Process Communication

Introduction

The application program interface to UDP provides a *message passing* abstraction – the simplest form of interprocess communication.

This enables a sending process to transmit a single message to a receiving process.

The independent packets containing these messages are called *datagrams*.

In the Java and UNIX APIs, the sender specifies the destination using a socket – an indirect reference to a particular port used by the destination process at a destination computer.

The application program interface to TCP provides the abstraction of a two-way *stream* between pairs of processes. The information communicated consists of a stream of data items with no message boundaries.

Streams provide a building block for producer-consumer communication.

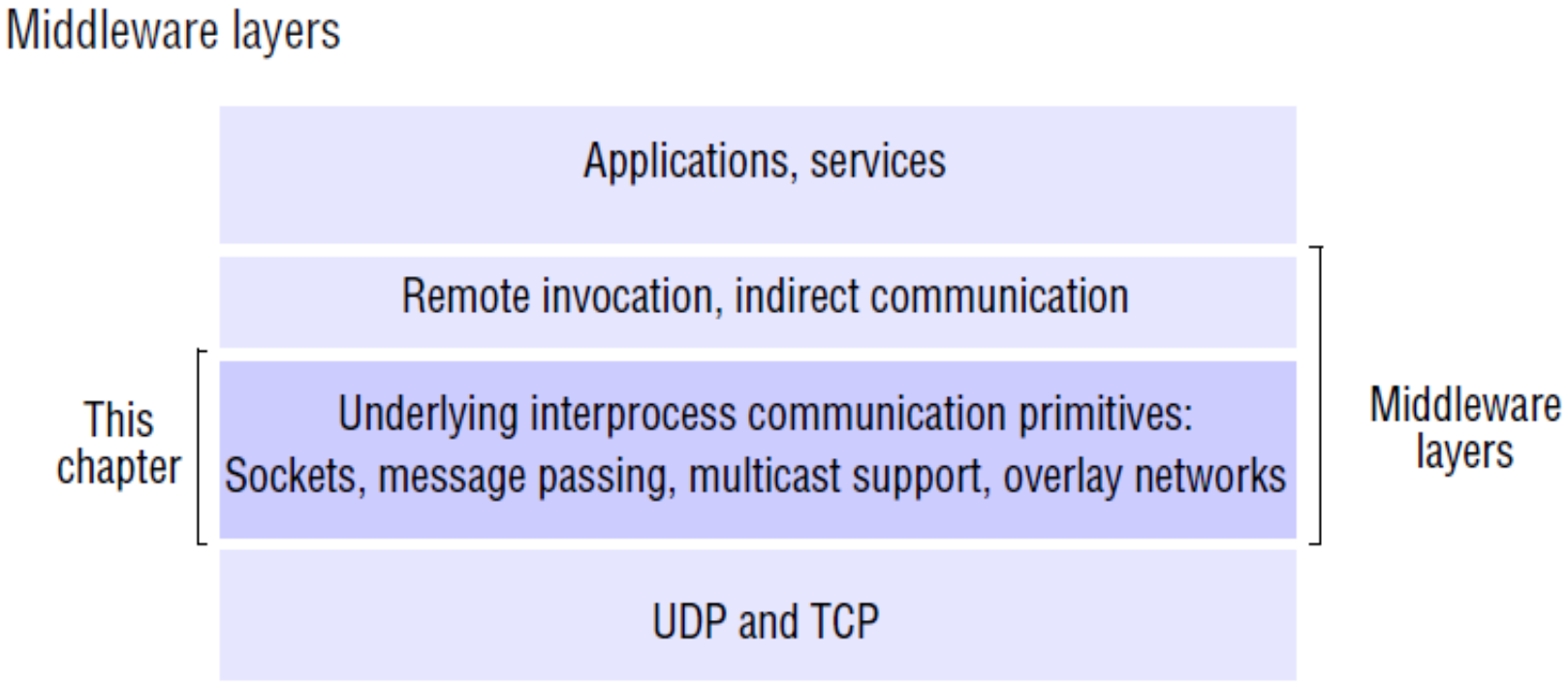
A producer and a consumer form a pair of processes in which the role of the first is to produce data items and the role of the second is to consume them.

Process Communication

The data items sent by the producer to the consumer are queued on arrival at the receiving host until the consumer is ready to receive them.

The consumer must wait when no data items are available.

The producer must wait if the storage used to hold the queued data items is exhausted.



Process Communication

The characteristics of interprocess communication

Message passing between a pair of processes can be supported by two message communication operations, *send* and *receive*, defined in terms of destinations and messages.

To communicate, one process sends a message (a sequence of bytes) to a destination and another process at the destination receives the message.

This activity involves the communication of data from the sending process to the receiving process and may involve the synchronization of the two processes.

Synchronous and asynchronous communication •

A queue is associated with each message destination.

Sending processes cause messages to be added to remote queues and receiving processes remove messages from local queues

In the *synchronous* form of communication, the sending and receiving processes synchronize at every message.

In this case, both *send* and *receive* are *blocking* operations.

Process Communication

Whenever a *send* is issued the sending process (or thread) is blocked until the corresponding *receive* is issued.

Whenever a *receive* is issued by a process (or thread), it blocks until a message arrives.

In the *asynchronous* form of communication, the use of the *send* operation is *nonblocking*

in that the sending process is allowed to proceed as soon as the message has

been copied to a local buffer, and the transmission of the message proceeds in parallel

with the sending process.

The *receive* operation can have blocking and non-blocking variants.

In the non-blocking variant, the receiving process proceeds with its program after issuing a *receive* operation, which provides a buffer to be filled in the background, but it must separately receive notification that its buffer has been filled, by polling or interrupt

Process Communication

Message destinations • In the Internet protocols, messages are sent to (*Internet address*, *local port*) pairs.

A local port is a message destination within a computer, specified as an integer.

A port has exactly one receiver but can have many senders.

Processes may use multiple ports to receive messages.

Any process that knows the number of a port can send a message to it. Servers generally publicize their port numbers for use by clients

Reliability • reliable communication in terms of validity and integrity.

As far as the validity property is concerned, a point-to-point message service can be described as reliable if messages are guaranteed to be delivered despite a ‘reasonable’ number of packets being dropped or lost.

In contrast, a point-to-point message service can be described as unreliable if messages are not guaranteed to be delivered in the face of even a single packet dropped or lost.

Process Communication

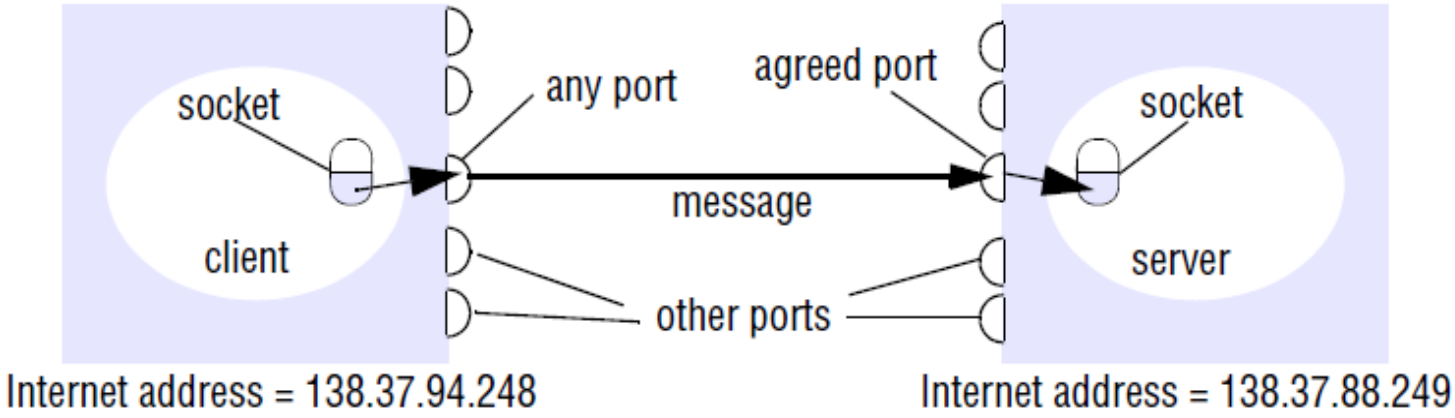
In telecommunications, a point-to-point connection refers to a communications connection between two communication endpoints or nodes.

An example is a telephone call, in which one telephone is connected with one other, and what is said by one caller can only be heard by the other.

Ordering • Some applications require that messages be delivered in *sender order* – that is, the order in which they were transmitted by the sender.

The delivery of messages out of sender order is regarded as a failure by such applications

Sockets and ports



Process Communication

Sockets

Both forms of communication (UDP and TCP) use the *socket* abstraction, which provides an endpoint for communication between processes.

Sockets originate from BSD UNIX but are also present in most other versions of UNIX, including Linux as well as Windows and the Macintosh OS.

Interprocess communication consists of transmitting a message between a socket in one process and a socket in another process, as illustrated in

Figure 4.2. For a process to receive messages, its socket must be bound to a local port and one of the Internet addresses of the computer on which it runs.

Messages sent to a particular Internet address and port number can be received only by a process whose socket is associated with that Internet address and port number.

Processes may use the same socket for sending and receiving messages.

Each computer has a large number (216) of possible port numbers for use by local processes for receiving messages

Process Communication

Java API for Internet addresses • As the IP packets underlying UDP and TCP are sent to Internet addresses, Java provides a class, *InetAddress*, that represents Internet addresses.

Users of this class refer to computers by Domain Name System (DNS) hostnames . For example, instances of *InetAddress* that contain Internet addresses can be created by calling a static method of *InetAddress*, giving a DNS hostname as the argument.

The method uses the DNS to get the corresponding Internet address.

For example, to get an object representing the Internet address of the host whose DNS name is *bruno.dcs.qmul.ac.uk*, use:

InetAddress aComputer = InetAddress.getByName("bruno.dcs.qmul.ac.uk");

This method can throw an *UnknownHostException*.

Process Communication

UDP datagram communication

A datagram sent by UDP is transmitted from a sending process to a receiving process without acknowledgement or retries.

If a failure occurs, the message may not arrive.

A datagram is transmitted between processes when one process *sends* it and another *receives* it. To send or receive messages a process must first create a socket bound to an Internet address of the local host and a local port.

A server will bind its socket to a *server port* – one that it makes known to clients so that they can send messages to it.

A client binds its socket to any free local port

The following are some issues relating to datagram communication:

Message size: The receiving process needs to specify an array of bytes of a particular size in which to receive a message. If the message is too big for the array, it is truncated on arrival.

The underlying IP protocol allows packet lengths of up to 216 bytes, which includes the headers as well as the message. However, most environments impose a size restriction of 8 kilobytes.

Process Communication

Blocking: Sockets normally provide non-blocking *sends* and blocking *receives* for datagram communication (a non-blocking *receive* is an option in some implementations).

The *send* operation returns when it has handed the message to the underlying UDP and IP protocols, which are responsible for transmitting it to its destination.

On arrival, the message is placed in a queue for the socket that is bound to the destination port.

The message can be collected from the queue by an outstanding or future invocation of *receive* on that socket.

The method *receive* blocks until a datagram is received, unless a **timeout** has been set on the socket. If the process that invokes the *receive* method has other work to do while waiting for the message, it should arrange to use a separate thread.

Timeouts: The *receive* that blocks forever is suitable for use by a server that is waiting to receive requests from its clients.

But in some programs, it is not appropriate that a process that has invoked a *receive* operation should wait indefinitely in situations where the sending process may have crashed or the expected message may have been lost.

To allow for such requirements, timeouts can be set on sockets.

Choosing an appropriate timeout interval is difficult, but it should be fairly large in comparison with the time required to transmit a message.

Receive from any: The *receive* method does not specify an origin for messages.

Instead, an invocation of *receive* gets a message addressed to its socket from any origin. The *receive* method returns the Internet address and local port of the sender, allowing the recipient to check where the message came from.

Failure model for UDP datagrams

UDP datagrams suffer from the following failures:

Omission failures: Messages may be dropped occasionally, either because of a checksum error or because no buffer space is available at the source or destination.

To simplify the discussion, we regard send-omission and receive-omission failures (see Figure 2.15) as omission failures in the communication channel.

Ordering: Messages can sometimes be delivered out of sender order.

Use of UDP • For some applications, it is acceptable to use a service that is liable to occasional omission failures. For example, the Domain Name System, which looks up DNS names in the Internet, is implemented over UDP.

Voice over IP (VOIP) also runs over UDP.

UDP datagrams are sometimes an attractive choice because they do not suffer from the overheads associated with guaranteed message delivery.

Process Communication

There are three main sources of overhead:

- the need to store state information at the source and destination;
- the transmission of extra messages;
- latency for the sender.

UDP client sends a message to the server and gets a reply

```
import java.net.*;
import java.io.*;
public class UDPClient{
   public static void main(String args[]){
      // args give message contents and server hostname
      DatagramSocket aSocket = null;
      try {
            aSocket = new DatagramSocket();
            byte [] m = args[0].getBytes();
            InetAddress aHost = InetAddress.getByName(args[1]);
            int serverPort = 6789;
            DatagramPacket request =
                  new DatagramPacket(m,  m.length(), aHost, serverPort);
            aSocket.send(request);
            byte[] buffer = new byte[1000];
            DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
            aSocket.receive(reply);
            System.out.println("Reply: " + new String(reply.getData()));
       } catch (SocketException e){System.out.println("Socket: " + e.getMessage());
       } catch (IOException e){System.out.println("IO: " + e.getMessage());
       } finally { if(aSocket != null) aSocket.close();}
   }
}
```

UDP server repeatedly receives a request and sends it back to the client

```
import java.net.*;
import java.io.*;
public class UDPServer{
   public static void main(String args[]){
      DatagramSocket aSocket = null;
      try{
            aSocket = new DatagramSocket(6789);
            byte[] buffer = new byte[1000];
            while(true){
               DatagramPacket request = new DatagramPacket(buffer, buffer.length);
               aSocket.receive(request);
               DatagramPacket reply = new DatagramPacket(request.getData(),
                     request.getLength(), request.getAddress(), request.getPort());
               aSocket.send(reply);
            }
      } catch (SocketException e){System.out.println("Socket: " + e.getMessage());
      } catch (IOException e) {System.out.println("IO: " + e.getMessage());
      } finally {if (aSocket != null) aSocket.close();}
   }
}
```

DatagramSocket: This class supports sockets for sending and receiving UDP datagrams.
It provides a constructor that takes a port number as its argument, for use by processes that need to use a particular port.
It also provides a no-argument constructor that allows the system to choose a free local port.
These constructors can throw a *SocketException* if the chosen port is already in use or if a reserved port (a number below 1024) is specified when running over UNIX.
The class *DatagramSocket* provides methods that include the following: *send* and *receive*:
These methods are for transmitting datagrams between a pair of sockets.
The argument of *send* is an instance of *DatagramPacket* containing a message and its destination.
The argument of *receive* is an empty *DatagramPacket* in which to put the message, its length and its origin.
The methods *send* and *receive* can throw *IOExceptions*.
setSoTimeout: This method allows a timeout to be set.
With a timeout set, the *receive* method will block for the time specified and then throw an *InterruptedIOException*.

connect: This method is used for connecting to a particular remote port and Internet address, in which case the socket is only able to send messages to and receive messages from that address.

The program for a client that creates a socket, sends a message to a server at port 6789 and then waits to receive a reply.
The arguments of the *main* method supply a message and the DNS hostname of the server. The message is converted to an array of bytes, and the DNS hostname is converted to an Internet address.
The program for the corresponding server, which creates a socket bound to its server port (6789) and then repeatedly waits to receive a request message from a client, to which it replies by sending back the same message.

TCP stream communication

The API to the TCP protocol, which originates from BSD 4.x UNIX, provides the abstraction of a stream of bytes to which data may be written and from which data may be read.

The following characteristics of the network are hidden by the stream abstraction:

Message sizes: The application can choose how much data it writes to a stream or reads from it.

It may deal in very small or very large sets of data.

The underlying implementation of a TCP stream decides how much data to collect before transmitting it as one or more IP packets.

On arrival, the data is handed to the application as requested. Applications can, if necessary, force data to be sent immediately.

Lost messages: The TCP protocol uses an acknowledgement scheme. As an example of a simple scheme (which is not used in TCP), the sending end keeps a record of each.

IP packet sent and the receiving end acknowledges all the arrivals. If the sender does not receive an acknowledgement within a timeout, it retransmits the message.
The more sophisticated sliding window scheme [Comer 2006] cuts down on the number of acknowledgement messages required.
Flow control: The TCP protocol attempts to match the speeds of the processes that read from and write to a stream. If the writer is too fast for the reader, then it is blocked until the reader has consumed sufficient data.
Message duplication and ordering: Message identifiers are associated with each IP packet, which enables the recipient to detect and reject duplicates, or to reorder messages that do not arrive in sender order.
Message destinations: A pair of communicating processes establish a connection before they can communicate over a stream.
Once a connection is established, the processes simply read from and write to the stream without needing to use Internet
addresses and ports.
Establishing a connection involves a *connect* request from client to server followed by an *accept* request from server to client before any communication can take place

The following are some outstanding issues related to stream communication:

Matching of data items: Two communicating processes need to agree as to the contents of the data transmitted over a stream. For example, if one process writes an *int* followed by a *double* to a stream, then the reader at the other end must read an *int* followed by a *double.* When a pair of processes do not cooperate correctly in their use of a stream, the reading process may experience errors when interpreting the data or may block due to insufficient data in the stream.

Blocking: The data written to a stream is kept in a queue at the destination socket. When a process attempts to read data from an input channel, it will get data from the queue or it will block until data becomes available. The process that writes data to a stream may be blocked by the TCP flow-control mechanism if the socket at the other end is queuing as much data as the protocol allows.

Threads: When a server accepts a connection, it generally creates a new thread in which to communicate with the new client. The advantage of using a separate thread for each client is that the server can block when waiting for input without delaying other clients.

Failure model

When a connection is broken, a process using it will be notified if it attempts to read or write. This has the following effects:

- The processes using the connection cannot distinguish between network failure and failure of the process at the other end of the connection.
- The communicating processes cannot tell whether the messages they have sent recently have been received or not.

Use of TCP • Many frequently used services run over TCP connections, with reserved port numbers. These include the following:

HTTP: The Hypertext Transfer Protocol is used for communication between web browsers and web servers;

FTP: The File Transfer Protocol allows directories on a remote computer to be browsed and files to be transferred from one computer to another over a connection.

Telnet: Telnet provides access by means of a terminal session to a remote computer.

SMTP: The Simple Mail Transfer Protocol is used to send mail between computers.

External Data Representation and Marshalling

The information stored in running programs is represented as data structures – for example, by sets of interconnected objects – whereas the information in messages consists of sequences of bytes.

Irrespective of the form of communication used, the data structures must be flattened (converted to a sequence of bytes) before transmission and rebuilt on arrival.

The individual primitive data items transmitted in messages can be data values of many different types, and not all computers store primitive values such as integers in the same order.

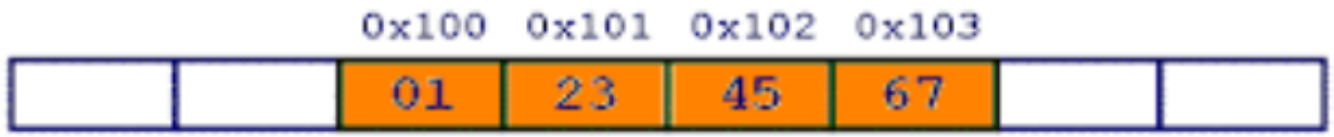
The representation of floating-point numbers also differs between architectures. There are two variants for the ordering of integers: the so-called *big-endian* order, in which the most significant byte comes first; and *little-endian* order, in which it comes last.

Another issue is the set of codes used to represent characters: for example, the majority of applications on systems such as UNIX use ASCII character coding, taking one byte per character, whereas the Unicode standard allows for the representation of texts in many different languages and takes two bytes per character.

What are these?

Little and big endian are two ways of storing multibyte data-types (int, float, etc). In little endian machines, last byte of binary representation of the multibyte data-type is stored first. On the other hand, in big endian machines, first byte of binary representation of the multibyte data-type is stored first.

Suppose integer is stored as 4 bytes (For those who are using DOS-based compilers such as C++ 3.0, integer is 2 bytes) then a variable x with value 0x01234567 will be stored as following.



Big Endian

Little Endian

One of the following methods can be used to enable any two computers to exchange binary data values:

• The values are converted to an agreed external format before transmission and converted to the local form on receipt; if the two computers are known to be the same type, the conversion to external format can be omitted.

• The values are transmitted in the sender's format, together with an indication of the format used, and the recipient converts the values if necessary.

An agreed standard for the representation of data structures and primitive values is called an ***external data representation.***

Marshalling is the process of taking a collection of data items and assembling them into a form suitable for transmission in a message.

Unmarshalling is the process of disassembling them on arrival to produce an equivalent collection of data items at the destination.

Thus marshalling consists of the translation of structured data items and primitive values into an external data representation.

Similarly, unmarshalling consists of the generation of primitive values from their external data representation and the rebuilding of the data structures.

• CORBA's common data representation, which is concerned with an external representation for the structured and primitive types that can be passed as the arguments and results of remote method invocations in CORBA.
It can be used by a variety of programming languages .
• Java's object serialization, which is concerned with the flattening and external data representation of any single object or tree of objects that may need to be transmitted in a message or stored on a disk. It is for use only by Java.
• XML (Extensible Markup Language), which defines a textual fomat for representing structured data.
It was originally intended for documents containing textual self-describing structured data – for example documents accessible on the Web – but it is now also used to represent the data sent in messages exchanged by clients and servers in web services .

CORBA's Common Data Representation (CDR)

CORBA CDR is the external data representation defined with CORBA 2.0 [OMG 2004a].

CDR can represent all of the data types that can be used as arguments and return values in remote invocations in CORBA.

These consist of 15 primitive types, which include *short* (16-bit), *long* (32-bit), *unsigned short*, *unsigned long*, *float* (32-bit), *double* (64-bit), *char*, *boolean* (TRUE, FALSE), *octet* (8-bit), and *any* (which can represent any basic or constructed type); together with a range of composite types, which are described in Figure 4.7

CORBA CDR for constructed types

Type	*Representation*
sequence	length (unsigned long) followed by elements in order
string	length (unsigned long) followed by characters in order (can also have wide characters)
array	array elements in order (no length specified because it is fixed)
struct	in the order of declaration of the components
enumerated	unsigned long (the values are specified by the order declared)
union	type tag followed by the selected member

Primitive types: CDR defines a representation for both big-endian and little-endian orderings.
Values are transmitted in the sender's ordering, which is specified in each message.
The recipient translates if it requires a different ordering.
Constructed types: The primitive values that comprise each constructed type are added to a sequence of bytes in a particular order, as shown in Figure 4.7.
Figure 4.8 shows a message in CORBA CDR that contains the three fields of a *struct* whose respective types are *string*, *string* and *unsigned long*.
The figure shows the sequence of bytes with four bytes in each row.
The representation of each string consists of an *unsigned long* representing its length followed by the characters in the string.
For simplicity, we assume that each character occupies just one byte

CORBA CDR message

index in sequence of bytes	← 4 bytes →	*notes on representation*
0–3	5	*length of string*
4–7	"Smit"	*'Smith'*
8–11	"h___"	
12–15	6	*length of string*
16–19	"Lond"	*'London'*
20–23	"on__"	
24–27	1984	*unsigned long*

The flattened form represents a *Person* struct with value: {'Smith', 'London', 1984}

Marshalling in CORBA • Marshalling operations can be generated automatically from the specification of the types of data items to be transmitted in a message. The types of the data structures and the types of the basic data items are described in CORBA IDL (see Section 8.3.1), which provides a notation for describing the types of the arguments and results of RMI methods. For example, we might use CORBA IDL to describe the data structure in the message in Figure 4.8 as follows:

```
struct Person{
        string name;
        string place;
        unsigned long year;
};
```

The CORBA interface compiler (see Chapter 5) generates appropriate marshalling and unmarshalling operations for the arguments and results of remote methods from the definitions of the types of their parameters and results.

Java object serialization

In Java RMI, both objects and primitive data values may be passed as arguments and results of method invocations.

An object is an instance of a Java class.

For example, the Java class equivalent to the *Person struct* defined in CORBA IDL might be:

```
public class Person implements Serializable {
private String name;
private String place;
private int year;
public Person(String aName, String aPlace, int aYear) {
name = aName;
place = aPlace;
year = aYear;
}
// followed by methods for accessing the instance variables
}
```

The above class states that it implements the *Serializable* interface, which has no methods.
In Java, the term *serialization* refers to the activity of flattening an object or a connected set of objects into a serial form that is suitable for storing on disk or transmitting in a message, for example, as an argument or the result of an RMI.
Deserialization consists of restoring the state of an object or a set of objects from their serialized form.
It is assumed that the process that does the deserialization has no prior knowledge of the types of the objects in the serialized form.
Therefore some information about the class of each object is included in the serialized form.
This information enables the recipient to load the appropriate class when an object is deserialized.

The information about a class consists of the name of the class and a version number. The version number is intended to change when major changes are made to the class. It can be set by the programmer or calculated automatically as a hash of the name of the class and its instance variables, methods and interfaces.

The process that deserializes an object can check that it has the correct version of the class.

To serialize an object, its class information is written out, followed by the types and names of its instance variables.

Serialized values				*Explanation*
Person	8-byte version number		h0	*class name, version number*
3	int year	java.lang.String name	java.lang.String place	*number, type and name of instance variables*
1984	5 Smith	6 London	h1	*values of instance variables*

The true serialized form contains additional type markers; h0 and h1 are handles.

The contents of the instance variables that are primitive types, such as integers, chars, booleans, bytes and longs, are written in a portable binary format using methods of the *ObjectOutputStream* class.
Strings and characters are written by its *writeUTF* method using the Universal Transfer Format (UTF-8), which enables ASCII characters to be represented unchanged (in one byte), whereas Unicode characters are represented by multiple bytes. Strings are preceded by the number of bytes they occupy in the stream.

As an example, consider the serialization of the following object:
Person p = new Person("Smith", "London", 1984);

The serialized form is illustrated in Figure 4.9, which omits the values of the handles and of the type markers that indicate the objects, classes, strings and other objects in the full serialized form.
The first instance variable (1984) is an integer that has a fixed length; the second and third instance variables are strings and are preceded by their lengths

Serialization and deserialization of the arguments and results of remote invocations are generally carried out automatically by the middleware, without any participation by the application programmer.

Extensible Markup Language (XML)

XML is a markup language that was defined by the World Wide Web Consortium (W3C) for general use on the Web.

In general, the term *markup language* refers to a textual encoding that represents both a text and details as to its structure or its appearance.

Both XML and HTML were derived from SGML (Standardized Generalized Markup Language) [ISO 8879], a very complex markup language.

HTML was designed for defining the appearance of web pages.

XML was designed for writing structured documents for the Web.

XML is a software- and hardware-independent tool for storing and transporting data.

What is XML?

•XML stands for eXtensible Markup Language
•XML is a markup language much like HTML
•XML was designed to store and transport data
•XML was designed to be self-descriptive
•XML is a W3C Recommendation

Example

```
<note>
 <to>Tove</to>
 <from>Jani</from>
 <heading>Reminder</heading>
 <body>Don't forget me this weekend!</body>
</note>
```

The XML above is quite self-descriptive:

•It has sender information
•It has receiver information
•It has a heading
•It has a message body

But still, the XML above does not DO anything. XML is just information wrapped in tags.

The Difference Between XML and HTML

XML and HTML were designed with different goals:

- XML was designed to carry data - with focus on what data is
- HTML was designed to display data - with focus on how data looks
- XML tags are not predefined like HTML tags are

XML Does Not Use Predefined Tags

The XML language has no predefined tags.

The tags in the example above (like <to> and <from>) are not defined in any XML standard. These tags are "invented" by the author of the XML document.

HTML works with predefined tags like <p>, <h1>, <table>, etc.

With XML, the author must define both the tags and the document structure.

XML Simplifies Things

- XML simplifies data sharing
- XML simplifies data transport
- XML simplifies platform changes
- XML simplifies data availability

Many computer systems contain data in incompatible formats. Exchanging data between incompatible systems (or upgraded systems) is a time-consuming task for web developers. Large amounts of data must be converted, and incompatible data is often lost.

XML stores data in plain text format. This provides a software- and hardware-independent way of storing, transporting, and sharing data.

XML also makes it easier to expand or upgrade to new operating systems, new applications, or new browsers, without losing data.

With XML, data can be available to all kinds of "reading machines" like people, computers, voice machines, news feeds, etc.
XML is a W3C Recommendation (**World Wide Web Consortium**)
XML became a W3C Recommendation as early as in February 1998.

XML is *extensible* in the sense that users can define their own tags, in contrast to HTML, which uses a fixed set of tags. However, if an XML document is intended to be used by more than one application, then the names of the tags must be agreed between them
Most XML applications will work as expected even if new data is added (or removed).

```
<note>
  <date>2015-09-01</date>
  <hour>08:30</hour>
  <to>Tove</to>
  <from>Jani</from>
  <body>Don't forget me this weekend!</body>
</note>
```

Sometimes ID references are assigned to elements. These IDs can be used to identify XML elements

XML definition of the *Person* structure

```
<person id="123456789">
        <name>Smith</name>
        <place>London</place>
        <year>1984</year>
        <!-- a comment -->
</person >
```

What is an XML Element?

An XML element is everything from (including) the element's start tag to (including) the element's end tag.

```
<price>29.99</price>
```

XML Attributes

Attributes are designed to contain data related to a specific element.

XML Attributes Must be Quoted

Attribute values must always be quoted. Either single or double quotes can be used.

For a person's gender, the <person> element can be written like this:

```
<person gender="female">
```

XML Namespaces

Name Conflicts

In XML, element names are defined by the developer. This often results in a conflict when trying to mix XML documents from different XML applications.

This XML carries HTML table information:

```
<table>
 <tr>
  <td>Apples</td>
  <td>Bananas</td>
 </tr>
</table>
```

his XML carries information about a table (a piece of furniture):

```
<table>
 <name>African Coffee Table</name>
 <width>80</width>
 <length>120</length>
</table>
```

Solving the Name Conflict Using a Prefix

Name conflicts in XML can easily be avoided using a name prefix. This XML carries information about an HTML table, and a piece of furniture:

```
<h:table>
 <h:tr>
  <h:td>Apples</h:td>
  <h:td>Bananas</h:td>
 </h:tr>
</h:table>
<f:table>
 <f:name>African Coffee Table</f:name>
 <f:width>80</f:width>
 <f:length>120</f:length>
</f:table>
```

In the example above, there will be no conflict because the two <table> elements have different names.

XML Schema

An XML Schema describes the structure of an XML document, just like a DTD.

An XML document with correct syntax is called "Well Formed".

An XML document validated against an XML Schema is both "Well Formed" and "Valid".

XML Schema

XML Schema is an XML-based alternative to DTD:

```
<xs:element name="note">
<xs:complexType>
 <xs:sequence>
  <xs:element name="to" type="xs:string"/>
  <xs:element name="from" type="xs:string"/>
  <xs:element name="heading" type="xs:string"/>
  <xs:element name="body" type="xs:string"/>
 </xs:sequence>
</xs:complexType>
</xs:element>
```

xs:element name="note"> defines the element called "note"
•<xs:complexType> the "note" element is a complex type
<xs:sequence> The Schema above is interpreted like this:
•<the complex type is a sequence of elements
•<xs:element name="to" type="xs:string"> the element "to" is of type string (text)
•<xs:element name="from" type="xs:string"> the element "from" is of type string
•<xs:element name="heading" type="xs:string"> the element "heading" is of type string
•<xs:element name="body" type="xs:string"> the element "body" is of type string

XML Schema: It is primarily used to define the elements, attributes and data types the document can contain.