first started
   public void run(){
      System.out.println ("This thread feels a little ill....");
      // Sleep for five seconds
      try{
         Thread.sleep(5000);
      }
      catch (InterruptedException ie) {}
   }
```

Controlling Threads
Waiting Until a Thread is Dead (Example-Cont.)

```
   // Main method to create and start threads
   public static void main(String args[]) throws java.lang.InterruptedException{
      // Create and start dying thread
      Thread dying = new WaitForDeath();
      dying.start();
      // Prompt user and wait for input
      System.out.println ("Waiting for thread death");
      // Wait till death
      try{
          dying.join();
      }
      catch(InterruptedException ex){}
      System.out.println ("Thread has died");
   }
}
```

Threads Priorities

- Java assigns every thread a priority.
- By default, a thread inherits the priority of the thread that spawned it.
- You can increase or decrease the priority of any thread by using the ***setPriority*** method, and you can get the thread's priority by using the ***getPriority*** method.
- Priorities are numbers ranging from 1 to 10.
- The Thread class has the int constants **MIN_PRIORITY**, **NORM_PRIORITY**, and **MAX_PRIORITY**, representing 1, 5, and 10, respectively.
- The priority of the main thread is **Thread.NORM_PRIORITY**.

Threads Priorities (Cont.)

- The JVM always picks the currently runnable thread with the highest priority.
- A lower-priority thread can run only when no higher-priority threads are running.
- If all runnable threads have equal priorities, each is assigned an equal portion of the CPU time in a circular queue.
  - This is called round-robin scheduling.

Thread Synchronization

- An important consideration when designing multi-threaded applications is conflict over access to data.
- If two threads are fighting for the same resource, and a mechanism to resolve access conflicts is not put into place, the integrity of the application is at stake.
- Built into the Java language are two mechanisms for preventing concurrent access to resources:
  - Method-level synchronization and,
  - Block-level synchronization.

Method-Level Synchronization

- Method-level synchronization prevents two threads from executing methods on an object at the same time.
- Methods that must be "thread-safe" are marked as synchronized.
- When a synchronized method of an object is invoked, a thread takes out an object lock, or monitor.
  - If another thread attempts to execute any synchronized method, it finds that it is locked, and enters a state of suspension until the lock on the object monitor is released.
- If several threads attempt to execute a method on a locked object, a queue of suspended threads will form.
  - When the thread that instituted the lock returns from the method, only one of the queued threads may access the object—the release of a monitor does not allow more than one object to take out a new monitor.
- One should note, however, that if a method is not synchronized and is executed while the object is locked, the thread will not block and the method can be run.

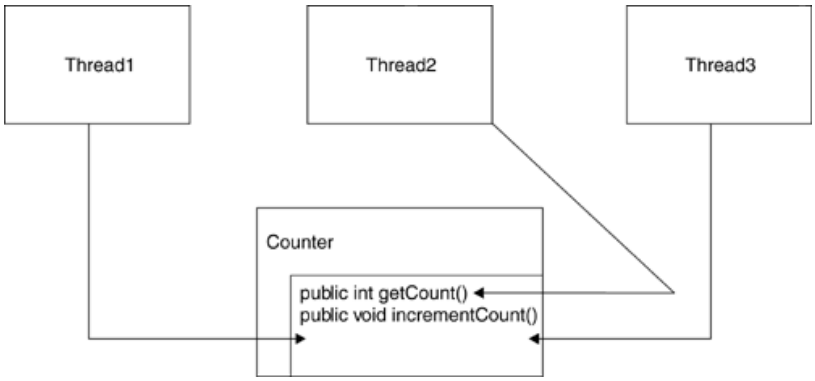
Method-Level Synchronization (Cont.)

- The ***synchronized*** keyword is used to indicate that a method should be protected by a monitor.
- Every method that could possibly be affected by concurrent access should be marked as synchronized. This keyword should be used sparingly, however, as it has a performance drawback.

```
public class SomeClass{
        public synchronized void changeData( ... ){
                ........
        }
        public synchronized Object getData ( ... ){
                ..........
        }
}
```

Method-Level Synchronization (Cont.)

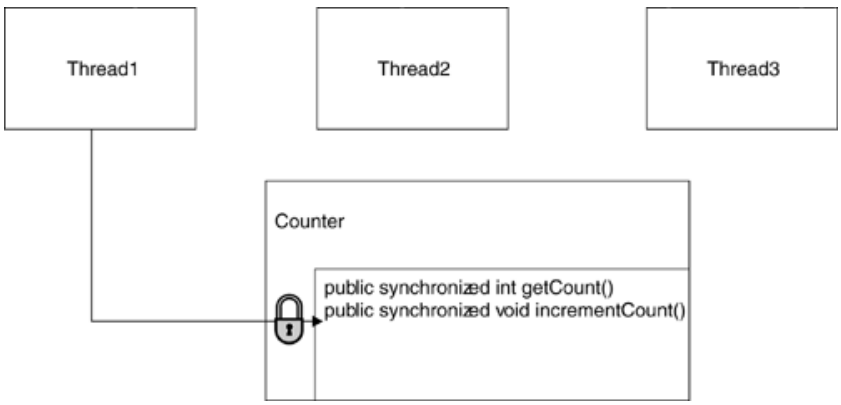
- Suppose we have a counter that can both be incremented and display a value.
- If the methods that provides access to the counter isn't thread-safe, and takes some time to complete, then two or more threads could access it at the same time.



Method-Level Synchronization (Cont.)

- The solution is to make the counter thread-safe, by synchronizing each method that performs a read or write operation.
- If a synchronized method is used, only one thread can update the value at any given moment.
  - The thread that first invokes a synchronized method locks the object's monitor, which is released only when that method terminates.
  - No other thread can access any synchronized method of the counter object.
- This restriction applies only to individual counter instances, and not the Counter class itself.

Method-Level Synchronization (Cont.)



Method-Level Synchronization (Example)

```
public class Counter{
   private int countValue;
   public Counter(){
      countValue = 0;
   }
   public Counter(int start){
      countValue = start;
   }
```

```
// Synchronized method to increase counter
public synchronized void increaseCount(){
   int count = countValue;
   try{
      Thread.sleep(5);
   }
   catch (InterruptedException ie) {}
   count = count + 1;
   countValue = count;
}
// Synchronized method to return counter value
public synchronized int getCount(){
   return countValue;
}
}
```

Method-Level Synchronization (Example-Cont.)

```
public class CountingThread implements Runnable{
   Counter myCounter;
   int countAmount;
   // Construct a counting thread to use the specified counter
   public CountingThread (Counter counter, int amount){
      myCounter = counter;
      countAmount = amount;
   }
   public void run()
   {
   // Increase the counter the specified number of times
      for (int i = 1; i <= countAmount; i++){
         // Increase the counter
         myCounter.increaseCount();
      }
   }
```

Method-Level Synchronization (Example-Cont.)

```
public static void main(String args[]) throws Exception{
    // Create a new, thread-safe counter
    Counter c = new Counter();
    // Our runnable instance will increase the counter
    // ten times, for each thread that runs it
    Runnable runner = new CountingThread( c, 10 );
    System.out.println ("Starting counting threads");
    Thread t1 = new Thread(runner);
    Thread t2 = new Thread(runner);
    Thread t3 = new Thread(runner);
    t1.start(); t2.start(); t3.start();
    // Wait for all three threads to finish
    t1.join(); t2.join(); t3.join();
    System.out.println ("Counter value is " + c.getCount() );
  }
}
```

Block-Level Synchronization

- Method-level synchronization is an effective means of preventing concurrent access to resources.
- But what if the resource has not been designed as thread-safe, and is a preexisting class that the developer cannot modify?
  - Such as a class in the Java API, or a third-party library.
- Block-level synchronization, in this case, is the best option.

Block-Level Synchronization (Cont.)

- Block-level synchronization uses the ***synchronized*** keyword, but instead of placing a lock around particular methods, a lock is placed around blocks of code.
- A block of code is synchronized against a particular object, and any thread attempting to enter that block of code is locked out, until the monitor for the specified object is released.
- The following code snippet shows the syntax for a synchronized block:

```
synchronized (Object o){
        ……
}
```

Block-Level Synchronization (Cont.)

- Block-level synchronization locks against a particular object.
- This means that multiple blocks can protect access to the same object, so block-level synchronization can be applied in thread code wherever an object is accessed or modified.

Block-Level Synchronization (Example)

```
public class SynchBlock implements Runnable{
   StringBuffer buffer;
   int counter;
   public SynchBlock(){
      buffer = new StringBuffer();
      counter= 1;
   }
```

Block-Level Synchronization (Example)

```
   public void run(){
    synchronized (buffer){
       System.out.print ("Starting synchronized block ");
       int tempVariable = counter++;
       // Create message to add to buffer, including linefeed
       String message = "Count value is : " + tempVariable +
             System.getProperty("line.separator");
       try{
          Thread.sleep(100);
       }
       catch (InterruptedException ie) {}
       buffer.append (message);
       System.out.println ("... ending synchronized block");
    }
   }
```

Block-Level Synchronization (Example)

```
public static void main(String args[]) throws Exception{
    // Create a new runnable instance
    SynchBlock block = new SynchBlock();
    Thread t1 = new Thread (block);
    Thread t2 = new Thread (block);
    Thread t3 = new Thread (block);
    Thread t4 = new Thread (block);
    t1.start(); t2.start(); t3.start(); t4.start();
    // Wait for all these threads to finish
    t1.join(); t2.join(); t3.join(); t4.join();
    System.out.println (block.buffer);
  }
}
```

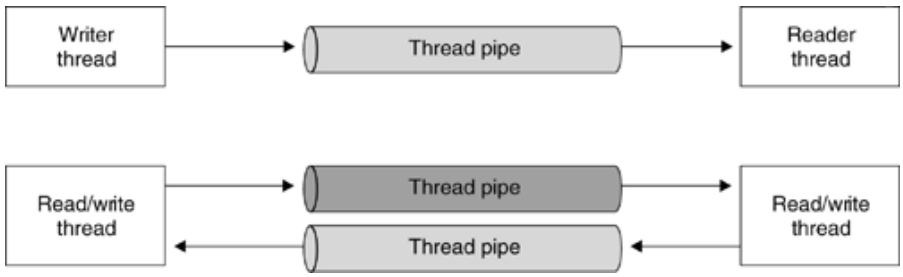
Inter-Thread Communication

- A design that requires no communication between threads lends itself to a far simpler implementation.
- However, sometimes it is necessary for threads to communicate with each other.
- Often, the type of communication will be fairly simple, such as reading or modifying a public member variable, or invoking an object method.
- Two good options for communication are:
  - Communication pipes and,
  - The wait()/notify() methods, which allow one thread to notify a waiting thread of an event.

Communication Pipes between Threads

- Like multi-process communication, which uses pipes to send data from one process to another, threads can also send data directly from one thread to another.
- This is achieved by using special types of input and output streams, which are linked together.
  - By passing either end of the pipe to another thread, that thread may listen to, or speak to, another thread.
  - In fact, there's no restriction preventing two pipes from being used—a thread could even have two-way communication with another.

Communication Pipes between Threads (Cont.)



Communication Pipes between Threads (Example)

```
import java.io.*;
public class PipeDemo extends Thread{
   PipedOutputStream output;
   // Create an instance of the PipeDemo class
   public PipeDemo(PipedOutputStream out){
      // Copy to local member variable
      output = out;
   }
```

Communication Pipes between Threads (Example-Cont.)

```
public static void main (String args[]){
   try{
      // Create a pipe for writing
      PipedOutputStream pout = new PipedOutputStream();
      // Create a pipe for reading, and connect it to output pipe
      PipedInputStream pin = new PipedInputStream(pout);
      // Create a new pipe demo thread, to write to our pipe
      PipeDemo pipedemo = new PipeDemo(pout);
      // Start the thread
      pipedemo.start();
      // Read thread data,
      int input = pin.read();
```

Communication Pipes between Threads (Example-Cont.)

```
        // Terminate when end of stream reached
        while (input != -1){
          // Print message
          System.out.print ( (char) input);
          // Read next byte
          input = pin.read();
        }
      }
      catch (Exception e){
        System.err.println ("Pipe error " + e);
      }
    }
```

Communication Pipes between Threads (Example-Cont.)

```
  public void run(){
    try{
      // Create a printstream for convenient writing
      PrintStream p = new PrintStream( output );
      // Print message
      p.println ("Hello from another thread, via pipes!");
      // Close the stream
      p.close();
    }
    catch (Exception e){
      // no code req'd
    }
  }
}
```

Notifying a Waiting Thread of an Event

- A common requirement in multi-threaded programming is that one thread cannot proceed until the completion of a task by another thread.
  - Sometimes a thread will be producing information or using resources. Other times, the order of execution is important, and a task cannot take place before another has completed.
- While it is possible for one thread to wait until another has died (thus indicating that the work was completed) by using the Thread.join() method, what if a thread performs an ongoing task and never terminates?

63

Notifying a Waiting Thread of an Event (Cont.)

- The solution is to notify other threads that a task has been completed.
  - Threads wait until they are notified, and notification can be a repeated process (with several cycles of waiting and notifying).
- This allows threads to synchronize their actions and communicate that a critical event has occurred, with-out requiring the extra complexity of pipe-based communication or invoking methods.
  - Sometimes a thread may not even know exactly which threads are waiting for it to complete, so a special type of notification is used.

64

Notifying a Waiting Thread of an Event (Cont.)

- Every Java object inherits from the java.lang.Object class:
  - The ability to maintain a queue of threads waiting for an object lock to be released, and
  - To notify one or more waiting threads that the object is freed.
- This provides a great way to notify a thread that an event has occurred, and for threads to wait indefinitely (or for a limited amount of time) until notification is sent.

Notifying a Waiting Thread of an Event (Cont.)

- To have threads wait for an indefinite amount of time, the Object.wait() method is used.
  - An overloaded version of this method also exists, which waits for a limited amount of time (specified in milliseconds).
- Before the wait() method may be invoked, however, the thread must hold a lock on the object's monitor.
  - To gain a lock on an object's monitor, it must be executing a synchronized method or using a synchronized block.
  - When the lock is released, another thread can obtain it—without this, the thread will wait indefinitely.

Notifying a Waiting Thread of an Event (Cont.)

- Once the wait() method is executed, the monitor is released and the thread is suspended until a call is made to the Object.notify() or Object.notifyAll() method.
- To awaken waiting threads, another thread may call either method.
  - However, the notify() method will only notify a single thread, even if multiple threads are waiting.
  - There is no choice over which thread is awakened, either (this is determined by the JVM implementation, so you cannot rely on, for example, a FIFO queue).
  - It is advised that the notifyAll() method is used if you want to notify a specific thread.

Notifying a Waiting Thread of an Event (Example)

```
public class WaitNotify extends Thread{
   public static void main(String args[]) throws Exception{
      Thread notificationThread = new WaitNotify();
      notificationThread.start();
      // Wait for the notification thread to trigger event
      synchronized (notificationThread){
         notificationThread.wait();
      }
      // Notify user that the wait() method has returned
      System.out.println ("The wait is over");
   }
```

Notifying a Waiting Thread of an Event (Example)

```
public void run(){
    System.out.println ("Hit enter to stop waiting thread");
    try{
       System.in.read();
    }
    catch (java.io.IOException ioe){}
    // Notify any threads waiting on this thread
    synchronized (this){
       this.notifyAll();
    }
  }
}
```

References

Chapter 7 of *Java™ Network Programming and Distributed Computing*, David Reilly and Michael Reilly.

Chapter 30 *of Introduction to Java Programming by Y. Daniel Liang, 10th edition.*

Networks and Internet Programming

HyperText Transfer Protocol

Eng. Asma Abdel Karim
Computer Engineering Department

1

Outline

- What is HTTP?
- How Does HTTP Work?
- Web Clients
- Web Servers
- HTTP and Java.
- Common Gateway Interface (CGI).

Eng. Asma Abdel Karim
Computer Engineering Department

2

What is HTTP?

- HTTP is an application-level protocol that uses the Transmission Control Protocol (TCP) as a transport mechanism.
- HTTP provides access to documents and files stored on a Web server.
- Web browsers use HTTP to request files or dynamically generated content produced by CGI scripts, and other server-side applications.
- Hypertext documents contain hyperlinks, which are links to other hypertext documents and files.
- The World Wide Web is a collection of hypertext documents, stored on a wide variety of Web servers and accessed by an even larger number of Web browsers and HTTP clients.

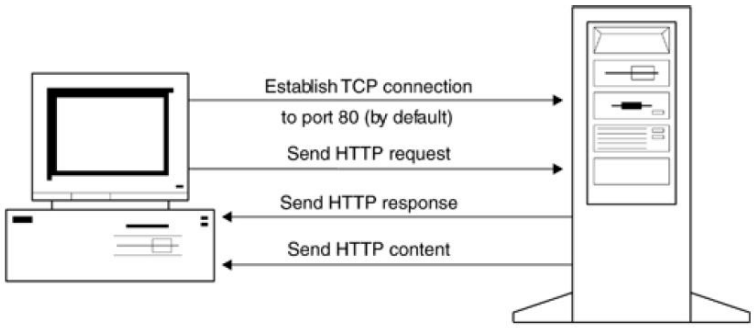
3

How Does HTTP Work?

- When an HTTP client, such as a Web browser or a search engine, needs to access a file, it establishes a TCP connection to the Web server (which, by default, uses TCP port 80 for communication).
- The client sends a request for a particular file, and receives an HTTP response, which will often include the contents of the file.
- The response includes:
  - A status code: indicating the success or failure of a request.
  - Some HTTP header information such as:
    - The length of the content and its type, and
    - If appropriate, the file contents.

4

How Does HTTP Work? (Cont.)



5

How Does HTTP Work? (Cont.)

```
GET /index.html HTTP/1.0
```
Client HTTP GET Request

```
HTTP/1.0 200 OK
```
←Status line

```
Last-Modified: Monday, 27-Dec-99
22:14
Content-Type: text/html
Content-Length: 1639
```
←Header (composed of several fields)

GMT

```
<HTML>
<HEAD>
   <TITLE> Our homepage </TITLE>
 </HEAD>
<BODY>
.....    // BODY GOES HERE
</BODY>
</HTML>
```
←Entity body

6

Web Clients

- A Web client, connecting to the Web server via a TCP socket, will send an HTTP request and then read back a server response.
- Under HTTP/1.0, there are three types of requests that a client application can issue to a Web server:

1. `GET`
2. `HEAD`
3. `POST`

- The most common HTTP request is a `GET` request, which fetches a resource from the Web server.
- Each request can also include header fields, which give the server more information about the client.

Web Clients
GET Request Method

- When a client needs to retrieve a resource from a Web server, it uses the GET request.
- The `GET` request takes two parameters: the pathname of the resource and the version of HTTP being used.
- CGI parameters can also be passed using the `GET` method.
  - While often only files are requested, it is possible to invoke a CGI script or server-side application.
- For example:
  - `GET /index.html HTTP/1.0`
  - `GET /images/banner.gif HTTP/1.0`
  - `GET /links.html HTTP/1.0`

Web Clients
HEAD Request Method

- Sometimes a client will be interested in information about a resource but not the resource itself.
  - For example, if a large file is already cached, the client may want to know if it has been modified recently. If so, a new copy would be downloaded, and if not, the cached version would be used.
  - Some clients may be unable to process certain types of content, so they may want to know the MIME content type of the resource.
- In these situations, a `HEAD` request can be made.
  - The `HEAD` request takes the same parameters as a `GET` request, and will return a normal HTTP response with information about the resource stored in header fields.
  - However, no actual content will be returned, conserving network bandwidth.
- An example of a HEAD request is as follows:

```
HEAD /files/averybigfile.zip HTTP/1.0
```

Web Clients
POST Request Method

- One of the great advantages of the Web is that Web sites can be interactive.
- The earliest form of interactivity came in the form of CGI scripts, written in languages such as Perl.
  - CGI is an acronym for the Common Gateway Interface, which defines a standard mechanism for communicating data from browser to server.
  - CGI scripts, and later, server-side applications such as Java servlets, make it possible for users to interact with server-side applications.
- There must be a way for the browser to pass information from the user to the Web server.
  - This is where the Common Gateway Interface comes in. CGI parameters are used for this purpose.

Web Clients
POST Request Method (Cont.)

- While it is possible to use the `GET` request to encode such parameters, limitations on the length of a URL are restrictive.
- A better way is to use the `POST` method, which allows clients to send much more information (including large files) to server-side scripts and applications.
- The `POST` request method has a format similar to a `GET` request.
- After header fields have been sent, however, the client sends an entity body, which contains CGI parameters and other information.
- When the entity body is complete, the server will process the information and output an HTTP response.

Web Clients
POST Request Method (Cont.)

```
POST /cgi-bin/information.pl HTTP/1.0
Content-type: application/x-www-form
-urlencoded
Content-length: 21
User-Agent: SomeBrowser/1.0
Name=David%20Reilly&answer=yes
HTTP/1.0 200 OK
Last-Modified: Monday, 27-Dec-99 22:14
GMT
Content-Type: text/html
Content-Length: 4855
<HTML>
<HEAD>
   <TITLE> Thanks for your feedback
</TITLE>
</HEAD>


<BODY>
.....    // BODY GOES HERE
</BODY>
</HTML>
```

- **Client HTTP `POST` request**
- **Header (composed of several fields)**
- **Entity body**

⟵HTTP Server Response

⟵Entity body

Web Clients
Client Request Header Fields

- Attached to any request may be header information and a number of optional fields.
- Header fields are sent after the request line, and are terminated with a carriage return/linefeed.
- If no headers are specified, the carriage return/linefeed must still be sent.
- The HTTP specification defines several header fields that can be used, and clients are free to add their own.
- However, not every server will honor such fields. If a field is not supported, it will be ignored—no error condition should be generated.

Web Clients
"Cookie" Field

- Persistent client-side objects (cookies) are small pieces of data sent by a Web server and echoed back on every subsequent request.
- Cookies are used for tracking client requests and customizing server output for individual users.
- Not every client supports cookies, and cookie support can be disabled for privacy and security in most clients.
- An example of this field is:

```
Cookie: secret_id=553235996
```

Web Clients
"From" Field

- The "From" field specifies the e-mail address of the user in control of the client.
- Most Web browsers do not divulge this information (much to the dismay of direct-mail marketers); however, specialized or experimental HTTP clients may.
- For example, a search engine that sends out requests to remote Web sites may include a contact e-mail address, to help establish contact if the search engine runs wild and requests too many pages in a given period.
- An example of this field is:

```
From: myemail@mydomain.com
```

Web Clients
"If-Modified-Since" Field

- A client that has already requested a resource can use this header field to check whether the resource has been updated.
- For example, a Web browser that caches pages may send the "If-Modified-Since" field, specifying the date when the resource was last requested.
- If the resource has been updated, the server will output a normal response and a normal entity body.
- If, however, it has not been updated, the server will issue a response with a status code of 304 and no entity body. The client then knows to use the cached version.
- An example of this field is:

```
If-Modified-Since: Tue, 27 Oct 1998, 09:00:00 GMT
```

Web Clients
"Referer" Field

- This is an extremely important field for the server and is often stored for statistical purposes.
- The "Referer" field specifies the URL that linked to the request URL.
- If an HTTP client follows a link to a new page, the "Referer" field will specify the page that linked to it.
- This information is useful for Web masters, as it shows which pages are linking to a site and which search engine queries delivered specific users.
- Most Web browsers send this field automatically, without the ability to disable it.
- An example is:
- `Referer: http://www.davidreilly.com/links.html`

Web Clients
"User-Agent" Field

- The "User-Agent" field identifies the type of HTTP client that is making the request.
- This information can be collected for statistical purposes or used to customize the response issued by the server.
- For example, an opportunistic Web master might output many random keywords to search engines in an effort to boost the traffic on his or her site. "Real" users would see the original page, since the server would read a browser identification string and not a search engine identification string.
  - Such behavior on the part of a Web master is frowned upon, however, and would eventually result in penalization, or blacklisting of the site.
- Developers of HTTP clients should always include this field, to identify the software that is making the request (by default, Java sends a field that identifies HTTP requests as being made by a Java application).
- Some sites, strange as it may seem, use this information to blacklist certain HTTP clients such as search engines, so it is important that a legitimate "User-Agent" field is sent.
- By convention, browser applications begin their identification string with the keyword Mozilla, which was the identification string of the Netscape browser.
- An example of this field is:

```
User-Agent: Mozilla (MyNewBrowser/1.0)
```

Web Servers

- When a client connects to a Web server, it will send an HTTP request that the server will read and process. The server returns a response.
- That response is made up of the following components:
  - A status line, with a numerical status code and human-readable text message.
  - A response header, with one or more header fields followed by a blank line.
  - An entity body (optional), which contains the contents of a file or server-side output.

Web Servers
Status Line

- The status line indicates whether a request could be performed successfully.
- If a request could not be completed, it also gives an indication of the reason.
- It does so by using a three-digit numerical code.
  - The first digit corresponds to an error group.
  - The two remaining digits indicate a precise error condition.
- Each group represents a generic error state.
- Clients can respond to the first digit (knowing only that an error occurred or that a request was accepted), or they can act based on the type of error that occurred.

Web Servers
Status Line (Cont.)

Status Group	Error Group Name
1xx	Informational
2xx	Successful
3xx	Redirection
4xx	Client Error
5xx	Server Error

Web Servers
Status Line (Cont.)

- Informational (1xx)
  - The informational set of status codes is rarely used.
  - These status codes are not a valid response for any HTTP/1.0 request and so should be used only in experimental systems, not for production systems.
- Successful (2xx)
  - This is, under ideal circumstances, the most common set of status codes returned.
  - This indicates that a request was processed and completed without any errors.
  - Although there are several status codes in this group, the most common will be the 200 status code

Web Servers
Status Line (Cont.)

- Redirection (3xx)
  - When a resource moves to a new location or a new server, redirection is occasionally used.
  - When a status code from the redirection group is issued, the client should look for a "Location" header field giving the new location of the resource.
  - The entity body of the response may also include a hyperlink to the new resource, in the event that the HTTP client is unable to automatically follow redirect requests.
  - There is also one circumstance in which a redirection status code will be sent, but where no redirection location is specified.
    - If a client issues a conditional request, by specifying an "If-Modified-Since" field in the request header, a 304 status code will be returned, indicating that the resource has not been modified and that the previous cached version should be used. No entity body is returned in this situation.

23

Web Servers
Status Line (Cont.)

- Client Error (4xx)
  - When an HTTP client sends an incorrect request, a status code from the client error group will be sent.
  - Reasons for the error can vary, from a bad request, to an invalid URL, to a URL that is forbidden for that client's IP address.
  - The most common of all client error status codes is 404, indicating that the resource was not found.
  - Though this is classed as a client error, sometimes a resource is deleted from the server, making invalid a URL that clients have previously accessed.
- Server Error (5xx)
  - Servers themselves are not impervious to error conditions.
  - When a server is overloaded, for example, it can issue a 503 status code.
  - A server-side CGI script or servlet could malfunction, causing a 501 status code to be issued.
  - Generally, however, servers are reasonably stable—there are more likely to be client errors than server errors.

24

Web Servers
Server Response Header Fields

- The response header is composed of two groups of header fields:
  - Response-Header field,
  - or Entity-Header field.
- In actual practice, the header group names are irrelevant, as the headers are sent together and only the field name is of interest to a client.
- Each header field is optional and will not be present in every request.
- However, some fields will be present in almost all situations, such as the "Content-Type" and "Last-Modified" fields.
- Covering every possible field would be impossible, as servers can add custom fields. However, the most frequently encountered fields described in the HTTP specification are worth being aware of when writing HTTP clients.

Web Servers
Server Response Header Fields

"Location" Field

- The location field specifies a URL to a resource.
- In the case where a redirection status code is specified, the client should fetch the content specified in this header field. For example, in response to a URL request to http://davidreilly.com/, a redirection to http://www.davidreilly.com/ is made using:

Location: http://www.davidreilly.com/

"Server" Field

- The "Server" field gives information about the server vendor and server version number.
- This information often isn't that useful to the client and, as the HTTP specification warns, could represent a security risk.
- If a server returns this field, and a known security flaw in that vendor/version combination exists, it may open the server to hostile attack.
- An example of this field is:

Server: MyServer v1.05

Web Servers
Server Response Header Fields

"Content-Length" Field

- This field indicates the number of bytes of the entity body.
- Although not strictly necessary, since the client will stop reading when the entity body terminates, this information can be useful when dealing with large file downloads. At the beginning of the transaction, the client can give the user an estimate of how much content needs to be downloaded, and the time remaining.
- If a connection terminates prematurely, the client can detect that the file was not completely downloaded, and attempt the request again.
- An example of this field is:

```
Content-Length: 5934
```

27

Web Servers
Server Response Header Fields

"Content-Type" Field

- This field specifies the MIME content type of the entity body.
- The MIME content type is divided into two sections, separated by a "/" character. The first section indicates the general type of the content (e.g., text, image, application file); the second section indicates the specific type.
- For example:
  - The content type of a Web page is text/html, whereas a plain text document is text/plain.
  - An image in GIF format is of type "image/gif," whereas a JPEG is of type "image/jpeg."
- The Internet Assigned Number Authority defines MIME content types, and it is best not to assign arbitrary types in production systems.
- An example is:

```
Content-Type: image/png
```

28

Web Servers
Server Response Header Fields

"Expires" Field

- When a server wishes to prevent caching of a resource for too long a period, it can specify a "use by“ date for a resource.
- For example, the front page of a news site could specify one hour into the future, to prevent proxy servers and HTTP clients from caching the content longer than an hour.
- Some content may also be marked as not cacheable, using the "Pragma" header field. However, the "Expires" field allows caching for a specified period of time.
- An example of this field is:

```
Expires: Thu, 12-Jan-2001 10:00:00 GMT
```

"Last-Modified" Field

- The "Last-Modified" field indicates when a resource was last changed or last updated.
- This information may be useful to proxy servers and clients, to prevent the downloading of large files that are already cached and have not changed since the last request.
- An example of this field is:

```
Last-Modified: Fri, 22-Feb-1998 15:23:11 GMT
```

Web Servers
Server Response Header Fields

"Pragma" Field

- Sometimes a server may wish to prevent caching of a resource entirely.
- For example, a page that changes every few minutes (such as a stock-price information page) or that contains sensitive information that would best not be cached by an intermediary proxy server may be marked with the "Pragma" field.
- The "Pragma" field can also be used to indicate other restrictions and information on the behavior of clients, though clients are not guaranteed to support such restrictions.
- An example is:

```
Pragma: nocache
```

Web Servers
Server Response Header Fields

"Set-Cookie" Field

- Although not defined in the original HTTP specification, cookies have been quickly adopted by Web browser manufacturers and the Web developer community.
- When a server-side application wants to send a cookie to the browser, it adds a "Set-Cookie" field to the HTTP response.
- Subsequent requests by that browser will include the cookie, so that the client can be tracked.
- Not all browsers have cookies enabled, and server-side applications should be aware that there is no error message sent if a browser does not accept cookies.
- An example of this field is:

```
Set-Cookie: secret_id=553235996; domain=mysite.com; path=/
```

Web Servers
Entity Body

- The entity body is the stream of bytes that form the actual content of the requested resource.
- This content may be:
  - Static (a file whose content changes infrequently or not at all) or
  - Dynamic (a custom server-side response to the client request).
- Meta-information about the entity body is contained in entity header fields such as "Content-Type" and "Content-Length."

HTTP and Java

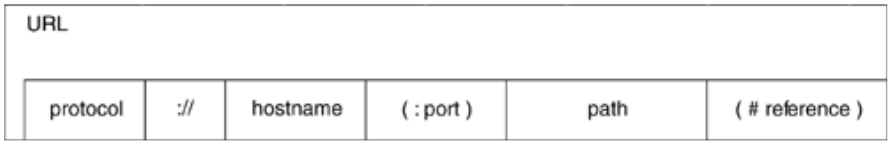
- Java provides extremely good support for the Hyper Text Transfer Protocol.
- While developers are free to write their own HTTP implementations using TCP sockets, the java.net package provides several classes that offer HTTP functionality:
  - `java.net.URL`
  - `java.net.URLConnection`
  - `java.net.HttpURLConnection`

URL Class

- The ***URL*** class represents one of the most frequently used address types of the Internet, the Uniform Resource Locator (URL).
- URLs can point to files, Web sites, ftp sites, newsgroups, email addresses, and other resources.
- Some fictitious examples of non-Web URLs are:
  - ftp://records.area51.mil/roswell/subjects/autopsy/
  - telnet://localhost:8000/
  - mailto:president@whitehouse.gov?subject=My%20Opinion
- In the context of HTTP, we'll be dealing with URLs that point to a Web site, but it is important to remember that other network protocols also use URLs.

URL Class (Cont.)

- The URL is composed of several components, each of which can be parsed by the *URL* class and returned separately.
- Some components (namely the port and the reference fields) are optional, and will not be present in many URLs.
- As mentioned earlier, CGI parameters can also be included as part of the path field.



Creating a URL

- The *URL* class can be used to parse URLs, or as an identifier of a remote resource that can be employed (in conjunction with the other Java HTTP classes) to retrieve that resource.
- There are six constructors for the *URL* class; the choice of which to use depends largely on how much control you require over the URL.
- For most situations, the following constructor will be used:

  URL(String url_str) throws java.net.MalformedURLException

  – creates a URL object based on the string parameter. If the URL cannot be correctly parsed, a MalformedURLException will be thrown.

Using a URL

- The *URL* class provides the following methods to parse a URL and extract individual components (such as the protocol or the hostname of the URL), as well as to open an HTTP connection to the resource that it specifies.
- **boolean equals(Object object)**: compares two URLs for equality. If the object is not an instance of the *URL* class, or if the object does not point to an identical resource, a value of "false" is returned.
- **Object getContent() throws java.io.IOException**:
  - Retrieves the contents of the resource located at the URL.
  - The type of object returned will vary, depending on the MIME content type of the remote resource and the available content handlers (classes responsible for processing and retrieving objects from a *URLConnection*).
  - This method is shorthand for calling the *openConnection()* method, which returns a *URLConnection*, and then invoking the *getContent()* method upon the *URLConnection* that was returned.
  - As a network connection will be established, an *IOException* may be thrown.

Using a URL (Cont.)

- **String getProtocol()**: returns the protocol component of a URL.
- **String getHost()**: returns the hostname component of a URL.
- **String getPort()**: returns the port component of a URL. This is an optional component, and if not present a value of –1 will be returned.
- **String getFile()**: returns the pathname component of a URL.
- **String getRef()**: returns the reference component of a URL. This is an optional component, and a reference may not be present. A null value will be returned if no reference was specified.
- **public int hashCode()**: returns an identifier for a *URL* object, for the purpose of hash table indexing.
- **URLConnection openConnection()**: returns a *URLConnection* object, which can be used to establish a connection to the remote resource. The name of this method can be deceiving, though, as no connection will be established until further methods of the *URLConnection* object are invoked.

Using a URL (Cont.)

- **InputStream openStream() throws java.io.IOException**: establishes a connection to the remote server where the resource is located, and provides an *InputStream* that can be used to read the resource's contents. This method provides a quick and easy way to retrieve the contents of a URL, without the added complexity of dealing with a *URLConnection* object.
- **boolean sameFile (URL url):** compares two URLs for equality, similar to that of the *equals(Object)* method. However, only the protocol, hostname, port, and pathname fields are compared—the reference field of the URL is excluded. While the *equals(Object)* method checks that a URL points to the same place in the same file, the *sameFile(URL)* method does not test where in the file the URL points to.
- **String toString()**: returns a String representation of a URL. There is no difference between this method and the *toExternalForm()* method.
- **String toExternalForm():** returns a String representation of a URL. There is no difference between this method and the *toString()* method.

Parsing with the URL Class

```
import java.net.*;
public class URLParser{
   public static void main(String args[]){
     int argc = args.length;
     // Check for valid number of parameters
     if (argc != 1){
              System.out.println ("Syntax : Java URLParser url");
              return;
     }
     try{
              // Create an instance of java.net.URL
              URL myURL = new URL ( args[0] );
              System.out.println ("Protocol : " +
              myURL.getProtocol() );
              System.out.println ("Hostname : " + myURL.getHost() );
              System.out.println ("Port : " + myURL.getPort() );
              System.out.println ("Filename : " + myURL.getFile() );
              System.out.println ("Reference: " + myURL.getRef() );
     }
     // MalformedURLException indicates parsing error
     catch (MalformedURLException mue){
              System.err.println ("Unable to parse URL!");
              return;
     }
    }
 }
```

Retrieving a Resource with the URL Class

- There are two URL methods that can assist in retrieving the contents of a remote resource:
  1. **InputStream URL.openStream()**
  2. **URLConnection URL.openConnection();**
- For greater control over how the request is made, a URLConnection object created by invoking the *URL.openConnection()* method would be used.
- In many situations, however, a simpler way to retrieve the contents of a resource is called for. The *openStream()* method returns an *InputStream*, which makes reading a resource simple.

Retrieving a Resource with the URL Class

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

public class FetchURL {

    public static void main(String args[]) throws Exception {
        int argc = args.length;
        if (argc != 1) {
          System.out.println("Syntax : java FetchURLConnection url");
          return;
        }
        try {
          // Create an instance of java.net.URL
          URL myURL = new URL(args[0]);
          // Fetch the content, and read from an InputStream
          InputStream in = myURL.openStream();
          // Buffer the stream, for better performance
          BufferedInputStream bufIn = new BufferedInputStream(in);
          // Repeat until end of file
          for (;;) {
            int data = bufIn.read();
            // Check for EOF
            if (data == -1) {
              break;
            } else {
              System.out.print((char) data);
            }
          }
          // Pause for user
          System.out.println();
          System.out.println("Hit enter to continue");
          System.in.read();
        } // MalformedURLException indicates parsing error
        catch (MalformedURLException mue) {
          System.err.println("Unable to parse URL!");
          return;
        } // IOException indicates network or I/O error
        catch (IOException ioe) {
          System.err.println("I/O Error : " + ioe);
          return;
        }
    }
}
```

URLConnection Class

- The ***URLConnection*** class is used to send HTTP requests and read HTTP responses.
- ***URLConnection*** has methods that allow you to:
  - Connect to a Web server
  - Set request header fields
  - Read response header fields
  - Read the contents of the resource.
- There are no public constructors for the ***URLConnection*** class.
- Instead, you should call the ***URL.openConnection()*** method, which will return a ***URLConnection*** instance.

```
URL url = new URL ( some_url );
URLConnection connection = url.openConnection();
```

Retrieving a Resource with the URLConnection Class

- While the ***URL*** class does allow you to retrieve a resource by using the ***URL.openStream()*** method, information about the resource is lost, as is the ability to prevent caching of requests and to specify additional header fields, since only a stream object is returned.
- The example that will be studied next shows how to use the URL Connection class to retrieve a resource and to determine its MIME content type and the length of the resource.
- Since we still use the URL class, some of the code is similar to previous examples.

Retrieving a Resource with the URLConnection Class

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

public class FetchURLConnection {

   public static void main(String args[]) throws Exception {
      int argc = args.length;
      // Check for valid number of parameters
      if (argc != 1) {
         System.out.println("Syntax :");
         System.out.println("java FetchURLConnection url");
         return;
      }
      try {
         // Create an instance of java.net.URL
         URL myURL = new URL(args[0]);
         // Create a URLConnection object, for this URL
         // NOTE : no connection has yet been established
         URLConnection connection = myURL.openConnection();
         // Now open a connection
         connection.connect();
         // Display the MIME content-type of the resource (e.g. text/html)
         String MIME = connection.getContentType();
         System.out.println("Content-type: " + MIME);
         // Display, if available, the content length
         int contentLength = connection.getContentLength();
         if (contentLength != -1) {
            System.out.println("Content-length: " + contentLength);
         }
         // Pause for user
         System.out.println("Hit enter to continue");
         System.in.read();
         // Read the contents of the resource from the connection
         InputStream in = connection.getInputStream();
         // Buffer the stream, for better performance
         BufferedInputStream bufIn = new BufferedInputStream(in);
         // Repeat until end of file
         for (;;) {
            int data = bufIn.read();
            // Check for EOF
            if (data == -1) {
               break;
            } else {
               System.out.print((char) data);
            }
         }
      } // MalformedURLException indicates parsing error
      catch (MalformedURLException mue) {
         System.err.println("Unable to parse URL!");
         return;
      } // IOException indicates network or I/O error
      catch (IOException ioe) {
         System.err.println("I/O Error : " + ioe);
         return;
      }
   }
}
```

How FetchURLConnection Works

- Rather than calling ***URL.openStream()*** once we have a URL object, however, the ***openConnection()*** method is called instead, returning a ***URLConnection*** instance.
- The name of the ***openConnection()*** method is somewhat misleading.
  - Although it creates an instance of the ***URLConnection*** object, it does not establish an HTTP session with the Web server.
  - This can be advantageous, as it allows us to set any request header fields we need.
- Once the connection is established and the request sent, a response will be issued by the Web server.
- This response will include a variety of header fields.
- One of the most important fields is the "Content-Type," which tells an application whether the resource is text, an image, a data file, or some other resource.
- The application could read this data by calling the ***URLConnection.getHeaderField(String)*** method and passing a value of "Content-Type."
- However, a shortcut method (used in this example) exists that makes for more readable source code.

How FetchURLConnection Works

- Another important piece of information is the length of the resource.
  - Large files can take minutes or even hours to download, and the user benefits from knowing the length of the resource at the beginning of the transaction.
  - This information is provided in the "Content-Length" header field, and a shortcut method exists for this data that converts it to an int value.
- After displaying the length, the application pauses, to allow the user time to read it before displaying the requested resource.
- The next step is to get an ***InputStream*** to the contents of the resource.
  - Just like a URL object, ***URLConnection*** provides a method to create an ***InputStream*** for reading a resource. For this purpose, the ***URLConnection.getInputStream()*** method is used.
- Once an ***InputStream*** has been obtained, the resource is read in the same way as the previous example.

Modifying and Examining Header Fields with URLConnection

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

public class HTTPHeaders {
   public static void main(String args[]) throws Exception {
      int argc = args.length;
      // Check for valid number of parameters
      if (argc != 1) {
         System.out.println("Syntax :");
         System.out.println("java HTTPHeaders url");
         return;
      }
      try {
         URL myURL = new URL(args[0]);
         URLConnection connection = myURL.openConnection();
         // Set some basic request fields
         // Set user agent, to identify the application as Netscape compatible
         connection.setRequestProperty("User-Agent","Mozilla/4.0 (compatible; JavaApp)");
         // Set our referer field - set to any URL you'd like
         connection.setRequestProperty("Referer","http://www.davidreilly.com/");
         // Set use-caches field, to prevent caching
         connection.setUseCaches(false);
         // Now open a connection
         connection.connect();
         // Examine request properties, to verify their settings
         System.out.println("Request properties....");
         System.out.println();
         System.out.println("User-Agent: "+ connection.getRequestProperty("User-Agent"));
         System.out.println("Referer: "connection.getRequestProperty("Referer"));
         System.out.println();
```

Modifying and Examining Header Fields with URLConnection

```
        // Examine response properties, to see their settings
        System.out.println("Response properties....");
        System.out.println();
        int i = 1;
        // Search through each header field, until no more exist
        while (connection.getHeaderField(i) != null) {
          // Get the name of this header field
          String headerName= connection.getHeaderFieldKey(i);
          // Get the value of this header field
          String headerValue= connection.getHeaderField(i);
          // Output header field key, and header field value
          System.out.println(headerName + ": "+ headerValue);
          // Goto the next element in the set of header fields
          i++;
        }
        // Pause for user
        System.out.println("Hit enter to continue");
        System.in.read();
      } // MalformedURLException indicates parsing error
      catch (MalformedURLException mue) {
        System.err.println("Unable to parse URL!");
        return;
      } // IOException indicates network or I/O error
      catch (IOException ioe) {
        System.err.println("I/O Error : " + ioe);
        return;
      }
    }
  }
```

How HTTPHeaders Works

- Like previous examples, this example uses an instance of the ***URLConnection*** class to issue HTTP requests.
- The chief difference here is that, before any request is sent, custom HTTP request fields are added.
- These header fields provide additional information to server-side applications, which can then be used to customize the HTTP response.
- When a Web browser sends a request, it identifies itself by sending a "User-Agent" field in the request.
  - Well-behaved HTTP clients do the same, and it is often advantageous to pose as a Web browser by including the *Mozilla* keyword in the identification string and then appending a legitimate-sounding application name.
  - CGI scripts and servlets sometimes offer different output depending on whether it's an HTTP agent like a search engine or an actual browser.
- Other request fields can also be set, such as the referring URL and the cache flag, which determines whether or not a unique request will be sent each time to the server.
- Once the request settings are made, the ***URLConnection*** object can send the request. If a call to the ***connect()*** method is made before assigning request properties, then the server will not receive them and they will not take effect

How HTTPHeaders Works (Cont.)

- The next set of header fields displayed by the application is from the server response.
- It would be impossible, however, to know the name of every field that might be sent back by a server.
- Not all servers support the same fields, and some server-side applications may send back custom fields that a client has never before encountered.
- The ***URLConnection*** offers several methods that provide access to request fields, two of which support a numerical index value rather than a key name.
- This allows us to read the nth key, and to iterate through every element in the set of header fields. The program prints out both the name of the field and its contents.

HttpURLConnection Class

- One of the problems of reading resources using ***URL.openStream()*** or the class is that access to HTTP-specific functionality is not available.
- Any ***URLConnection*** protocol for which a registered protocol handler exists can be fetched in this manner, including the File Transfer Protocol (FTP).
- But there is no notion of a request method, or a response status code, in these other protocols. How does an application know whether a resource was found, or a 404 "Not Found" error message was sent?
- The solution is to read the response status code, as it is the only appropriate way to determine the success or failure of requests.
- Though support for HTTP-specific functionality did not exist in earlier versions of Java, as of JDK1.1 there is a solution in the form of the ***HttpURLConnection*** class.
- ***HttpURLConnection*** extends the ***URLConnection*** class, and provides additional methods and fields that encapsulate HTTP functionality.

Creating a HttpURLConnection

- There are no public constructors for the ***HttpURLConnection*** class, just as in the case of ***URLConnection***.
- You should call the ***URL.openConnection***() method, which will return a ***URLConnection*** instance.
- The ***URLConnection*** class is the superclass of ***HttpURLConnection***, and if the protocol field of the URL is set to HTTP, this method will actually return an ***HttpURLConnection*** instance.
- To gain access to HTTP-specific functionality, you should test to see whether or not the object is an instance of ***HttpURLConnection***; if so, the object must be cast as such.

```
URL url = new URL ( some_url );
URLConnection connection = url.openConnection();
if (connection instanceof java.net.HttpURLConnection)
{
   HttpURLConnection httpConnection = (HttpURLConnection) connection;
   // do something with httpConnection
}
```

Using an HttpURLConnection

- The ***HttpURLConnection*** class inherits all of the functionality (including fields and methods) of its parent class, ***URLConnection***.
- It also adds additional functionality, in the form of methods that allow greater access to HTTP features, and static fields that represent common HTTP states.
- The ***HttpURLConnection*** class defines many static fields, which represent HTTP status codes.
- While an application can refer to a status code by a numerical value, these fields may make for more readable code.
- The following fields are all public static final int fields.

int HTTP_OK— HTTP status code (200) indicating that the request was successful.

int HTTP_CREATED— HTTP status code (201) indicating that a resource was created.

int HTTP_ACCEPTED— HTTP status code (202) indicating that the request was accepted but has not yet been acted upon.

Using an HttpURLConnection (Cont.)

int HTTP_MULT_CHOICE— HTTP status code (300) indicating that the resource can be found at multiple locations, from which the client can choose. When a resource is located elsewhere, a "Location" entity field will be sent, along with a 3xx redirection status code, but in the case of multiple choices of location, this status code will be issued.

int HTTP_MOVED_PERM— HTTP status code (301) indicating that the location of a resource has moved permanently and the client should look for a "Location" field in the HTTP response. The new location of the resource should be used in future.

int HTTP_MOVED_TEMP— HTTP status code (302) indicating that a temporary change has been made to the location of the resource, indicated by a "Location" field in the HTTP response.

int HTTP_SEE_OTHER— HTTP status code (303) indicating that a GET request should be used to fetch the resource, at a location specified by the "Location" field. This is often issued in response to a POST request, which processes the information and redirects to a standard page.

int HTTP_NOT_MODIFIED— HTTP status code (304) used to inform the client that a resource has not been modified and that no entity body was sent. This is used in conjunction with the "If-Modified-Since" request field, which performs a conditional GET request.

55

Using an HttpURLConnection (Cont.)

int HTTP_BAD_REQUEST— HTTP status code (400) that is issued in response to an invalid HTTP request, which fails to follow the correct syntax.

int HTTP_UNAUTHORIZED— HTTP status code (401) indicating that access to the resource requires user authentication.

int HTTP_PAYMENT_REQUIRED— HTTP status code (402) used to indicate that payment is required for access to this resource. This status code is reserved for the future, and is not in common use.

int HTTP_FORBIDDEN— HTTP status code (403) indicating that access to a resource is strictly forbidden.

int HTTP_NOT_FOUND— HTTP status code (404) used to notify a client that the resource could not be found or has been permanently removed.

int HTTP_SERVER_ERROR— HTTP status code (500) indicating that a server error occurred and the request could not be processed.

int HTTP_INTERNAL_ERROR— HTTP status code (501) indicating that a server did not know how to perform the request.

int HTTP_BAD_GATEWAY— HTTP status code (502) indicating that an error occurred while acting as a gateway or proxy server.

int HTTP_UNAVAILABLE— HTTP status code (503) indicating that the server could not process the request due to a temporary condition such a server overload.

56

Using an HttpURLConnection (Cont.)

void disconnect()— if a connection to the Web server is still active, the connection is closed.

InputStream getErrorStream()— returns an *InputStream* instance that can be used to read error messages sent by the server. If a connection has not yet been established, or no errors have yet occurred, this method returns null.

static boolean getFollowRedirects()— indicates whether HTTP redirects will be automatically followed. Returns "true" if redirection will occur automatically, and "false" if not.

String getRequestMethod()— returns the request method (e.g., GET) being used.

int getResponseCode()— returns the response status code. Applications can hardwire the numerical value of codes, or use the HttpURL Connection fields that define state conditions.

String getResponseMessage()— returns the message from the response status line, such as "OK," or "Not Found."

static void setFollowRedirects(boolean flag) throws java.lang.SecurityException— determines whether the resource specified in a redirection response will be automatically followed. This must be invoked prior to the *connect()* method for the setting to take effect. If this violates the settings of the security manager, a *SecurityException* will be thrown.

void setRequestMethod(String method) throws java.net.ProtocolException— sets the request method for this connection. This must be invoked prior to the *connect()* method for the setting to take effect. If the method is not supported, a *ProtocolException* will be thrown. The method name must be capitalized, as the protocol names are case sensitive.

boolean usingProxy()— shows whether a proxy server is being used for this connection. Returns "true" if using a proxy server, "false" if not.

Eng. Asma Abdel Karim
Computer Engineering Department

57

57

Accessing HTTP-Specific Functionality Using HttpURLConnection

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

public class UsingHttpURLConnection {

   public static void main(String args[]) throws Exception {
      int argc = args.length;
      if (argc != 1) {
         System.out.println("Syntax :");
         System.out.println("java UsingHttpURLConnection url");
         return;
      }
      try {
         URL myURL = new URL(args[0]);
         URLConnection connection = myURL.openConnection();
         // Check to see if connection is a HttpURLConnection instance
         if (connection instanceof java.net.HttpURLConnection) {
            // Yes... cast to a HttpURLConnection instance
            HttpURLConnection hConnection = (HttpURLConnection) connection;
            // Disable automatic redirection, to see the status header
            hConnection.setFollowRedirects(false);
            // Connect to server
            hConnection.connect();
            // Check to see if a proxy server is being used
            if (hConnection.usingProxy()) {
               System.out.println("Proxy server used to access resource");
            } else {
               System.out.println("No proxy server used to access resource");
            }
            // Get the status code
            int code = hConnection.getResponseCode();
            // Get the status message
            String msg = hConnection.getResponseMessage();
```

Eng. Asma Abdel Karim
Computer Engineering Department

58

58

Accessing HTTP-Specific Functionality Using HttpURLConnection

```
            // If a 'normal' response
            if (code == HttpURLConnection.HTTP_OK) {
              // Notify user
              System.out.println("Normal response returned: " + code + " " + msg);
            } else {
              // Output status code and message
              System.out.println("Abnormal response returned: " + code + " " + msg);
            }
            // Pause for user
            System.out.println("Hit enter to continue");
            System.in.read();
          } else {
            System.err.println("Invalid transport protocol -not http !");
            return;
          }
        } // MalformedURLException indicates parsing error
        catch (MalformedURLException mue) {
          System.err.println("Unable to parse URL!");
          return;
        } // IOException indicates network or I/O error
        catch (IOException ioe) {
          System.err.println("I/O Error : " + ioe);
          return;
        }
      }
    }
```

How UsingHttpURLConnection Works

- The program starts by creating a ***URL*** object, and from this, a ***URLConnection*** object.
- If the protocol being used to request the resource is HTTP, then the ***URLConnection*** will also be an instance of the ***HttpURLConnection*** class.
- A guard statement checks to see if it is an ***HttpURLConnection*** object and performs a casting operation.
  - If not, an error message is displayed and the program terminates.
- If all proceeds according to plan, the application now has an ***HttpURLConnection***, and the extra HTTP-specific functionality it gives.
- Before a connection is established, it is possible to modify the properties of the request.
- For example, a different request method could be used, or the "follow redirection" flag could be modified. So that users can see the redirection status code, automatic redirection is disabled by the application, and then the connection is established.

How UsingHttpURLConnection Works (Cont.)

- Once a connection has been established, all sorts of useful information becomes available, such as the status code and message and whether or not a proxy server is being used.
- The application checks for the presence of a proxy server, and displays it to the user, by using the boolean ***HttpURLConnection.usingProxy()*** method.
- Next, the status code and message are retrieved.
  - This is the most useful information of all, as it tells a client whether or not a request was successful and, if it was not successful, gives an indication of why.
- The human-readable shortcut for the 200 status code ***(HttpURLConnection.HTTP_OK)*** is used to check whether the request was successful. If not, the status code and message are displayed to the user, and an application could take further steps, such as resending a request or following a redirection notice.

61

Common Gateway Interface (CGI)

- The ***Common Gateway Interface*** (CGI) is an interface that allows HTTP clients, such as Web browsers and other user agents, to pass information back to a server for processing.
- CGI took the Web from static pages written by a Web master to interactive sites generated on the fly, in response to interactions with a user.
- When you use a search engine, buy a book at an online store, or read a customized newspaper tailored to your interests, your browser is using CGI to communicate with a server-side application.
- Earlier, we briefly discussed the POST method, which is used by HTTP clients to send information.
- The GET method may also be used to transmit information, although there are limitations on the length of data that may be passed.

62

Sending Data with the GET Method

- The GET method is used to request documents, images, and other files, and may also be used to call up server-side applications.
- Normally, when called, data will be passed to the CGI application using a query string.
- A query string is a string that is appended to the end of a URL in order to pass additional information such as the results of an HTTP form.

http://www.someserver.org/cgibin/form_submit?name=First%20Last&answer=no

- Because we know the format of a URL, we see nothing unexpected in the example before the "?“ character in the middle.
- But what does the question mark signify, and what comes after it?
- That’s the query string.

Sending Data with the GET Method (Cont.)

http://www.someserver.org/cgibin/form_submit?name=First%20Last&answer=no

- Everything that follows the question mark is a CGI parameter.
- These parameters are separated by an ampersand ("&") character.
- You'll notice too that instead of the expected space between one’s first and last names, there is a percentage sign and a number.
- This has been encoded, as Web browsers and CGI scripts don't easily handle certain ASCII characters (such as spaces, punctuation, and line separators).
- URL encoding substitutes problematic characters when sent through a query string.
- At the other end in the CGI script, URL decoding restores the characters to their original state.
- Since query strings are passed as a URL, they may be viewed by looking at the URL location string within a browser.
- In addition, the length of query strings is limited; long forms should be passed using the POST method.

Sending Data with the POST Method

- Sending data with the POST method removes the length limitations of GET and allows more complex data to be sent.
- While a query string may still be sent, data such as entire files and serialized objects may also be included.
- Query strings should still be URL encoded, and are sentby writing to the output stream of a ***URLConnection*** object.

Sending a GET Request in Java

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

public class SendGET {

   public static void main(String args[]) throws IOException {
      // Check command line parameters
      if (args.length < 1) {
         System.out.println("Syntax- SendGET baseurl");
         System.in.read();
         return;
      }
      // Get the base URL of the cgi-script/servlet
      String baseURL = args[0];
      // Start with a ? for the query string
      String arguments = "?";
      // Create a buffered reader, for reading CGI
      // parameters from the user
      BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
      // Loop until no parameters left
      for (;;) {
         System.out.println("Enter field ( . terminates )");
         String field = reader.readLine();
         // If a . char entered, terminate loop
         if (field.equals(".")) {
            break;
         }
         System.out.println("Enter value");
         String value = reader.readLine();
         // Encode the URL value
         arguments += URLEncoder.encode(field)+ "=" + URLEncoder.encode(value) + "&";
      }
```

Sending a GET Request in Java

```
// Construct the full GET request
String finalURL = baseURL + arguments;
System.out.println("Sending GET request - " + finalURL);
// Send the GET request, and display output
try {
  // Construct the url object
  URL url = new URL(finalURL);
  // Open a connection
  InputStream input = url.openStream();
  // Buffer the stream, for better performance
  BufferedInputStream bufIn = new BufferedInputStream(input);
  // Repeat until end of file
  for (;;) {
    int data = bufIn.read();
    // Check for EOF
    if (data == -1) {
      break;
    } else {
      System.out.print((char) data);
    }
  }
  // Pause for user
  System.out.println();
  System.out.println("Hit enter to continue");
  System.in.read();
} catch (MalformedURLException mue) {
  System.err.println("Bad URL - " + finalURL);
} catch (IOException ioe) {
  System.err.println("I/O error " + ioe);
}
  }
}
```

Running SendGET

- To run the application, you must specify the URL of a CGI script or Java servlet, and then enter one or more CGI parameters when prompted.
- To finish entering parameters, simply enter a "." character as the field name.
- A good start would be the AltaVista search engine (or your preferred search engine if you want to try something different).
- At the time of writing, a query could be sent by invoking the http://www.altavista.com/cgi-bin/query cgi-script, with a parameter "q" that represents the search query.
- For example, to search for "java networking" you might type the following command line and responses to questions:

```
java SendGET http://www.altavista.com/cgi-bin/query
Enter field ( . terminates )
q
Enter value
"java networking"
Enter field ( . terminates )
```

Sending a POST Request in Java

- A POST request is a bit more complex than a simple GET request.
- With a GET request, parameters are appended to a URL, but a POST request requires you to write parameters to the output stream of an HTTP connection.
- This means you can't use the *URL.openStream()* method, and must instead use a *URLConnection* object.

69

Sending a POST Request in Java

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

public class SendPOST {
   public static void main(String args[]) throws IOException {
      if (args.length < 1) {
         System.out.println("Syntax- SendPOST baseurl");
         System.in.read();
         return;
      }
      // Get the base URL of the cgi-script/servlet
      String baseURL = args[0];
      // No query string question mark required, so use a blank string
      String arguments = "";
      // Create a buffered reader, for reading CGI parameters from the user
      BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
      boolean firstParameter = true;
      // Loop until no parameters left
      for (;;) {
         System.out.println("Enter field ( . terminates )");
         String field = reader.readLine();
         // If a . char entered, terminate loop
         if (field.equals(".")) {
            break;
         }
         System.out.println("Enter value");
         String value = reader.readLine();
         if (!firstParameter) {
            arguments += "&";
         } else {
            firstParameter = false;
         }
         // Encode the URL value
         arguments += URLEncoder.encode(field)+ "=" + URLEncoder.encode(value);
      }
```

70

Sending a POST Request in Java

```
String query = arguments;
System.out.println("Sending POST request - " + query);
// Send the POST request, and display output
try {
  // Construct the url object representing cgi script
  URL url = new URL(baseURL);
  // Get a URLConnection object, to write to POST method
  URLConnection connect = url.openConnection();
  // Specify connection settings
  connect.setDoInput(true);
  connect.setDoOutput(true);
  // Get an output stream for writing
  OutputStream output = connect.getOutputStream();
  // Create a print stream, for easy writing
  PrintStream pout = new PrintStream(output);
  pout.print(query);
  pout.close();
  // Open a connection
  InputStream input = connect.getInputStream();
  // Buffer the stream, for better performance
  BufferedInputStream bufIn = new BufferedInputStream(input);
```

71

Sending a POST Request in Java

```
    // Repeat until end of file
    for (;;) {
      int data = bufIn.read();
      // Check for EOF
      if (data == -1) {
        break;
      } else {
        System.out.print((char) data);
      }
    }
    // Pause for user
    System.out.println();
    System.out.println("Hit enter to continue");
    System.in.read();
  } catch (MalformedURLException mue) {
    System.err.println("Bad URL - " + baseURL);
  } catch (IOException ioe) {
    System.err.println("I/O error " + ioe);
  }
 }
}
```

72

How SendPOST Works

- Writing to the POST method of a CGI script or servlet is accomplished by using a ***URLConnection*** object.
- Before the connection can be established, however, some initialization work needs to be done.
- By default, a ***URLConnection*** allows read access but no write access.
- You can override these defaults, by calling the ***setDoInput(boolean)*** and ***setDoOutput(boolean)*** methods, to allow data to be written as part of the POST HTTP request.

How SendPOST Works (Cont.)

- Now, the POST data must be sent to the CGI application by obtaining an ***OutputStream*** instance connected to the remote service.
- You can then connect any output stream or writer to it, so that the CGI parameters may be sent.

```
// Get an output stream for writing
OutputStream output = connect.getOutputStream();
// Create a print stream, for easy writing
PrintStream pout = new PrintStream (output);
pout.print ( query );
pout.close();
```

- Once sending the data for the request is complete, the results may be displayed by obtaining an ***InputStream t***o the ***URLConnection.***
- The request is sent, and the results returned just as if a GET request had been made.

Running SendPOST

- To run the application, you must specify the URL of a CGI script or Java servlet that supports the POST method.
- Not every script will, so you may need to search on your local intranet, or the Internet, for a suitable script.
- Once one is selected and passed as a command-line parameter, the application will prompt you for one or more CGI parameters. To finish entering parameters, simply enter a "." character as the field name.
- A good example is the Altavista search engine, which allows you to execute queries.
- You can perform searches on your own, by passing a query string to the search engine.
- For example, to search for "Java networking" as a term, you could pass the following data to Altavista:

```
java SendPOST http://www.altavista.com/sites/search/web
Enter field ( . terminates )
q
Enter value
java networking
Enter field ( . terminates )
```

References

Chapter 9 of *Java™ Network Programming and Distributed Computing*, David Reilly and Michael Reilly.