Chapter 1

Intersystem Communications: Architectures for integrating systems; DCOM, CORBA, and RMI

In this chapter, you will learn three architectures of distributed object oriented technologies or middleware technologies or Architectures for integrating systems such as DCOM, CORBA, RMI and their comparisons.

Intersystem Communications:

The ability of two or more computer systems to share input, output, and storage devices, and to send messages to each other by means of shared input and output channels or by channels that directly connect central processors.

1.1 Middleware Components and Middleware

A ***middleware component*** is software that connects two other separate applications.

For example, there are a number of middleware products that link a database system to a Web server. This allows users to request data from the database using forms displayed on a Web browser, and it enables the Web server to return dynamic Web pages based on the user's requests and profile.

The term ***middleware*** is used to describe separate products that serve as ***the glue*** between two applications. It is, therefore, distinct from import and export features that may be built into one of the applications. Middleware is sometimes called ***plumbing*** because it connects two sides of an application and passes data between them.

To achieve this, Some Distributed object technologies such as Distributed Component Object Model (DCOM), Common Object Request Broker Architecture (CORBA) and Java Remote Method Invocation (Java RMI) have been selected according to the following criteria:

i) Portability
ii) Platform independence
iii) Wide use
iv) Database independence

1.2 Distributed Object Technologies:

A ***distributed system*** is defined as a system in which hardware or software components located at networked computers communicate and co-ordinate their actions only by passing messages. Computing devices may be connected to a wide range of networks as for instance the Internet, mobile phone networks, corporate networks, home networks or to combinations of these.

Modern program systems like Internet-based and enterprise applications offer multi-tiered, component-based architectures that incorporate middleware for distributing components across heterogeneous platforms. The platforms range from mobile devices like personal digital assistants (PDA), laptops and mobile phones; ubiquitous devices like televisions, refrigerators and cars; to different types of computers like mainframes and PCs.

The three most dominating distributed object technologies or middleware are CORBA, DCOM and Java/RMI. These are extensions of traditional object-oriented systems by allowing objects to be distributed across a heterogeneous network. The objects may reside in their own address space outside of an application or on a different computer than the application and still be referenced as being part of the application.

All three distributed object technologies are based on a client/server approach implemented as network calls operating at the level of bits and bytes transported on network protocols like TCP/IP. In order to avoid the hard and error prone implementation of network calls directly in the client and server {objects}, the distributed technology standards address the complex networking interactions by abstract and hide the networking issues and instead let the programmer concentrate on programming the business logic.

The basic idea behind network abstractions like RPC (Remote Procedure Call) is to replace the local (server) and remote (client) end by stubs. This makes it possible for both client and server to strictly use local calling conventions and thereby be unaware of calling a remote implementation or being called remotely. To accomplish this the client call is handled by a client stub (proxy) that marshals the parameters and sends them by invoking a wire protocol like IIOP (*Internet Inter ORB Protocol)*, ORPC (Object Remote Procedure call) or JRMP(Java RMI Protocol), to the remote end where another stub receives the parameters, unmarshals and calls the true server. The marshaling and unmarshaling actions are responsible for converting data values from their local representation to a network format and on to the remote representation. Format differences like byte ordering and number representations are bridged this way.

The object services offered by all three approaches are defined through interfaces. The interface is for all defined as a collection of named operations, each with a defined signature and optionally a return type. The interface serves as a contract between the server and client.

Distributed Object Technologies or Distributed Object Computing (DOC) integrates heterogeneous applications. DOC extends an object-oriented system by providing a means to distribute objects across a network, allowing each component to interoperate as a unified whole. Objects look "local" to applications, even though they are distributed to different computers throughout a network.

Distributed Object Technologies are useful in many sever situations. A distributed processing is also widely used with a parallel processing approach such as clustering, especially in multi-tier environments.

Remoting:

Many Distributed Object Technologies such as DCOM, CORBA model and Java RMI are used to invoke a remote method, also known as ***remoting.***

To invoke a remote method,

- **(i)** The client makes a call to the client-side proxy or stub.
- **(ii)** The client-side proxy packs the call parameters into a request message and invokes a wire protocol to ship the message to the server.
- **(iii)** At the server side, the wire protocol delivers the message to the server-side stub.
- **(iv)** The server-side stub then unpacks the message and calls the actual method on the object.

In both CORBA and Java RMI, *the server stub is called the skeleton and client stub is called the stub or proxy*. In DCOM, *the client stub is called the proxy and the server stub is called the stub*.

CORBA

CORBA depends on an Object Request Broker (ORB), a central bus over which CORBA objects interact transparently. CORBA uses Internet Inter-ORB Protocol (II-ORB) for remoting objects.
To request a service, A CORBA client acquires an object reference to a CORBA server object. The client then makes method calls on the object reference as if the CORBA server object resided in the

client's address space. The ORB finds a CORBA object's implementation, prepares it to receive requests, communicates the requests, and carries the replies back to the client. CORBA can be used on a range of operating system platforms, from hand-held devices to mainframes.

DCOM

DCOM is as an extension of the *Component Object Model (COM)*, a Microsoft framework that supports program component objects. DCOM supports remoting objects through a protocol named ***Object Remote Procedure Call (ORPC)***. ORPC is a layer that interacts with COM's run-time services. A DCOM server is a body of code capable of serving up particular objects at run-time. Each DCOM server object supports multiple interfaces, each of which represent a different behavior of the object. A DCOM client calls into the exposed methods of a DCOM server by acquiring a ***pointer-to-server-object interface***. The client object calls into the server object's exposed methods through the interface pointer, as if the server object resided in the client's address space.

Java RMI

RMI is the Java version generally known as a ***remote procedure call (RPC)***, with the added ability to pass objects along with the request. The objects can include information that change the service performed in the remote computer. This property of RMI is often called ***"moving behavior."***

Java RMI relies on the ***Java Remote Method Protocol and on Java Object Serialization***, which allows objects to be transmitted as a stream. Since Java Object Serialization is specific to Java, both the Java RMI server object and the client object have to be written in Java. Java RMI allows client/server applications to invoke methods across a distributed network of servers running the *Java Virtual Machine*.

Although RMI is considered by many to be weaker than CORBA and DCOM, it offers such unique features as *distributed automatic object management* and has the ability to *pass objects between machines*. The naming mechanism, ***RMIRegistry***, runs on the server machine and holds information about available server objects.

A Java RMI client acquires a reference to a Java RMI server object by looking up a server object reference and invoking methods on the server object, as if the Java RMI server object resided on the client. These server objects are named using ***Universal Resource Locators (URL)***. A client acquires a reference by specifying the server object's URL, just as you would specify the URL to an HTML page.

While CORBA, DCOM, and Java RMI all provide similar mechanisms for transparently accessing remote distributed objects,

- DCOM is a proprietary solution that works best in Microsoft environments. For an organization that has adopted a Microsoft-centered strategy, DCOM is an excellent choice. However, if any other operating systems are required in the application architecture, DCOM is probably not the correct solution.
- Because of its easy-to-use native-JAVA model, RMI is the simplest and fastest way to implement distributed object architecture. It's a good choice for small applications implemented completely in Java. Since RMI's native-transport protocol, JRMP, can only communicate with other Java RMI objects, it's not a good choice for heterogeneous applications.
- CORBA and DCOM are similar in capability, but DCOM doesn't yet support operating system interoperability, which may discount it as a single solution. At the moment, CORBA is the logical choice for building enterprise wide, open-architecture, distributed object applications.

1.3 DCOM Architecture

DCOM – Distributed Component Object Model, is a standard developed by Microsoft and it is a distributed extension of the COM standard [COM] COM to fit into network environments. The COM standard is based on the development of compound document technology to integrate document parts (for instance spread-sheets, pictures, presentations, word processors etc.) created by different Windows applications. COM is the world most widely used component software model and dominates the desktop market. Fig. 1.1 shows the COM/DCOM architecture.

A client that wants to access a server object calls a COM ***runtime through COM interfaces***, which checks the client's permissions by invoking a ***security provider***. If the client has the appropriate permissions, the COM runtime invokes the ***Distributed Computing Environment Remote Procedure Call (DCE RPC)*** that uses the underlying protocol stack as one of communication methods. Before invoking a DCE RPC, a client side COM runtime performs a **marshalling through a proxy** that converts parameters into the form that could be transferred over the designated protocol.

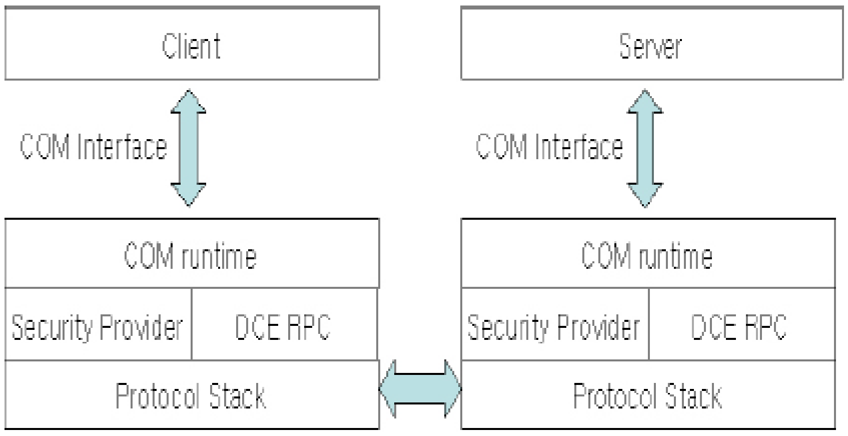


Figure 1.1 COM/DCOM architecture

The protocol stack on the server side receives the request, delivers it to the COM runtime, which performs an ***unmarshalling through a stub*** that converts network packets into parameters. The COM runtime activates the ***server object that calls the object method and processes the request***. A server object has the **3 types such as an *inproc*, a *local*, and a *service* type**. An inproc type takes the form of a *Dynamic Linking Library (DLL)*. An inproc type is the fastest type because it is loaded directly into a client process when activated. A local type takes an *executable form*. When activated, it is executed as a separate process. Lastly, a service type is also similar to the local type in that both of them take executable forms, but different from the local type in that the service type is always resident in a memory, and receives a request from a client. In order to export the methods of a server object, a server object should describe itself by using an ***Interface Description Language (IDL)*.**

For methods that are commonly used in the COM, COM previously creates a set of common interfaces such as *IUnknown*, *IDispatch*, and *IClassFactory*. Although Windows operating system incorporates it, both COM and DCOM are supported only by limited operating systems, and they can't be used in most of web environments due to protection policies.

Benefits of DCOM

- Large User Base and Component Market
- Binary Software Integration
    - Large-scale software reuse (no source code)
    - Cross-language software reuse
- On-line software update

- Allows updating a component in an application without recompilation, re-linking or even restarting
- Multiple interfaces per object
- Wide selection of programming tools available, most of which provide automation of standard code

1.4 CORBA Architecture

CORBA – Common Object Request Broker Architecture, is an open distributed object computing infrastructure standardized by the Object Management Group (OMG) and is a specification based on technologies proposed (and partly provided) by the software industry [CORBA]. CORBA is the most used middleware standard in the non-Windows market. The OMG was founded in 1989 to promote the adoption of object-oriented technology and reusable software components. Fig. 1.2 shows the CORBA architecture.

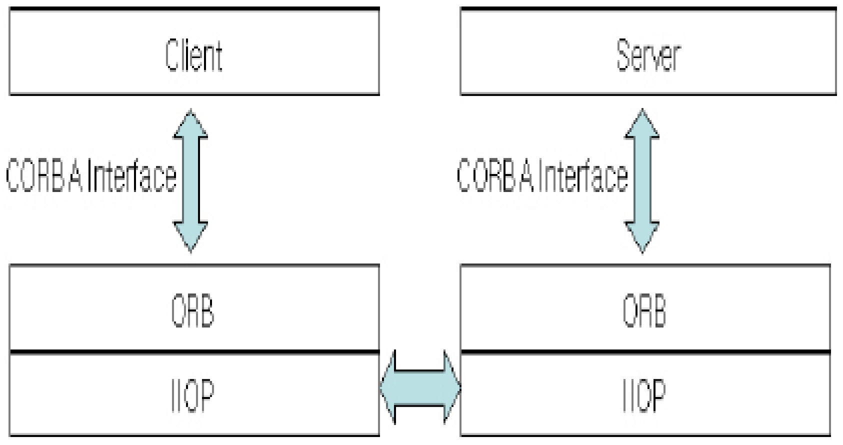


Figure 1.2. CORBA architecture

A client that wants to access a server object should first bind it through a CORBA interface. The client makes a bind request to a local ***Object Request Broker (ORB) through CORBA interfaces***. If the object exists at the local machine, the ORB activates locally the object that processes the request. Otherwise, the ORB makes a request to the other ORBs connected by the network using the ***Internet Inter ORB Protocol (IIOP) over TCP/IP***. After binding the appropriate object, the client side ORB sends the requested method call to the server side ORB. Before sending and receiving parameters, a *stub* performs marshalling and a *skeleton* performs unmarshalling at the CORBA. CORBA has the 2 server types such as a ***Basic Object Adapter (BOA)* and a *Portable Object Adapter (POA).*** A BOA type was the early model, and its ambiguity caused many different implementations at an early stage. A POA type as developed to solve this incompatibility problem, and came to be a standard. A CORBA also has the ***IDL that is similar to the COM IDL.***

Although CORBA is supported by various operating systems, its users should also choose their own CORBA vendors because various vendors implementing CORBA specification exist, and they can't be used in most of web environments due to protection policies.

Benefits of CORBA

- Programming-language independent interface
- Legacy integration
- Rich distributed object infrastructure
- Location transparency
- Network transparency
- Direct object communication
- Dynamic Invocation Interface

1.5 Java/RMI – Java/Remote Method Invocation

Java/RMI – Java/Remote Method Invocation, is a standard developed by JavaSoft. Java has grown from a programming language to three basic and completely compatible platforms; J2SE (Java 2 Standard Edition), J2EE (Java 2 Enterprise Edition) and J2ME (Java 2 Micro Edition). J2SE is the foundational programming language and tool-set for coding and component development. J2EE supplements J2SE and is a set of technologies and components for enterprise and Internet development. J2ME is used for creating software for embedded, mobile, consumer and other small devices like Personal Digital Assistants (PDA) and mobile phones.

RMI supports remote objects by running on a protocol called the *Java Remote Method Protocol (JRMP).* Object serialisation is heavily used to marshal and unmarshal objects as streams. Both client and server have to be written in Java to be able to use the object serialisation. The Java server object defines interfaces that can be used to access the object outside the current Java virtual machine (JVM) from another JVM on for instance a different machine. A RMI registry on the server machine holds information of the available server objects and provides a naming service for RMI. A client acquires a server object reference through the RMI registry on the server and invokes methods on the server object as if the object resided in the client address space. The server objects are named using URLs and the client acquires the server object reference by specifying the URL.

When a Java/RMI client requests a service from the Java/RMI server, it does the following [Java/RMI]:

(i) initiates a connection with the remote JVM containing the remote object,

(ii) marshals the parameters to the remote JVM,

(iii) waits for the result of the method invocation,

(iv) unmarshals the return value or exception returned, and

(v) returns the value to the caller.

By using serialization of the objects, both data and code can be passed between a server and a client – this allows different instances of an object to run on both client and server machines. To insure that code is downloaded or uploaded safely, RMI provides extra security.

To declare remote access to server objects in Java, every server object must implement the java.rmi.Remote interface. java.rmi.server.RemoteObject and its subclasses, java.rmi.server.RemoteServer, java.rmi.server.UnicastRemoteObject and java.rmi.activation.Activatable provide RMI server functions. The class java.rmi.server.RemoteObject provides implementations for the java.lang.Object methods, hashCode, equals, and toString that are sensible for remote objects. The classes UnicastRemoteObject and Activatable provide the methods needed to create remote objects and make them available to remote clients. The subclasses identify the semantics of the remote reference, for example whether the server is a simple remote object or is an activatable remote object (one that executes when invoked) [Java/RMI]. The java.rmi.server.UnicastRemoteObject class defines a singleton (unicast) remote object whose references are valid only while the server process is alive. The class java.rmi.activation.Activatable is an abstract class that defines an activatable remote object that starts executing when its remote methods are invoked and can shut itself down when necessary [Java/RMI]. Java/RMI can be used on a diversity of platforms and operating systems as long as there is a JVM implementation on the platform.

Benefits of RMI:

- Support seamless remote invocation on objects in different virtual machines
- Support callbacks from servers to applets
- Integrate the distributed object model into the Java language in a natural way while retaining most of the Java language's object semantics
- Make differences between the distributed object model and local Java object model apparent
- Make writing reliable distributed applications as simple as possible
- Preserve the safety provided by the Java runtime environment
- Single language

- Free

1.6 Architecture comparison

The following table 1.1 shows the similarities and dissimilarities among three architectures

Table 1.1 Architecture comparison (CORBA, DCOM, Java/RMI)

	CORBA	**DCOM**	**Java/RMI**
Similarity1: Component based methodology	To maximize reusability and rapid application development, component based approach is applied. Component based approach heavily depends on interface definition. OMG IDL, COM IDL, and so called Java IDL (simply mapping for Java to the OMG IDL) are used.		
Similarity2: Distributed Network	Components can be distributed over network.		
Similarity3: Naming and locating service	All services provide some sort of registry or repository to help locate the corresponding service.		
Dissimilarity1: Vendor and players	Multiple vendor support	Microsoft with some third parties	Sun microsystem with other vendors
Dissimilarity2: Language	A variety of languages	Some languages such as VB/ VC++ / MS-Java	Java
Dissimilarity3: Platform	Platform independent with specification	Win32	Platform independent by JVM

Chapter 2

Web Services and Middleware; Network programming; Message and queuing services; Low level data communications

2.1 Web Services and Middleware

Web services:

Web Services is an application integration technology. Web Services allow applications to be integrated more rapidly, easily and less expensively. The Web did for program-to-user interactions; Web Services are developed for program-to- program interactions. Web Services allow companies to reduce the cost of doing e-business, to deploy solutions faster and to open up new opportunities. Web services model built on existing and emerging standards such as HTTP, Extensible Markup Language (XML), Simple Object Access Protocol (SOAP), Web Services Description Language (WSDL) and Universal Description, Discovery and Integration (UDDI).

- XML Web Services *expose useful functionality* to Web users through a standard Web protocol. In most cases, the protocol used is SOAP.
- XML Web services *provide a way to describe their interfaces in enough detail* to allow a user to build a client application to talk to them. This description is usually provided in an XML document called a Web Services Description Language (WSDL) document.
- XML Web services *are registered* so that potential users can find them easily. This is done with Universal Discovery Description and Integration (UDDI).

Web service is developed in order to distribute an object and serve it to various users in the web environments. It can be also used in the server situations while solving the web-scalability problem of the other distributed object technologies. Fig.2.1 demonstrates the web service architecture.

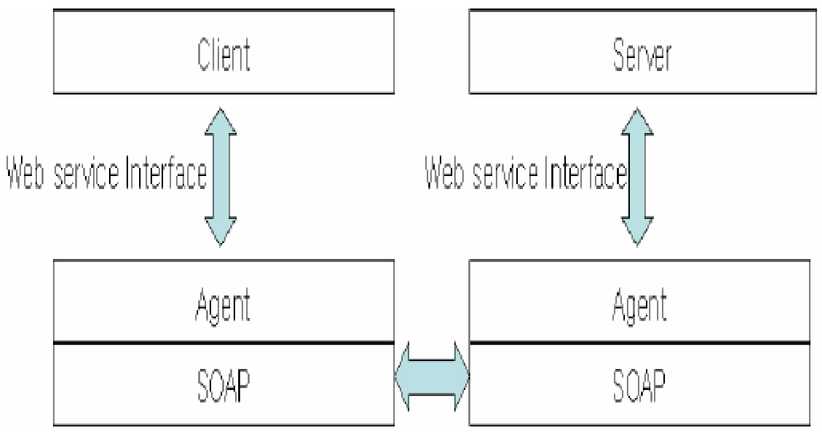


Figure.2.1. Web service architecture

1. A client that wants to be serviced should first find the supported services from the pre-existing *registry* before compiling a code.
2. After finding its services through searching, the client gains the *Web Service Description Language (WSDL)* that a server previously registers. From the WSDL, the client knows the service provider location and the parameters to the found method.
3. After the client binds the described service during the compile time, it calls the local agent whenever the client invokes a method call, and the local agent delivers it to the server side agent through *Simple Object Access Protocol (SOAP)* over HTTP, FTP, SMTP, IIOP, and TCP during the runtime.
4. The server side agent activates the appropriate object, and delivers the calls to the object.

All of the communication methods such as a WSDL, and SOAP exploit the eX*tensible Markup Language (XML)*. WSDL is an XML describing the web service. SOAP is an XML describing the called method, its parameters, and its return value, can be delivered over the HTTP.

2.2 Network Programming

2.2.1 Basic Network Concepts

Network:

- A network is a collection of computers and other devices that can send data to and receive data from each other.
- A network is often connected by wires.
- However, wireless networks transmit data through infrared light and microwaves.

Node:

- Each machine on a network is called a node.
- Most nodes are computers, but printers, routers, bridges, gateways etc.. can also be nodes.
- Nodes that are fully functional computers are also called hosts.

Packet:

- 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.

Protocol:

- 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.
- There are many different protocols defining different aspects of network communication.

IP:

- IP was designed to allow multiple routes between any two points and to route packets of data around damaged routers*.*

TCP:

- Since there are multiple routes between two points, and since the quickest path between two points may change over time as a function of network traffic and other factors), the 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.

UDP:

- 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.

Ports:

- Each computer with an IP address has several thousand logical ports.
- Each port is identified by a number between 1 and 65,535. Each port can be allocated to a particular service.
- Port numbers 1 through 255 are reserved by IP for well-known services. A well-known service is a service that is widely implemented which resides at a published, "well-known", port. If you connect to port 80 of a host, for instance, you may expect to find an HTTP server.

2.2.2 Network Programming Concepts:

Internet:

The Internet is the world's largest IP-based network. The Internet is all about connecting machines together. One of the most exciting aspects of Java is that it incorporates an easy-to-use, cross-platform model for network communications.

What is a Socket?

Sockets are a means of using IP to communicate between machines, so sockets are one major feature that allows Java to interoperate with legacy systems by simply talking to existing servers using their pre-defined protocol.

Other common protocols are layered on top of the Internet protocol: User Datagram Protocol (UDP) and Transmission Control Protocol (TCP). Applications can make use of these two protocols to communicate over the network.

Internet Addresses or IP Addresses

Every network node has an address, a series of bytes that uniquely identify it. Internet addresses are manipulated in Java by the use of the InetAddress class. InetAddress takes care of the Domain Name System (DNS) look-up and reverse look-up; IP addresses can be specified by either the host name or

the raw IP address. InetAddress provides methods to getByName(), getAllByName(), getLocalHost(), getAddress(), etc.
IP addresses are a 32-bit number, often represented as a "quad" of four 8-bit numbers separated by periods. They are organized into classes (A, B, C, D, and E). For example 126.255.255.255

Client/Server Computing

You can use the Java language to communicate with remote file systems using a client/server model. A server listens for connection requests from clients across the network or even from the same machine. Clients know how to connect to the server via an IP address and port number. Upon connection, the server reads the request sent by the client and responds appropriately. In this way, applications can be broken down into specific tasks that are accomplished in separate locations.
The data that is sent back and forth over a socket can be anything you like. Normally, the client sends a request for information or processing to the server, which performs a task or sends data back. The IP and port number of the server is generally well-known and advertised so the client knows where to find the service.

How UDPclients and UDPservers communicate over sockets

Creating UDP Servers:

To create a server with UDP, do the following:

1. Create a DatagramSocket attached to a port.

```
int port = 1234;
DatagramSocket socket = new DatagramSocket(port);
```

2. Allocate space to hold the incoming packet, and create an instance of DatagramPacket to hold the incoming data.

```
byte[] buffer = new byte[1024];
DatagramPacket packet = new DatagramPacket(buffer, buffer.length);
```

3. Block until a packet is received, then extract the information you need from the packet.

```
// Block on receive()
socket.receive(packet);
// Find out where packet came from
// so we can reply to the same host/port
InetAddress remoteHost = packet.getAddress();
int remotePort = packet.getPort();
// Extract the packet data
byte[] data = packet.getData();
```

The server can now process the data it has received from the client, and issue an appropriate reply in response to the client's request.

Creating UDP Clients

Writing code for a UDP client is similar to what we did for a server. Again, we need a DatagramSocket and a DatagramPacket. The only real difference is that we must specify the destination address with each packet, so the form of the DatagramPacket constructor used here specifies the destination host and port number. Then, of course, we initially send packets instead of receiving.

1. First allocate space to hold the data we are sending and create an instance of DatagramPacket to hold the data.

```
byte[] buffer = new byte[1024];
int port = 1234;
InetAddress host = InetAddress.getByName("magelang.com");
DatagramPacket packet = new DatagramPacket(buffer, buffer.length, host, port);
```

2. Create a DatagramSocket and send the packet using this socket.

```
DatagramSocket socket = new DatagramSocket();
socket.send(packet);
```

The DatagramSocket constructor that takes no arguments will allocate a **free local port to use**. You can find out what local port number has been allocated for your socket, along with other information about your socket if needed.

```
// Find out where we are sending from
InetAddress localHostname = socket.getLocalAddress();
int localPort = socket.getLocalPort();
```

The client then waits for a reply from the server. Many protocols require the server to reply to the host and port number that the client used, so the client can now invoke socket.receive() to wait for information from the server.

How TCPclients and TCPservers communicate over sockets

Creating TCP Servers:

To create a TCP server, do the following:

1. Create a ServerSocket attached to a port number.

```
ServerSocket server = new ServerSocket(port);
```

2. Wait for connections from clients requesting connections to that port.

```
// Block on accept()
Socket channel = server.accept();
```

You'll get a Socket object as a result of the connection.

3. Get input and output streams associated with the socket.

```
out = new PrintWriter (channel.getOutputStream());
reader = new InputStreamReader (channel.getInputStream());
in = new BufferedReader (reader);
```

Now you can read and write to the socket, thus, communicating with the client.

```
String data = in.readLine();
out.println("Hey! I heard you over this socket!");
```

When a server invokes the accept() method of the ServerSocket instance, the main server thread blocks until a client connects to the server; it is then prevented from accepting further client connections until the server has processed the client's request. This is known as an *iterative* server, since the main server method handles each client request in its entirety before moving on to the next request. Iterative servers are good when the requests take a known, short period of time. For example, requesting the current day and time from a time-of-day server.

Creating TCP Clients:

To create a TCP client, do the following:

1. Create a Socket object attached to a remote host, port.

```
Socket client = new Socket(host, port);
```

When the constructor returns, you have a connection.

2. Get input and output streams associated with the socket.

```
out = new PrintWriter (client.getOutputStream());
reader = new InputStreamReader (client.getInputStream());
in = new BufferedReader (reader);
```

Now you can read and write to the socket, thus, communicating with the server.

```
out.println("Watson!" + "Come here...I need you!");
String data = in.readLine();
```

2.4 Low Level Data Communication

Data communications are the exchange of data between two devices via some form of transmission medium such as a wire cable.

TCP/IP(Transmission Control Protocol/Internet Protocol)

- The Protocol upon which the whole Internet is based
  - Each node must be configured for TCP/IP to function properly.
- A software-based protocol
- TCP/IP is basically the binding together of Internet Protocols used to connect hosts on the internet- Main ones are IP and TCP
- TCP and IP have special packet structure

- IP (Internet Protocol) is responsible for delivering packets of data between systems on the internet and specifies their format. Packets forwarded based on a four byte destination IP address (IP number)
- IP DOES NOT MAKE GUARANTEES! It is very simple - essentially: *send and forget*.
- TCP (Transmission Control Protocol) is responsible for verifying the correct delivery of data/packets from client to server. Data can be lost - so TCP also adds support to detect errors and retransmit data until completely received.
- Together these help form TCP/IP - a means of specifying packets, and delivering them safely.
- There are other protocols in TCP/IP - such as User Datagram Protocol UDP. UDP is a simpler alternative to TCP for aiding the delivery of packets. It makes no guarantees regarding delivery - but does guarantee *data integrity.* UDP also has no flow control, i.e. if messages are sent too quickly, data may be lost

2.4.1 IP Packet Structure

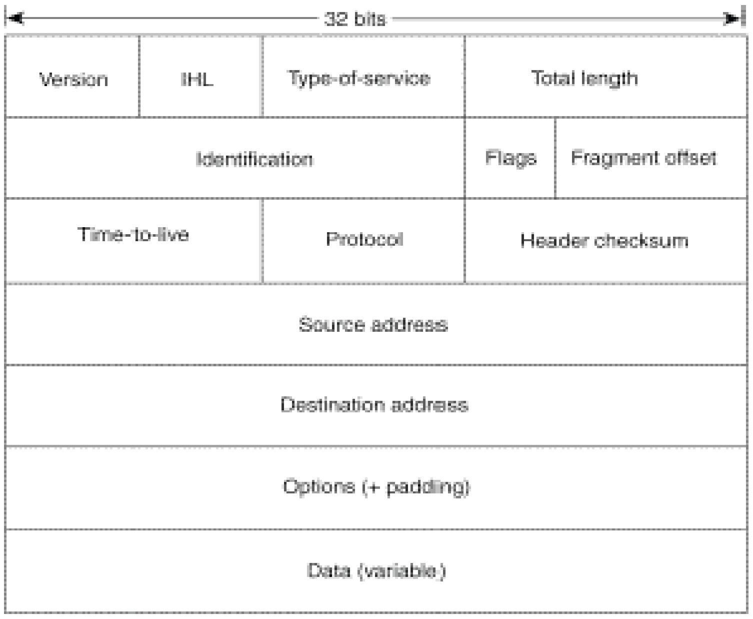


Figure.2.2. IP Packet Structure

IP uses a *Datagram* to transfer *packets* between *end systems* (usually computers) using *routers*. There are fourteen fields in an IP Packet .

Version — Indicates the version of IP currently used.

IP Header Length (IHL) — Indicates the datagram header length in 32-bit words.

Type-of-Service — Specifies how an upper-layer protocol would like a current datagram to be handled, and assigns datagram various levels of importance.

Total Length — Specifies the length, in bytes, of the entire IP packet, including the data and header.

Identification — Contains an integer that identifies the current datagram. This field is used to help piece together datagram fragments.

Flags — Consists of a 3-bit field of which the two low-order (least-significant) bits control fragmentation. The low-order bit specifies whether the packet can be fragmented. The middle bit specifies whether the packet is the last fragment in a series of fragmented packets. The third or high-order bit is not used.

Fragment Offset — Indicates the position of the fragment's data relative to the beginning of the data in the original datagram, which allows the destination IP process to properly reconstruct the original datagram.

Time-to-Live — Maintains a counter that gradually decrements down to zero, at which point the datagram is discarded. This keeps packets from looping endlessly.

Protocol — Indicates which upper-layer protocol receives incoming packets after IP processing is complete.

Header Checksum — Helps ensure IP header integrity.

Source Address — Specifies the sending node.

Destination Address — Specifies the receiving node.

Options — Allows IP to support various options, such as security.

Data — Contains upper-layer sent in packet.

2.4.2 TCP Packet Structure:

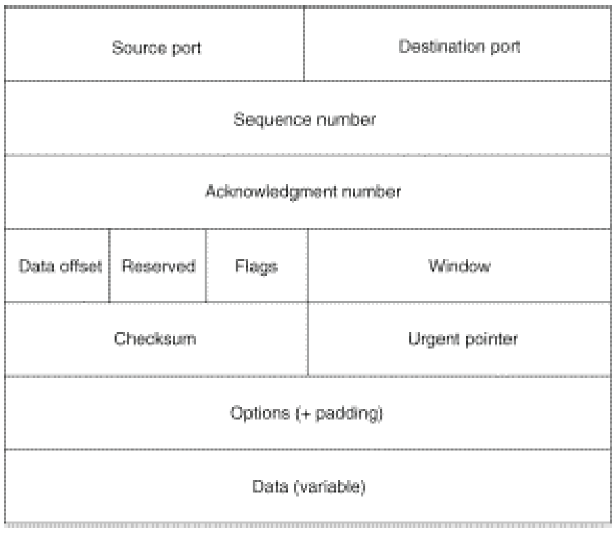


Figure.2.2. TCP Packet Structure

There are 12 fields in TCP Packet:

Source Port and **Destination Port** — Identifies points at which upper-layer source and destination processes receive TCP services.

Sequence Number — Usually specifies the number assigned to the first byte of data in the current message. In the connection-establishment phase, this field also can be used to identify an initial sequence number to be used in an upcoming transmission.

Acknowledgment Number — Contains the sequence number of the next byte of data the sender of the packet expects to receive.

Data Offset — Indicates the number of 32-bit words in the TCP header.

Reserved — Remains reserved for future use.

Flags — Carries a variety of control information, including the SYN and ACK bits used for connection establishment, and the FIN bit used for connection termination.

Window — Specifies the size of the sender's receive window (that is, the buffer space available for incoming data).

Checksum — Indicates whether the header was damaged in transit.

Urgent Pointer — Points to the first urgent data byte in the packet.

Options — Specifies various TCP options.

Data — Contains upper-layer sent in packet.

Chapter 3

Data Mapping and Exchange: Metadata; Data representation and encoding; XML, DTD, XML schemas

3.1 Data Mapping and Exchange

Data Mapping:

In computing and data management, **data mapping** is the process of creating data element mappings between two distinct data models

In metadata, the term **data element** is an atomic unit of data that has precise meaning or precise semantics. A data element has:

1. An identification such as a data element name
2. A clear data element definition
3. One or more representation terms
4. Optional enumerated values Code (metadata)
5. A list of synonyms to data elements

A **data model** organizes data elements and standardizes how the data elements relate to one another. Since data elements document real life people, places and things and the events between them, the data model represents reality.

Why Need Data Mapping

Data mapping is used as a first step for a wide **variety of data integration** tasks including:

- Data transformation or data mediation between data source and destination
- Identification of data relationships
- Discovery of hidden sensitive data
- Consolidation of multiple databases into a single data base and identifying redundant columns of data for consolidation or elimination

Data integration involves combining data residing in different sources and providing users with a unified view of these data. This process becomes significant in a variety of situations, which include both commercial (when two similar companies need to merge their databases) and scientific (combining research results from different bioinformatics repositories) domains.

Data Exchange:

Data exchange is the process of taking data structured under a *source* schema and actually transforming it into data structured under a *target* schema, so that the target data is an accurate representation of the source data.

Data Exchange Format

Often there are a few dozen different source and target schema (proprietary data formats) in some specific domain. Often people develop a **exchange format** or **interchange format** for some single domain, and then write a few dozen different routines to (indirectly) translate each and every source schema to each and every target schema by using the interchange format as an intermediate step. That requires a lot less work than writing and debugging the hundreds of different routines that would be required to directly translate each and every source schema directly to each and every target schema.

Example

Standard Interchange Format for geospatial data, Data Interchange Format for spreadsheet data, Quicken Interchange Format for financial data etc.

Data Exchange Language

A data exchange language is a language that is domain-independent and can be used for any kind of data. The term is also applied to any file format that can be read by more than one program, including proprietary formats such as Microsoft Office documents. Some of the formal languages are better suited for this task than others, since their specification is driven by a formal process instead of a particular software implementation needs.

Example

Resource Description Framework (RDF), JSON (JavaScript Object Notation), Rebol, YAML, Gellish, XML

3.2 Metadata

Metadata (metacontent) is defined as the data providing information about one or more aspects of the data, such as:

- Means of creation of the data
- Purpose of the data
- Time and date of creation
- Creator or author of the data
- Location on a computer network where the data were created

Example

Digital image may include metadata that describe the picture size, the color depth, the image resolution, time and date of image creation.

A text document's metadata may contain information about how long the document is, who the author is, when the document was written, and a short summary of the document.

3.3 Introduction to XML

- XML stands for Extensible Markup Language
- XML is a markup language much like HTML
- XML was designed to describe data, not to display data
- XML tags are not predefined. You must define your own tags
- XML is designed to be self-descriptive
- XML is a W3C Recommendation
- XML does not DO anything

Difference between XML and HTML

- XML is not a replacement for HTML; XML is a complement to HTML.
- ***XML is a software- and hardware-independent tool for carrying information.***
- XML was designed to describe data, with focus on what data is

- HTML was designed to display data, with focus on how data looks

XML Does Not DO Anything:

The following example is a note to Tove, from Jani, stored as XML:

```
<note>
 <to>Tove</to>
 <from>Jani</from>
 <heading>Reminder</heading>
 <body>Don'tforget me this weekend!</body>
</note>
```

The note above is quite self descriptive. It has sender and receiver information, it also has a heading and a message body.

But still, this XML document does not DO anything. It is just information wrapped in tags. ***Someone must write a piece of software to send, receive or display it.***

How Can XML be used?

XML is used in many aspects of web development, often to simplify data storage and sharing.

1. XML Separates Data from HTML
2. XML Simplifies Data Sharing
3. XML Simplifies Data Transport
4. XML Simplifies Platform Changes

Internet Languages Written in XML

Several Internet languages are written in XML. Here are some examples: XHTML, XML Schema, SVG, WSDL and RSS

3.4 XML Tree

XML Documents Form a Tree Structure

- XML documents must contain a **root element**. This element is "the parent" of all other elements.
- The elements in an XML document form a document tree. The tree starts at the root and branches to the lowest level of the tree.
- All elements can have sub elements (child elements):

-
  ```
  <root>
    <child>
      <subchild>.....</subchild>
    </child>
  </root>
  ```
- The terms parent, child, and sibling are used to describe the relationships between elements. Parent elements have children. Children on the same level are called siblings (brothers or sisters).
- All elements can have text content and attributes (just like in HTML).

XML element

- An XML document contains XML Elements.
- An XML element is everything from (including) the element's start tag to (including) the element's end tag.
- An element can contain:
  - other elements

- text
- attributes
- or a mix of all of the above...

Empty XML Elements

An alternative syntax can be used for XML elements with no content: Instead of writing a book element (with no content) like this:

<book></book>

It can be written like this:

<book />

This sort of element syntax is called self-closing.

XML Naming Rules

XML elements must follow these naming rules:

- Names can contain letters, numbers, and other characters
- Names cannot start with a number or punctuation character
- Names cannot start with the letters xml (or XML, or Xml, etc)
- Names cannot contain spaces

Any name can be used, no words are reserved.

XML Attributes

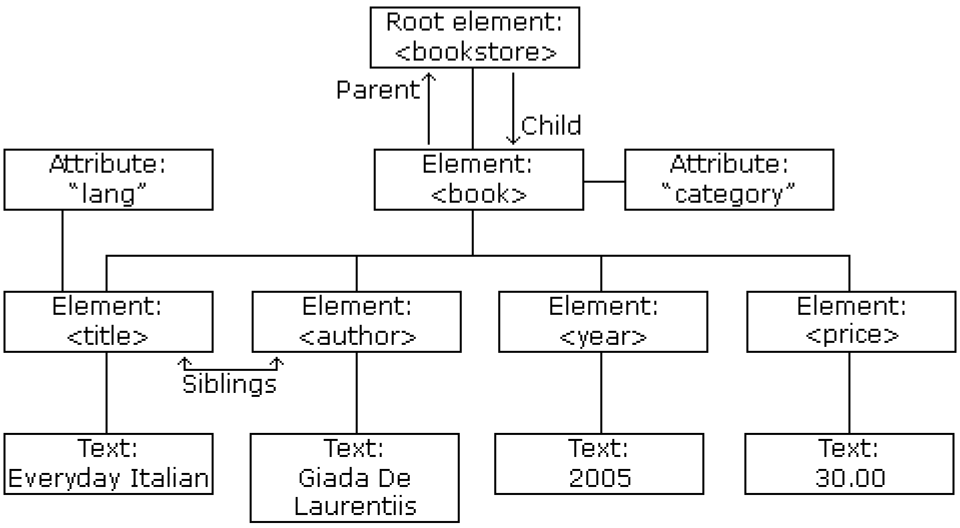
XML elements can have attributes, just like HTML. Attributes provide additional information about an element. Attributes often provide information that is not a part of the data. In the example below, the file type is irrelevant to the data, but can be important to the software that wants to manipulate the element:

<file type="gif">computer.gif</file>

<img src="computer.gif">

<a href="demo.asp">

Example:



- The image above represents one book in the XML below:

-
  ```
  <bookstore>
    <book category="COOKING">
      <title lang="en">Everyday Italian</title>
      <author>Giada De Laurentiis</author>
      <year>2005</year>
      <price>30.00</price>
    </book>
    <book category="Programming Language">
      <title lang="en">XML</title>
      <author>Dr.J.VijiPriya </author>
      <year>2014</year>
      <price>150</price>
    </book>
   </bookstore>
  ```
- The root element in the example is <bookstore>. All <book> elements in the document are contained within <bookstore>.
- The <book> element has 4 children: <title>,< author>, <year>, <price>.

3.5 XML Syntax Rules

The syntax rules of XML are very simple and logical.

1. All XML Elements Must Have a Closing Tag

- In HTML, some elements do not have to have a closing tag:
- <p>This is a paragraph.
  <br>
- In XML, it is illegal to omit the closing tag. All elements **must** have a closing tag:
- <p>This is a paragraph.</p>
  <br />

2. XML Tags are Case Sensitive

- XML tags are case sensitive. The tag <Letter> is different from the tag <letter>.
- Opening and closing tags must be written with the same case:
- <Message>This is incorrect</message>
  <message>This is correct</message>

3. XML Elements Must be Properly Nested

- In HTML, you might see improperly nested elements:
- <b><i>This text is bold and italic</b></i>
- In XML, all elements **must** be properly nested within each other:
- <b><i>This text is bold and italic</i></b>
- In the example above, "Properly nested" simply means that since the <i> element is opened inside the <b> element, it must be closed inside the <b> element.

4. XML Documents Must Have a Root Element

- XML documents must contain one element that is the **parent** of all other elements. This element is called the **root** element.

-
  ```
  <root>
    <child>
      <subchild>.....</subchild>
    </child>
  </root>
  ```

5. XML Attribute Values Must be Quoted

- XML elements can have attributes in name/value pairs just like in HTML.
- In XML, the attribute values must always be quoted.
- <note date=12/11/2007>
  <to>Tove</to>
  <from>Jani</from>
  </note>
- <note date="12/11/2007">
  <to>Tove</to>
  <from>Jani</from>
  </note>
- The error in the first document is that the date attribute in the note element is not quoted.

6. Entity References

- Some characters have a special meaning in XML.
- If you place a character like "<" inside an XML element, it will generate an error because the parser interprets it as the start of a new element.
- This will generate an XML error:
- <message>if salary < 1000 then</message>
- To avoid this error, replace the "<" character with an **entity reference**:
- <message>if salary &lt; 1000 then</message>
- There are 5 predefined entity references in XML:

&lt;	<	less than
&gt;	>	greater than
&amp;	&	ampersand
&apos;	'	Apostrophe
&quot;	"	quotation mark

Note: Only the characters "<" and "&" are strictly illegal in XML. The greater than character is legal, but it is a good habit to replace it.

7. Comments in XML

- The syntax for writing comments in XML is similar to that of HTML.
- <!-- This is a comment -->

8. White-space is preserved in XML

- HTML truncates multiple white-space characters to one single white-space:

HTML:	Hello Tove
Output:	Hello Tove

- With XML, the white-space in a document is not truncated.

9. XML Stores New Line as LF

- Windows applications store a new line as: carriage return and line feed (CR+LF).
- Unix and Mac OSX uses LF.
- Old Mac systems uses CR.
- XML stores a new line as LF.

10. Well Formed XML

- XML documents that conform to the syntax rules above are said to be "Well Formed" XML documents.

3.6 XML Declaration or XML Prolog

Example 1: A Sample XML Document (example1.xml)

```
<?xml version="1.0" encoding="UTF-8"?>
<document>
   <heading> Hello From XML    </heading>
   <message> This is an XML document! </message>
</document>
```

Like all XML documents, this one starts with an XML declaration, <?xml version="1.0" encoding="UTF-8"?>. This XML declaration indicates that we're using XML version 1.0, and using the UTF-8 character encoding,

This XML declaration, <?xml?>, uses two attributes, **version** and **encoding**, to set the version of XML and the character set we're using. Next we create a new XML element named <document>. XML tags themselves always start with < and end with >.Then we store other elements in our <document> element, or text data, as we wish.

Character Encodings: ASCII, Unicode, and UCS

The characters in an XML document are stored using numeric codes. That can be an issue, because different character sets use different codes, which means an XML processor might have problems trying to read an XML document that uses a character set called a character encoding

Which character sets are supported in XML? ASCII? Unicode? UCS?

There are many character encodings that an XML processor can support, such as the following:

- US-ASCII— U.S. ASCII
- UTF-8— Compressed Unicode
- UTF-16— Compressed UCS
- ISO-10646-UCS-2— Unicode
- ISO-10646-UCS-4— UCS
- ISO-2022-JP— Japanese
- ISO-2022-CN— Chinese
- ISO-8859-5— ASCII and Cyrillic

3.7 Cascading Style Sheets (CSS)

One of the most popular reasons for using style sheets with XML is that you store your data in an XML document, and specify how to display that data using a separate document, the style sheet shown in Figure 3.1. By separating the presentation details from the data, you can change the entire presentation with a few changes in the style sheet, instead of making multiple changes in your data itself.

There's plenty of support for working with XML documents and style sheets in both **Internet Explorer and Netscape Navigator**. There are **two kinds of style sheets** you can use with XML document:

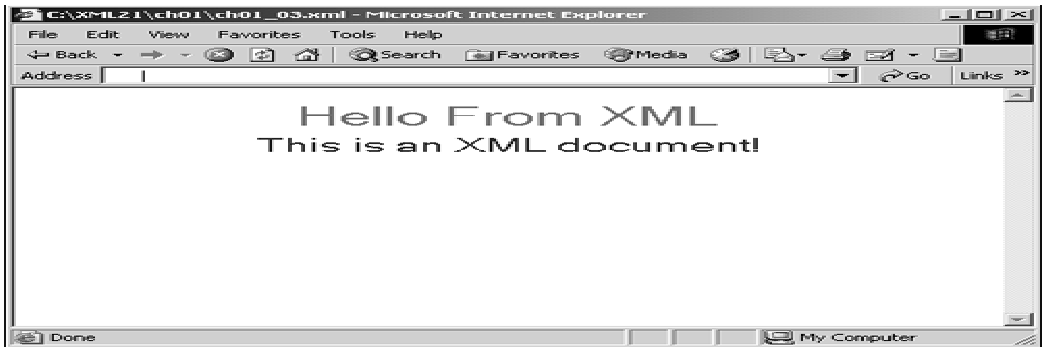
1. **Cascading Style Sheets (CSS)**, which you can also use with HTML documents
2. **Extensible Style sheet Language** style sheets **(XSL)**, designed to be used only with XML documents

Example 2: An XML Document Using a Style Sheet (example2.xml)

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/css" href="css1.css"?>
<document>
   <heading>  Hello From XML    </heading>
   <message> This is an XML document!   </message>
</document>
```

A CSS Style Sheet (css1.css)

```
heading {display: block; font-size: 24pt; color: #ff0000; text-align: center}
message {display: block; font-size: 18pt; color: #0000ff; text-align: center}
```


C:\XML21\ch01\ch01_03.xml - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address
Go Links
Done
My Computer

Hello From XML
This is an XML document!

Figure 3.1: Viewing an XML document in Internet Explorer viewing

Extracting Data from an XML Document

You can extract data from an XML document yourself using Scripting language like java script and to work with that data, rather than simply telling a browser how to display it. For example, suppose you want to extract the text from our <heading> element shown in Figure 3.2:

Example 3: (example3.xml)

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/css" href="css1.css"?>
<document>
  <heading>
    Hello From XML
  </heading>
  <message>
    This is an XML document!
  </message>
</document>
```

Extracting Data from an XML Document Using JavaScript (example3.html)

```
<HTML>
  <HEAD>
     <TITLE>
       Retrieving data from an XML document
     </TITLE>

     <XML ID="firstXML" SRC="example3.xml"></XML>

     <SCRIPT LANGUAGE="JavaScript">
       function getData()
       {
          xmldoc= document.all("firstXML").XMLDocument;

          nodeDoc = xmldoc.documentElement;
          nodeHeading = nodeDoc.firstChild;

          outputMessage = "Heading: " +
              nodeHeading.firstChild.nodeValue;
          message.innerHTML=outputMessage;
       }
     </SCRIPT>
  </HEAD>

  <BODY>
    <CENTER>
      <H1>
         Retrieving data from an XML document
      </H1>

      <DIV ID="message"></DIV>
      <P>
      <INPUT TYPE="BUTTON" VALUE="Read the heading" ONCLICK="getData()">
    </CENTER>
  </BODY>
</HTML>
```

Extracting data from an XML document in Internet Explorer

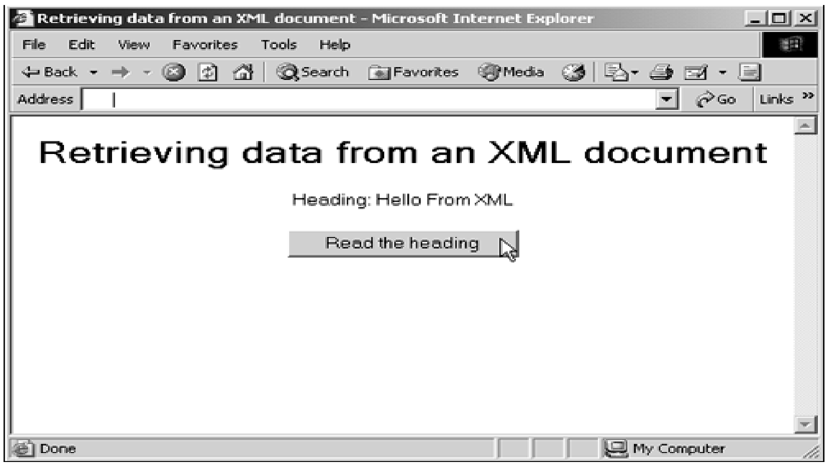

Retrieving data from an XML document - Microsoft Internet Explorer
Retrieving data from an XML document
Heading: Hello From XML
Read the heading

Figure 3.2. Extracting data from an XML document in Internet Explorer

3.8 XML DTD

How does an XML processor check your document?

There are two main checks that XML processors make:

1. Checking that your document is well-formed
2. Checking that it's valid

Why need XML Validator

- Use our XML validator to syntax-check your XML.
- Errors in XML documents will stop your XML applications.
- The W3C XML specification states that a program should stop processing an XML document if it finds an error.
- HTML browsers will display HTML documents with errors (like missing end tags). **With XML, errors are not allowed.**

Creating Valid XML Documents

An XML processor will usually check whether your XML document is well-formed, but only some are also capable of checking whether it's valid that is syntax in either a Document Type Definition (DTD) or an XML schema.

XML DTD:

- An XML document with correct syntax is called "Well Formed".
- An XML document validated against a DTD is "Well Formed" and "Valid".
- The purpose of a DTD is to **define the structure of an XML document**. It defines the **structure with a list of legal elements:**

As an example, you can see how you add a DTD to our XML document. DTDs can be separate documents, or they can be built into an XML document as we've done here using a **special element named <!DOCTYPE>.**

An XML Document with a DTD (example4.xml)

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/css" href="css1.css"?>
<!DOCTYPE document
 [
  <!ELEMENT document (heading, message)>
  <!ELEMENT heading (#PCDATA)>
  <!ELEMENT message (#PCDATA)>
]>
<document>
  <heading>
    Hello From XML
  </heading>
  <message>
    This is an XML document!
  </message>
</document>
```

Valid XML Document with DTD (example.5.xml)

A "Valid" XML document is a "Well Formed" XML document, which also conforms to the rules of a DTD:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE note SYSTEM "Note.dtd">
<note>
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
</note>
```

The DOCTYPE declaration, in the example above, is a reference to an external DTD file **"Note.dtd"**.

The content of the file is shown in below:

Note.dtd

```
<!DOCTYPE note
[
<!ELEMENT note (to,from,heading,body)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT from (#PCDATA)>
<!ELEMENT heading (#PCDATA)>
<!ELEMENT body (#PCDATA)>
]>
```

The DTD above is interpreted like this:

- !DOCTYPE note defines that the root element of the document is note
- !ELEMENT note defines that the note element contains four elements: "to, from, heading, body"
- !ELEMENT to defines the to element to be of type "#PCDATA"
- !ELEMENT from defines the from element to be of type "#PCDATA"
- !ELEMENT heading defines the heading element to be of type "#PCDATA"
- !ELEMENT body defines the body element to be of type "#PCDATA"

Note

#PCDATA means **parse-able text data.**

When NOT to Use a Document Definition?

When you are working with small XML files, creating document definitions may be a waste of time.

3.9 XML Schema

- Another way of validating XML documents: using XML schemas.
- The XML Schema language is also referred to as **XML Schema Definition (XSD),** describes the structure of an XML document.
- defines the legal building blocks (elements and attributes) of an XML document like DTD.
- defines which elements are child elements
- defines the number and order of child elements
- defines whether an element is empty or can include text
- defines data types for elements and attributes
- defines default and fixed values for elements and attributes

It is believed that XML Schemas will be used in most Web applications as a replacement for DTDs.

Here are some reasons:

- XML Schemas are extensible to future additions
- XML Schemas are richer and more powerful than DTDs
- XML Schemas are written in XML
- XML Schemas support data types and namespaces

Creating XML Schemas by Using XML Schema-Creation Tools

Software tools that can generate XML schemas for you. A growing number of XML schema-creation tools are:

- HiT Software- online automatic XML schema generator and DTD to XML schema converter. You just let it upload a document, and it creates an XML schema for free.
- xmlArchitect- XML editor for creating schemas.
- XMLspy- XMLspy is a product family of tools that aid in the creation of XML schemas.
- XML Ray-This tool provides support for XML schemas and has an integrated online XML tutorial system.
- Microsoft Visual Studio .NET-Visual Studio .NET can also generate XML schemas for you automatically.

XSD Simple Element

A simple element is an XML element that can contain only text. It cannot contain any other elements or attributes. The text can be of many different types. It can be one of the types included in the XML Schema definition (boolean, string, date, etc.), or it can be a custom type that you can define yourself. You can also add restrictions (facets) to a data type in order to limit its content, or you can require the data to match a specific pattern.

The syntax for defining a simple element is:

```
<xs:element name="xxx" type="yyy"/>
```

Where xxx is the name of the element and yyy is the data type of the element.

XML Schema has a lot of built-in data types. The most common types are:

- xs:string
- xs:decimal
- xs:integer
- xs:boolean
- xs:date
- xs:time

Example

Here are some XML elements:

```
<lastname>Refsnes</lastname>
<age>36</age>
<dateborn>1970-03-27</dateborn>
```

And here are the corresponding simple element definitions:

```
<xs:element name="lastname" type="xs:string"/>
<xs:element name="age" type="xs:integer"/>
<xs:element name="dateborn" type="xs:date"/>
```

Default and Fixed Values for Simple Elements

Simple elements may have a default value or a fixed value specified.

1. A default value is automatically assigned to the element when no other value is specified. In the following example the default value is "red":

   `<xs:element name="color" type="xs:string" default="red"/>`

2. A fixed value is also automatically assigned to the element, and you cannot specify another value.

   In the following example the fixed value is "red":

   `<xs:element name="color" type="xs:string" fixed="red"/>`

XSD Attributes

All attributes are declared as simple types. Simple elements cannot have attributes. If an element has attributes, it is considered to be of a complex type. But the attribute itself is always declared as a simple type.

The syntax for defining an attribute is:

`<xs:attribute name="xxx" type="yyy"/>`

where xxx is the name of the attribute and yyy specifies the data type of the attribute.

Example

Here is an XML element with an attribute:

`<lastname lang="EN">Smith</lastname>`

And here is the corresponding attribute definition in XML schema:

`<xs:attribute name="lang" type="xs:string"/>`

Default and Fixed Values for Attributes

Attributes may have a default value or a fixed value specified.

In the following example the default value is "EN":

```
<xs:attribute name="lang" type="xs:string" default="EN"/>
```

In the following example the fixed value is "EN":

```
<xs:attribute name="lang" type="xs:string" fixed="EN"/>
```

Optional and Required Attributes

Attributes are optional by default. To specify that the attribute is required, use the "use" attribute:

```
<xs:attribute name="lang" type="xs:string" use="required"/>
```

XSD Complex Elements

A complex element is an XML element that contains other elements and/or attributes.

There are four kinds of complex elements:

1. empty elements
2. elements that contain only other elements
3. elements that contain only text
4. elements that contain both other elements and text

Note: Each of these elements may contain attributes as well!

Examples of Complex Elements

A complex XML element, "product", **which is empty:**

```
<product pid="1345"/>
```

A complex XML element, "employee", **which contains only other elements:**

```
<employee>
  <firstname>John</firstname>
  <lastname>Smith</lastname>
</employee>
```

A complex XML element, "food", **which contains only text:**

```
<food type="dessert">Ice cream</food>
```

A complex XML element, "description", **which contains both elements and text:**

```
<description>

It happened on <date lang="norwegian">03.03.99</date>

</description>
```

XSD Elements Only

How to Define a Complex Element using XML Scheme

Look at this complex XML element, "employee", which contains only other elements:

```
<employee>
  <firstname>John</firstname>
  <lastname>Smith</lastname>
</employee>
```

The "employee" element can be declared directly by naming the element, like this:

```
<xs:element name="employee">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="firstname" type="xs:string"/>
      <xs:element name="lastname" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

If you use the method described above, only the "employee" element can use the specified complex type. Note that the child elements, "firstname" and "lastname", are surrounded by the <sequence> indicator. This means that the child elements must appear in the same order as they are declared. The "employee" element can have a type attribute that refers to the name of the complex type to use:

XSD Empty Elements

An empty complex element cannot have contents, only attributes.

An empty XML element:

```
<product prodid="1345" />
```

It is possible to declare the "product" element more compactly, like this:

```
<xs:element name="product">
  <xs:complexType>
    <xs:attribute name="prodid" type="xs:positiveInteger"/>
  </xs:complexType>
</xs:element>
```

XSD Indicators

We can control HOW elements are to be used in documents with indicators.

Order Indicators

Order indicators are used to define the order of the elements.

Order indicators are:

- All
- Choice
- Sequence

All Indicator

The <all> indicator specifies that the child elements can appear in any order, and that each child element must occur only once:

```
<xs:element name="person">
  <xs:complexType>
    <xs:all>
      <xs:element name="firstname" type="xs:string"/>
      <xs:element name="lastname" type="xs:string"/>
    </xs:all>
  </xs:complexType>
</xs:element>
```

Choice Indicator

The <choice> indicator specifies that either one child element or another can occur:

```
<xs:element name="person">
  <xs:complexType>
    <xs:choice>
      <xs:element name="employee" type="employee"/>
      <xs:element name="member" type="member"/>
    </xs:choice>
  </xs:complexType>
</xs:element>
```

Sequence Indicator

The <sequence> indicator specifies that the child elements must appear in a specific order:

```
<xs:element name="person">
   <xs:complexType>
    <xs:sequence>
      <xs:element name="firstname" type="xs:string"/>
      <xs:element name="lastname" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

An XML Document

Let's have a look at this XML document called "shiporder.xml":

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<shiporder orderid="889923">
  <orderperson>John Smith</orderperson>
  <shipto>
    <name>Ola Nordmann</name>
    <address>Langgt 23</address>
    <city>4000 Stavanger</city>
    <country>Norway</country>
  </shipto>
 </shiporder>
```

The XML document above consists of a root element, "shiporder", that contains a required attribute called "orderid". The "shiporder" element contains child elements: "orderperson" and “shipto”.

Create an XML Schema

Now we want to create a schema for the XML document above. We start by opening a new file that we will call "shiporder.xsd". To create the schema we could simply follow the structure in the XML document and define each element as we find it. We will start with the standard **XML declaration followed by the xs:schema element** that defines a schema:

```
<?xml version="1.0" encoding="UFT-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
```

```
...
</xs:schema>
```

In the schema above we use the standard namespace (xs), and the URI associated with this namespace is the Schema language definition, which has the standard value of http://www.w3.org/2001/XMLSchema.

Next, we have to define the **"shiporder" element**. This element has an attribute and it contains other elements, therefore we consider it as a **complex type**. The child elements of the "shiporder" element is surrounded by a **xs:sequence element** that defines an ordered sequence of sub elements:

```
<xs:element name="shiporder">
  <xs:complexType>
    <xs:sequence>
      ...
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Then we have to define the **"orderperson" element as a simple type** (because it does not contain any attributes or other elements). The type (xs:string) is prefixed with the namespace. The prefix associated with XML Schema that indicates a predefined schema data type:

```
<xs:element name="orderperson" type="xs:string"/>
```

Next, we have to define two elements that are of the complex type: "shipto". We start by defining the "shipto" element:

```
<xs:element name="shipto">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="address" type="xs:string"/>
      <xs:element name="city" type="xs:string"/>
      <xs:element name="country" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

We can now declare the attribute of the "shiporder" element. Since this is a required attribute we specify use="required".

Note: The attribute declarations must always come last:

```
<xs:attribute name="orderid" type="xs:string" use="required"/>
```

Here is the complete listing of the schema file called "shiporder.xsd":

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

<xs:element name="shiporder">
  <xs:complexType>
    <xs:sequence>

      <xs:element name="orderperson" type="xs:string"/>
  <xs:element name="shipto">
   <xs:complexType>

    <xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="address" type="xs:string"/>
      <xs:element name="city" type="xs:string"/>
      <xs:element name="country" type="xs:string"/>
    </xs:sequence>

   </xs:complexType>
  </xs:element>

</xs:sequence>
<xs:attribute name="orderid" type="xs:string" use="required"/>
</xs:complexType>
</xs:element>

</xs:schema>
```

An XSD Example

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

```
<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>
```

The Schema above is interpreted like this:

- <xs:element name="note"> defines the element called "note"
- <xs:complexType> the "note" element is a complex type
- <xs:sequence> 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

Everything is wrapped in "Well Formed" XML.

3.10 XML Parser (Parsing XML documents)

All modern browsers have a built-in XML parser. An XML parser converts an XML document into an XML DOM object - which can then be manipulated with JavaScript.

Parse an XML Document

The following code fragment parses an XML document into an XML DOM object:

```
if (window.XMLHttpRequest)
  {// code for IE7+, Firefox, Chrome, Opera, Safari
  xmlhttp=new XMLHttpRequest();
  }
else
  {// code for IE6, IE5
  xmlhttp=new ActiveXObject("Microsoft.XMLHTTP");
  }
xmlhttp.open("GET","books.xml",false);
xmlhttp.send();
xmlDoc=xmlhttp.responseXML;
```

XML DOM

The XML DOM defines a standard way for accessing and manipulating XML documents. The XML DOM views an XML document as a tree-structure. All elements can be accessed through the DOM tree. Their content (text and attributes) can be modified or deleted, and new elements can be created. The elements, their text, and their attributes are all known as nodes.

The HTML DOM

The HTML DOM defines a standard way for accessing and manipulating HTML documents.

All HTML elements can be accessed through the HTML DOM.

Load an XML File - Cross-browser Example

The following example parses an XML document ("note.xml") into an XML DOM object and then extracts some information from it with a JavaScript:

Example

```
<html>
<body>

 <span id="to"></span>
<span id="from"></span>
<span id="message"></span>

<script>
if (window.XMLHttpRequest)
  {// code for IE7+, Firefox, Chrome, Opera, Safari
  xmlhttp=new XMLHttpRequest();
  }
else
  {// code for IE6, IE5
  xmlhttp=new ActiveXObject("Microsoft.XMLHTTP");
  }
xmlhttp.open("GET","note.xml",false);
xmlhttp.send();
xmlDoc=xmlhttp.responseXML;

document.getElementById("to").innerHTML=
        xmlDoc.getElementsByTagName("to")[0].childNodes[0].nodeValue;
document.getElementById("from").innerHTML=
        xmlDoc.getElementsByTagName("from")[0].childNodes[0].nodeValue;
document.getElementById("message").innerHTML=
        xmlDoc.getElementsByTagName("message")[0].childNodes[0].nodeValue;
</script>

</body>
</html>
```

Important Note!

To extract the text "Tove" from the <to> element in the XML file above ("note.xml"), the syntax is: getElementsByTagName("to")[0].childNodes[0].nodeValue Notice that even if the XML file contains only ONE <to> element you still have to specify the array index [0]. This is because the getElementsByTagName() method returns an array.

Chapter 4

XSL, XSLT and XPath

4.1 XSL

- XSL stands for E**X**tensible **S**tylesheet **L**anguage.
- It is an XML-based Stylesheet Language.
- XSL describes how the XML document should be displayed
- XSL consists of three parts:
  - XSLT - a language for transforming XML documents
  - XPath - a language for navigating in XML documents
  - XSL-FO - a language for formatting XML documents

4.2 XSLT

- XSLT stands for XSL Transformations,
- **XSLT transforms an XML source-tree into an XML result-tree**.
- XSLT transforms an XML document into another XML document that is recognized by a browser, like HTML and XHTML.
- With XSLT you can add/remove elements and attributes to or from the output file.
- With XSLT You can also rearrange and sort elements, perform tests and make decisions about which elements to hide and display, and a lot more.
- XSLT uses XPath to find information in an XML document.
- XPath is used to navigate through elements and attributes in XML documents.
- **How Does it Work?**
  - In the transformation process, XSLT uses XPath to define parts of the source document that should match one or more predefined templates.
  - When a match is found, XSLT will transform the matching part of the source document into the result document.
- All major browsers such as Internet Explorer ,Chrome, Firefox, Safari and Opera supports XML, XSLT, and XPath

The XML File

```
<?xml version="1.0" encoding="UTF-8"?>
<catalog>
  <cd>
    <title>Empire Burlesque</title>
    <artist>Bob Dylan</artist>
    <country>USA</country>
    <company>Columbia</company>
    <price>10.90</price>
    <year>1985</year>
  </cd>
.
.
</catalog>
```

XSLT <xsl:template> Element

- An XSL style sheet consists of one or more set of rules that are called templates.

- A template contains rules to apply when a specified node is matched.
- The <xsl:template> element is used to build templates.
- The **match** attribute
  - The **match** attribute is used to associate a template with an XML element.
  - The match attribute can also be used to define a template for the entire XML document.
  - The value of the match attribute is an XPath expression (i.e. match="/" defines the whole document).

Example

```
<?xml version="1.0" encoding="UTF-8"?>

<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
 <html>
 <body>
 <h2>My CD Collection</h2>
 <table border="1">
  <tr bgcolor="#9acd32">
   <th>Title</th>
   <th>Artist</th>
  </tr>
  <xsl:for-each select="catalog/cd">
   <xsl:sort select="artist"/>
   <tr>
    <td><xsl:value-of select="title"/></td>
    <td><xsl:value-of select="artist"/></td>
   </tr>
  </xsl:for-each>
 </table>
 </body>
 </html>
</xsl:template>
</xsl:stylesheet>
```

Example Explained

- Since an XSL style sheet is an XML document, it always begins with the XML declaration: **<?xml version="1.0" encoding="UTF-8"?>**.
- The next element, **<xsl:stylesheet>**, defines that this document is an XSLT style sheet document (along with the version number and XSLT namespace attributes).
- The **<xsl:template>** element defines a template. The **match="/"** attribute associates the template with the root of the XML source document.
- The content inside the <xsl:template> element defines some HTML to write to the output.
- The last two lines define the end of the template and the end of the style sheet.

XSLT <xsl:value-of> Element

- The <xsl:value-of> element can be used to extract the value of an XML element and add it to the output stream of the transformation:

Note: The **select** attribute, in the example above, contains an XPath expression. An XPath expression works like navigating a file system; a forward slash (/) selects subdirectories.

XSLT <xsl:for-each> and <xsl:sort> Element

- **<xsl:for-each>** element to loop through the XML elements, and display all of the records.
- The **<xsl:sort>** element is used to sort the output.
- To sort the output, simply add an <xsl:sort> element inside the <xsl:for-each> element in the XSL file:

XSLT <xsl:if> Element

- The <xsl:if> element is used to put a conditional test against the content of the XML file.
  Syntax
  <xsl:if test="*expression*">
    ...some output if the expression is true...
  </xsl:if>

- To add a conditional test, add the <xsl:if> element inside the <xsl:for-each> element in the XSL file instead of **<xsl:sort>** element in the above example :

Example

```
<?xml version="1.0" encoding="UTF-8"?>

<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">

 <html>
 <body>
 <h2>My CD Collection</h2>
 <table border="1">
   <tr bgcolor="#9acd32">
     <th>Title</th>
     <th>Artist</th>
     <th>Price</th>
   </tr>
   <xsl:for-each select="catalog/cd">
     <xsl:if  test="price &gt; 10">
       <tr>
         <td><xsl:value-of select="title"/></td>
         <td><xsl:value-of select="artist"/></td>
         <td><xsl:value-of select="price"/></td>
       </tr>
     </xsl:if>
   </xsl:for-each>
 </table>
 </body>
 </html>
</xsl:template>
</xsl:stylesheet>
```

Note: The value of the required **test attribute contains the expression to be evaluated**. The code above will only output the title and artist elements of the CDs that has a price that is higher than 10.

XSLT <xsl:choose> Element

- The <xsl:choose> element is used in conjunction with <xsl:when> and <xsl:otherwise> to express multiple conditional tests.

  Syntax

  ```
  <xsl:choose>
    <xsl:when test="expression">
      ... some output ...
    </xsl:when>
    <xsl:otherwise>
      ... some output ....
    </xsl:otherwise>
  </xsl:choose>
  ```

- To insert a multiple conditional test against the XML file, add the <xsl:choose>, <xsl:when>, and <xsl:otherwise> elements to the XSL file:

Example

```
<?xml version="1.0" encoding="UTF-8"?>

<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">

 <html>
 <body>
 <h2>My CD Collection</h2>
 <table border="1">
   <tr bgcolor="#9acd32">
     <th>Title</th>
     <th>Artist</th>
   </tr>
   <xsl:for-each select="catalog/cd">
   <tr>
     <td><xsl:value-of select="title"/></td>
     <xsl:choose>
       <xsl:when test="price &gt; 10">
         <td bgcolor="#ff00ff">
         <xsl:value-of select="artist"/></td>
       </xsl:when>

       <xsl:otherwise>
         <td><xsl:value-of select="artist"/></td>
       </xsl:otherwise>
     </xsl:choose>
   </tr>
   </xsl:for-each>
 </table>
 </body>
 </html>

</xsl:template>
</xsl:stylesheet>
```

The code above will add a pink background-color to the "Artist" column WHEN the price of the CD is higher than 10.

XSLT <xsl:apply-templates> Element

- The <xsl:apply-templates> element applies a template to the current element or it's child nodes.
- If we add a select attribute to the <xsl:apply-templates> element it will process only the child element that matches the value of the select attribute. We can use the select attribute to specify the order in which the child nodes are processed.

Example

```
<?xml version="1.0" encoding="UTF-8"?>

<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">

 <html>
 <body>
 <h2>My CD Collection</h2>
 <xsl:apply-templates/>
 </body>
 </html>
</xsl:template>

<xsl:template match="cd">
 <p>
 <xsl:apply-templates select="title"/>
 <xsl:apply-templates select="artist"/>
 </p>
</xsl:template>

<xsl:template match="title">
 Title: <span style="color:#ff0000">
 <xsl:value-of select="."/></span>
 <br />
</xsl:template>

<xsl:template match="artist">
 Artist: <span style="color:#00ff00">
 <xsl:value-of select="."/></span>
 <br />
</xsl:template>

</xsl:stylesheet>
```

A Sample XSL Style Sheet That Has Multiple Matches

In this Example which reads the data in the XML document and displays it in an HTML table—including the units for various values, as applicable. To extract some data from ex_01.xml, you need an XSLT style sheet, which is called ex_02.xsl. (XSLT style sheets usually use the extension .xsl.) This example just strips out the names of the states and places them into a basic HTML document.

A Sample XML Document (ex_01.xml)

```
<?xml version="1.0" encoding ="UTF-8"?>
<states>

  <state>
    <name>California</name>
    <population units="people">33871648</population><!--2000 census-->
    <capital>Sacramento</capital>
    <bird>Quail</bird>
    <flower>Golden Poppy</flower>
    <area units="square miles">155959</area>
  </state>

  <state>
    <name>Massachusetts</name>
    <population units="people">6349097</population><!--2000 census-->
    <capital>Boston</capital>
    <bird>Chickadee</bird>
    <flower>Mayflower</flower>
    <area units="square miles">7840</area>
  </state>

  <state>
    <name>New York</name>
    <population units="people">18976457</population><!--2000 census-->
    <capital>Albany</capital>
    <bird>Bluebird</bird>
    <flower>Rose</flower>
    <area units="square miles">47214</area>
  </state>


</states>
```

XSL Style Sheet (ex_02.xsl)

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="/states">
    <HTML>
      <HEAD>
        <TITLE>
          State Data
        </TITLE>
      </HEAD>
      <BODY>
```

```
        <H1>
          State Data
        </H1>
        <TABLE BORDER="1">
          <TR>
            <TD>Name</TD>
            <TD>Population</TD>
            <TD>Capital</TD>
            <TD>Bird</TD>
            <TD>Flower</TD>
            <TD>Area</TD>
          </TR>
          <xsl:apply-templates/>
        </TABLE>
      </BODY>
    </HTML>
 </xsl:template>

 <xsl:template match="state">
   <TR>
     <TD><xsl:value-of select="name"/></TD>
     <TD><xsl:apply-templates select="population"/></TD>
     <TD><xsl:apply-templates select="capital"/></TD>
     <TD><xsl:apply-templates select="bird"/></TD>
     <TD><xsl:apply-templates select="flower"/></TD>
     <TD><xsl:apply-templates select="area"/></TD>
   </TR>
</xsl:template>

 <xsl:template match="population">
    <xsl:value-of select="."/>
    <xsl:text> </xsl:text>
    <xsl:value-of select="@units"/>
 </xsl:template>

 <xsl:template match="capital">
    <xsl:value-of select="."/>
 </xsl:template>

 <xsl:template match="bird">
    <xsl:value-of select="."/>
 </xsl:template>

 <xsl:template match="flower">
    <xsl:value-of select="."/>
 </xsl:template>

 <xsl:template match="area">
    <xsl:value-of select="."/>
    <xsl:text> </xsl:text>
    <xsl:value-of select="@units"/>
 </xsl:template>
```

```
</xsl:stylesheet>
```

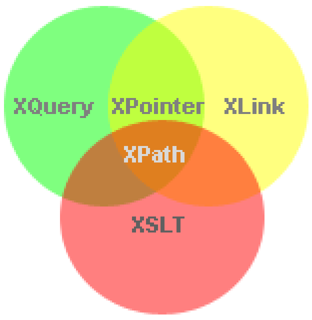
Note the expression "." here. You use "." with the **select attribute to specify the current node**. You can handle attributes very much like you handle elements. All that's different is that you have to preface the **attribute name with @.** For example, say that you want to recover the value of the units attributes in the <population> and <area> elements of the XML example. To get the values of the units attribute, you simply need to refer to it as @units.

Here's the HTML you get, including the HTML table:

```
<HTML>
  <HEAD>
    <META http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <TITLE>
      State Data
    </TITLE>
  </HEAD>

  <BODY>
    <H1>
      State Data
    </H1>
    <TABLE BORDER="1">
      <TR>
        <TD>Name</TD>
        <TD>Population</TD>
        <TD>Capital</TD>
        <TD>Bird</TD>
        <TD>Flower</TD>
        <TD>Area</TD>
      </TR>

      <TR>
        <TD>California</TD>
        <TD>33871648 people</TD>
        <TD>Sacramento</TD>
        <TD>Quail</TD>
        <TD>Golden Poppy</TD>
        <TD>155959 square miles</TD>
      </TR>

      <TR>
        <TD>Massachusetts</TD>
        <TD>6349097 people</TD>
        <TD>Boston</TD>
        <TD>Chickadee</TD>
        <TD>Mayflower</TD>
        <TD>7840 square miles</TD>
      </TR>
      <TR>
        <TD>New York</TD>
        <TD>18976457 people</TD>
        <TD>Albany</TD>
        <TD>Bluebird</TD>
        <TD>Rose</TD>
        <TD>47214 square miles</TD>
      </TR>
    </TABLE>
  </BODY>
</HTML>
```

4.3 XML and XPath

XPath (the XML Path language) is a language for finding information in an XML document.



- XPath is a syntax for defining parts of an XML document
- XPath uses path expressions to navigate in XML documents
- XPath contains a library of standard functions
- XPath is also used in XSLT, XQuery, XPointer and XLink
- Without XPath knowledge you will not be able to create XSLT documents.
- XPath is a W3C recommendation

XPath Path Expressions

XPath uses path expressions to select nodes or node-sets in an XML document. These path expressions look very much like the expressions you see when you work with a traditional computer file system. Today XPath expressions can also be used in JavaScript, Java, XML Schema, PHP, Python, C and C++, and lots of other languages.

XPath Example

We will use the following XML document:

```
<?xml version="1.0" encoding="UTF-8"?>
<bookstore>
<book category="COOKING">
  <title lang="en">Everyday Italian</title>
  <author>Giada De Laurentiis</author>
  <year>2005</year>
  <price>30.00</price>
</book>
<book category="CHILDREN">
  <title lang="en">Harry Potter</title>
  <author>J K. Rowling</author>
  <year>2005</year>
  <price>29.99</price>
</book>
```

```
<book category="WEB">
  <title lang="en">XQuery Kick Start</title>
  <author>James McGovern</author>
  <author>Per Bothner</author>
  <author>Kurt Cagle</author>
  <author>James Linn</author>
  <author>Vaidyanathan Nagarajan</author>
  <year>2003</year>
  <price>49.99</price>
</book>

<book category="WEB">
  <title lang="en">Learning XML</title>
  <author>Erik T. Ray</author>
  <year>2003</year>
  <price>39.95</price>
</book>
</bookstore>
```

In the table below we have listed some XPath expressions and the result of the expressions:

XPath Expression	Result
/bookstore/book[1]	Selects the first book element that is the child of the bookstore element
/bookstore/book[last()]	Selects the last book element that is the child of the bookstore element
/bookstore/book[last()-1]	Selects the last but one book element that is the child of the bookstore element
/bookstore/book[position()<3]	Selects the first two book elements that are children of the bookstore element
//title[@lang]	Selects all the title elements that have an attribute named lang
//title[@lang='eng']	Selects all the title elements that have an attribute named lang with a value of 'eng'
/bookstore/book[price>35.00]	Selects all the book elements of the bookstore element that have a price element with a value greater than 35.00
/bookstore/book[price>35.00]/title	Selects all the title elements of the book elements of the bookstore element that have a price element with a value greater than 35.00

Selecting Unknown Nodes

XPath wildcards can be used to select unknown XML elements.

Wildcard	Description
*	Matches any element node
@*	Matches any attribute node
node()	Matches any node of any kind

In the table below we have listed some path expressions and the result of the expressions:

Path Expression	Result
/bookstore/*	Selects all the child nodes of the bookstore element
//*	Selects all elements in the document
//title[@*]	Selects all title elements which have any attribute

Selecting Several Paths

By using the | operator in an XPath expression you can select several paths.

In the table below we have listed some path expressions and the result of the expressions:

Path Expression	Result
//book/title | //book/price	Selects all the title AND price elements of all book elements
//title | //price	Selects all the title AND price elements in the document
/bookstore/book/title | //price	Selects all the title elements of the book element of the bookstore element AND all the price elements in the document

Location Path Expression

A location path can be absolute or relative.

An absolute location path starts with a slash (/) and a relative location path does not. In both cases the location path consists of one or more steps, each separated by a slash:

An absolute location path: /step/step/...

A relative location path: step/step/...

Each step is evaluated against the nodes in the current node-set.

A step consists of:

- an axis (defines the tree-relationship between the selected nodes and the current node)
- a node-test (identifies a node within an axis)
- zero or more predicates (to further refine the selected node-set)

The syntax for a location step is:

axisname::nodetest[predicate]

Examples

Child::para [position()=1]

Example	Result
child::book	Selects all book nodes that are children of the current node
attribute::lang	Selects the lang attribute of the current node
child::*	Selects all element children of the current node
attribute::*	Selects all attributes of the current node
child::text()	Selects all text node children of the current node
child::node()	Selects all children of the current node
descendant::book	Selects all book descendants of the current node
ancestor::book	Selects all book ancestors of the current node
ancestor-or-self::book	Selects all book ancestors of the current node - and the current as well if it is a book node

child::*/child::price	Selects all price grandchildren of the current node

XPath Axes and Node Tests

XPath Axes

An axis defines a node-set relative to the current node.

In the location step child::bird, which refers to a <bird> element that is a child of the current node, child is called the axis. XPath supports many different axes, and it's important to know what they are. Here's the list:

- ancestor— This axis contains the ancestors of the context node. An ancestor node is the parent of the context node, the parent of the parent, and so forth, back to (and including) the root node.
- ancestor-or-self— This axis contains the context node and the ancestors of the context node.
- attribute— This axis contains the attributes of the context node.
- child— This axis contains the children of the context node.
- descendant— This axis contains the descendants of the context node. A descendant is a child or a child of a child and so on.
- descendant-or-self— This axis contains the context node and the descendants of the context node.
- following— This axis contains all nodes that come after the context node.
- following-sibling— This axis contains all the following siblings of the context node.
- namespace— This axis contains the namespace nodes of the context node.
- parent— This axis contains the parent of the context node.
- preceding— This axis contains all nodes that come before the context node.
- preceding-sibling— This axis contains all the preceding siblings of the context node.
- self— This axis contains the context node.

Note that although the match attribute can only use the child or attribute axes in location steps (that's the major restriction on the match attribute compared to the select attribute), the select attribute can use any of the 13 axes. (The term sibling in XML refers to an item on the same level as the current item.)

For example, this template extracts the value of the <name> element by using the location path child::name:

```
<xsl:template match="state">
  <HTML>
    <BODY>
      <xsl:value-of select="child::name"/>
    </BODY>
  </HTML>
</xsl:template>
```

This is really he same as the version you've already been using because, as mentioned, you can abbreviate it by omitting the child:: part:

```
<xsl:template match="state">
  <HTML>
    <BODY>
      <xsl:value-of select="name"/>
    </BODY>
  </HTML>
</xsl:template>
```

In the location step child::name, child is the axis and name is the node test, which is described in the following section.

Node Tests

After you specify the axis you want to use in a location step, you specify the node test. A node test indicates what type of node you want to match. You can use names of nodes as node tests, or you can use the wildcard * to select element nodes. For example, the expression child::*/child::flower selects all <flower> elements that are grandchildren of the current node. Besides nodes and the wildcard character, you can also use these node tests:

- comment()— This node test selects comment nodes.
- node()— This node test selects any type of node.
- processing-instruction()— This node test selects a processing instruction node. You can specify, in the parentheses, the name of the processing instruction to select.
- text()— This node test selects a text node.

Predicates

The last part of a location step is the predicate. In a location step, the (optional) predicate narrows the search down even more. For example, the location step child::state[position() = 1] uses the predicate [position() = 1] to select not just a child <state> element but the first <state> child element.

Predicates can get pretty involved because there are all kinds of XPath expressions that you can work with in predicates. And there are various types of legal XPath expressions; here are the possible types:

- Booleans
- Node sets
- Numbers
- Strings

NOTE

There's also another type of XPath expression—result tree fragments—that you can work with in predicates. A result tree fragment is a part of an XML document that is not a complete node or complete set of nodes. There are really only two things to do with result tree fragments: You can use the string()

function or the boolean() function to turn them into strings or Booleans (that is, true/false values). Because they don't represent legal XML, they've fallen into disuse.
The following sections look at how expressions help you in XSLT.

Boolean Expressions

XPath Boolean values are true/false values, and you can use the built-in XPath logical operators to produce Boolean results.
For example, here's how to use a logical operator to match all <state> elements after the first three, using the position() function (which you'll see in the next section):

```
<xsl:template match="state[position() > 3]">
   <xsl:value-of select="."/>
</xsl:template>
```

You can also use the keywords and and or to connect Boolean expressions. The following example selects all <state> elements after the first three and before the tenth one:

```
<xsl:template match="state[position() > 3 and position() < 10]">
   <xsl:value-of select="."/>
</xsl:template>
```

In addition, you can use the not() function to reverse the logical sense of an expression. The following example selects all <state> elements except the last one, using the last() function (which you'll see in the next section):

```
<xsl:template match="state[not(position() = last())]">
   <xsl:value-of select="."/>
</xsl:template>
```

Node Sets

Besides Boolean values, XPath can also work with node sets. A node set is just a set of nodes. By collecting nodes into a set, XPath lets you work with multiple nodes at once. For example, the location step child::state/child::bird returns a node list of all <bird> elements that are children of <state> elements.
You can use various XPath functions to work with node sets. For example, the last() function picks out the last node in the node set. The following are the node set functions:

- last()— Returns the number of nodes in the node set.
- position()— Returns the position of the context node in the node set. (The first node is Node 1.)

- count(node-set)— Returns the number of nodes in node-set.
- id(ID)— Returns a node set that contains the element whose ID value matches ID.
- local-name(node-set)— Returns the name of the first node in node-set.
- namespace-uri(node-set)— Returns the URI of the namespace of the first node in node-set.
- name(node-set)— Returns the qualified name of the first node in node-set.

Some of these functions can be very useful. For example, you can number the states in the XML sample from earlier today by using the position() function, as shown in Example .

Example : An XSL Style Sheet That Uses position()

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="states">
    <HTML>
      <HEAD>
        <TITLE>
          The States
        </TITLE>
      </HEAD>
      <BODY>
        <H1>
          The States
        </H1>
        <xsl:apply-templates select="state"/>
      </BODY>
    </HTML>
  </xsl:template>

  <xsl:template match="state">
    <P>
      <xsl:value-of select="position()"/>.
      <xsl:value-of select="name"/>
    </P>
  </xsl:template>

</xsl:stylesheet>
```

Here's what an XSLT processor produces when you use this style sheet on the sample XML document:

```
<HTML>
  <HEAD>
    <META http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <TITLE>
      The States
    </TITLE>
  </HEAD>

  <BODY>
```

```
        <H1>
          The States
        </H1>
        <P>1. California</P>
        <P>2. Massachusetts</P>
        <P>3. New York</P>
    </BODY>
</HTML>
```

Note that the states are indeed numbered. Also, as with previous other examples, the whitespace and indenting here have been cleaned up. The figure below shows the result of this transformation.

When you're working on the nodes in a node set, you can use functions such as position() to target specific nodes. For example, child::state[position() = 1] selects the first <state> child of the node, where you apply this location step, and child::state[position() = last()] selects the last.

Numbers

XPath can use numbers in expressions (for example, the 1 in the expression child::state[position() = 1]).

For example, if you use <xsl:value-of select="2 + 2"/>, you get the string "4" in the output document. The following example selects all states that have at least 200 people per square mile:

```
<xsl:template match="states">
    <HTML>
        <BODY>
            <P>
                <xsl:apply-templates select="state[population div area > 200]"/>
            </P>
        </BODY>
    </HTML>
</xsl:template>
```

For example, here's how to find the total population of the states in ex_01.xml by using sum():

```
<xsl:template match="states">
    <HTML>
        <BODY>
            <P>
                The total population is:
                <xsl:value-of select="sum(child::population)"/>
            </P>
        </BODY>
    </HTML>
</xsl:template>
```

Strings

Strings in XPath are treated as Unicode characters. A number of XPath functions are specially designed to work on strings. Here they are:

- concat(string1, string2, ...)— Returns the strings joined together.
- contains(string1, string2)— Returns true if the first string contains the second one.
- format-number(number1, string2, string3)— Returns a string that holds the formatted string version of number1, using string2 as a formatting string, and string3 as an optional locale string. (You create formatting strings as you would for Java's java.text.DecimalFormat method.)
- normalize-space(string1)— Returns string1 after stripping leading and trailing whitespace and replacing multiple consecutive empty spaces with a single space.
- starts-with(string1, string2)— Returns true if the first string starts with the second string.
- string-length(string1)— Returns the number of characters in string1.
- substring(string1, offset, length)— Returns length characters from the string, starting at offset.
- substring-after(string1, string2)— Returns the part of string1 after the first occurrence of string2.
- substring-before(string1, string2)— Returns the part of string1 up to the first occurrence of string2.
- translate(string1, string2, string3)— Returns string1 with all occurrences of the characters in string2 replaced with the matching characters in string3.

Chapter 5

Integrative Coding: Design Patterns; Interfaces; Inheritance. **Miscellaneous Issues:** Adopt and Adapt vs. make; Versioning and version control

5.1 Design Patterns

Design pattern is a Lower level framework for structuring an application than architectures (Sometimes, called *micro-architecture*). It is reusable collaborations that solve sub problems within an application. Design patterns have become an essential part of ***object-oriented design and programming***. They provide elegant and maintainable solutions to commonly encountered programming problems.

Why Design Patterns?

- Design patterns support *object-oriented reuse* at a high level of abstraction
- Design patterns provide a "framework" that guides and constrains object-oriented implementation

Design Pattern Description Template

Patterns are defined using a **description template**. The **Gang of Four (**GoF) developed a description template that is used to define their patterns. This is critical because the information has to be conveyed to peer developers in order for them to be able to evaluate, select and utilize patterns.

Section	Description
Pattern Name and Classification	The name of the pattern, and its classification (Creational, Structural, or Behavioral).
Intent	A short statement about what the pattern does.
Also Known As	Alternate well known names for the pattern.
Motivation	An illustrative design problem that shows how the pattern can solve the problem.
Applicability	Situations where the pattern can be used.
Structure	A graphical (UML) representation showing the classes in the pattern.
Participants	The classes that participate in the pattern and their responsibilities.
Collaborations	How the participants collaborate.
Consequences	Benefits and trade-offs of using the pattern.

Implementation	Hints, pitfalls, and techniques that can be used to help implement the pattern.
Sample Code	Code illustrations of using the pattern.
Known Uses	Examples of the pattern used in real systems.
Related Patterns	Other patterns closely related to the current one.

The primary reference is the **Gang of Four *Design Patterns* book**. The Gang of Four (GoF) *Design Patterns* book describes **twenty-three patterns arranged into three groups**. The groups help classify how the patterns are used.

1. ***Creational patterns*** are used to help make a system independent of how its objects are created, composed and represented.
2. ***Structural patterns*** are concerned with how classes and objects are organized and composed to build larger structures.
3. ***Behavioral patterns*** are used to deal with assignment of responsibilities to objects and communication between objects.

Creational Patterns

1) Abstract Factory

An *Abstract Factory* is a class that is used to create instances of other objects that are related to or depend on each other. An example of this is creating GUI components for different GUI toolkits. The Abstract Factory will let the application see a unified component, while it creates the appropriate concrete object for a given look-and-feel.

2) Builder

The *Builder* pattern is used by a Director class to construct different complex objects based on a specific requirement. An example is creating a text converter object that can convert text to different formats depending on how the converter is built.

3) Factory Method

A *Factory Method* provides a common interface for creating subclasses. The factory method is defined in the top level class definition, but instantiation of the subclasses determine which specific instance of the factory method is used.

4) Prototype

The *Prototype* pattern can reduce the number of different classes used by creating new object instances based on a copy or clone of an original prototype instance. The copies can then be modified to carry out their different responsibilities.

5) ***Singleton***

Often there will be situations where there needs to be one and only one instance of a class. Rather than relying on the programmer to create only one instance of a class, the *Singleton* pattern will create only one instance of a class, and makes that instance accessible by other objects.

Structural Patterns

6) ***Adapter***

The *Adapter* pattern is used to adapt an interface of one class so that it can be used by a different class that originally could not use it. The adapter is sometimes called a wrapper. For example, adapters or wrappers are often written for existing libraries so that a new application can use the library.

7) ***Bridge***

A *Bridge* is used to decouple an abstraction from its implementation. The Bridge pattern is often used to build drivers, such as a printer driver, which connect an application program with a real printer. It is possible to vary the abstraction and the implementation independently.

8) ***Composite***

The *Composite* pattern is used to compose whole-part hierarchies into a tree structure that will let the client treat either the individually objects, or compositions of those objects uniformly. An example is a graphical object, which can be made up of individual graphics (like a line or a square), or other graphical objects (a picture in a picture).

9) ***Decorator***

The goal of the *Decorator* pattern is to add more responsibilities to an object dynamically, and avoid subclassing. An example is adding scrolling to a text view. The scrolling object surrounds the text view, handles the mechanics of the scroll bar, and then tells the text view to display itself appropriately.

10) ***Facade***

The *Facade* pattern provides a higher level unified interface to a set of objects in a subsystem. The extra layer provides a simpler interface to the subsystem, and helps to avoid coupling between classes.

11) ***Flyweight***

The *Flyweight* pattern is used to efficiently use a large number of small objects with sharing. An example is sharing representation of individual characters in a document formatting program.

12) Proxy

The *Proxy* pattern is similar to the Adapter in that it provides an interface layer between objects. While an Adapter will change the interface, a Proxy doesn't change the interface, but instead will control the access to one object from another.

Behavioral Patterns

13) Chain of Responsibility

The *Chain of Responsibility* pattern allows a sender to issue a request to a series or chain of objects and allowing each object a chance to handle the request. The receiving objects pass the request along until some object handles the request. An example is a context sensitive help system that will pass a request for help along a chain until the appropriate object can provide the help.

14) Command

The *Command* pattern provides a way to encapsulate a command request as an object without the object that issues the command needing to know what the response will be. An example is a GUI menu system that will issue a command in response to a menu selection. Any menu command can issue a request to a Command object in a uniform way without having to know how the object will handle the command.

15) Interpreter

The *Interpreter* pattern recognizes that sometimes it is better to define a language with a grammar, and to provide an interpreter for that language. An object that can recognize and respond to a regular expression search pattern is an example that can use the Interpreter pattern.

16) Iterator

The *Iterator* pattern is used to provide a means to access all the elements of some collection of objects sequentially without exposing the collection's internal representation. The Iterator is so common and useful that Java provides iterators for its collection objects as a standard feature.

17) Mediator

The *Mediator* pattern is used to define an object that knows how to use several other objects, and to provide a means for those objects to refer to each other using the Mediator instead of directly. This increases encapsulation and decreases coupling.

18) Memento

The *Memento* pattern is used to capture and save the current state of an object without violating its encapsulation. For example, an editor program would use the Memento pattern to save the state of whatever was being edited so that it could be restored via an undo command.

19) Observer

The *Observer* pattern is used when any numbers of objects (the Observers) need to be notified automatically whenever another object (the Observable) changes its state.

20) State

The *State* pattern is used to allow an object to change its behavior depending on how its internal state is changed. An example is a network monitoring object that will change its behavior depending on whether the network connection is open or closed.

21) Strategy

The *Strategy* pattern is used to define a family of interchangeable algorithms. The client will be able to use the object that implements the Strategy without necessarily knowing which strategy is used, or how it differs from other strategies. An example might be a Strategy that implements different sort algorithms, with each algorithm appropriate for different kinds of data.

22) Template Method

The *Template Method* is used to define an operation as a superclass whose implementation will be deferred to subclasses. This lets the subclasses redefine an operation without affecting how the method is used.

23) Visitor

The *Visitor* pattern is used to perform an operation on some structure of objects. The Visitor allows new operations to be defined without changing any of the elements that are part of the structure. An example would be visiting each leaf of a tree of objects to perform some operation.

Wmvc and MovieCat use several patterns, including Observer, Command, Singleton, and Iterator

5.2 Interfaces

Application Programming Interfaces (APIs) are sets of requirements that govern how one application can talk to another. APIs do all this by "exposing" some of a program's internal functions to the outside world in a limited fashion. That makes it possible for applications to share data and take actions on one another's behalf without requiring developers to share all of their software's code. APIs clearly define exactly how a program will interact with the rest of the software world—saving time, resources and potentially nasty legal entanglements along the way.

Example 1:

Whenever you use a desktop or laptop, APIs are what make it possible to move information between programs—for instance, by cutting and pasting a snippet of a LibreOffice document into an Excel spreadsheet. **System-level APIs** makes it possible for applications like LibreOffice to run on top of an OS like Windows in the first place.

Example 2:

On the Web, APIs make it possible for big services like Google Maps or Facebook to let other apps "piggyback" on their offerings. Think about the way Yelp, for instance, displays nearby restaurants on a Google Map in its app, or the way some video games now let players chat, post high scores and invite friends to play via Face book, right there in the middle of a game.

How APIs Work

These days, APIs are especially important because they dictate how developers can create new apps that tap into big Web services—social networks like Facebook or Pinterest, for instance, or utilities like Google Maps or Dropbox. The developer of a game app, for instance, can use the Dropbox API to let users store their saved games in the Dropbox cloud instead of working out some other cloud-storage option from scratch.

In one sense, then, APIs are great time savers. They also offer user convenience in many cases; Facebook users undoubtedly appreciate the ability to sign into many apps and Web sites using their Facebook ID—a feature that relies upon Facebook APIs to work.

5.3 Inheritance

Inheritance means that you derive a new class based on an existing class, with modifications or extensions.

In OOP, we often organize classes **in *hierarchy* to *avoid duplication and reduce redundancy***. The classes in the lower hierarchy inherit all the variables (static attributes) and methods (dynamic behaviors) from the higher hierarchies. A class in the lower hierarchy is called a ***subclass* (or *derived*, *child*, *extended class*)**. A class in the upper hierarchy is called a ***superclass* (or *base*, *parent class*).** By pulling out all the common variables and methods into the superclasses, and leave the specialized variables and methods in the subclasses, *redundancy* can be greatly reduced or eliminated as these common variables and methods do not need to be repeated in all the subclasses.

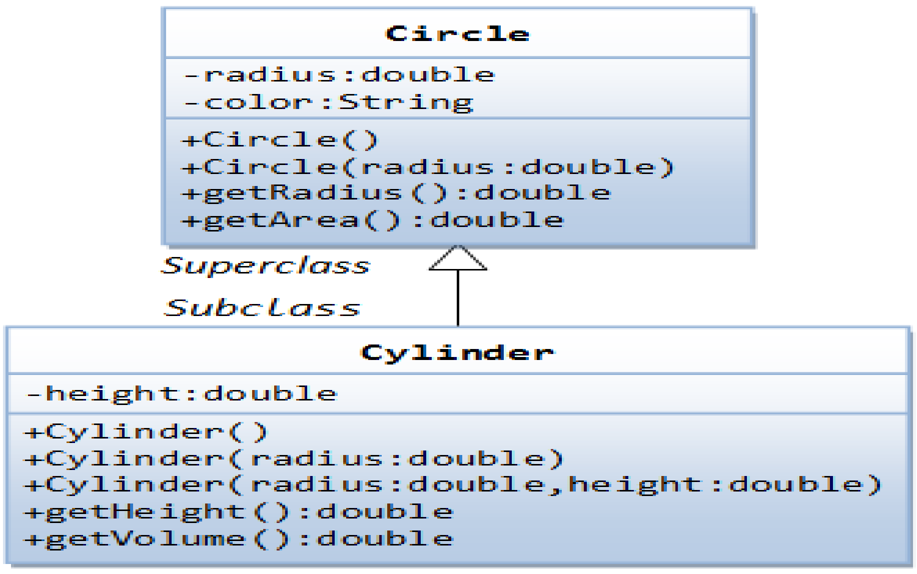
A subclass inherits all the variables and methods from its superclasses, including its immediate parent as well as all the ancestors. It is important to note that a subclass is not a "subset" of a superclass. In contrast, subclass is a "superset" of a superclass. It is because a subclass inherits all the variables and methods of the superclass; in addition, it extends the superclass by providing more variables and methods.

In Java, you define a subclass using the keyword "extends", e.g., class Goalkeeper **extends** SoccerPlayer {......}

class MyApplet **extends** java.applet.Applet {.....}

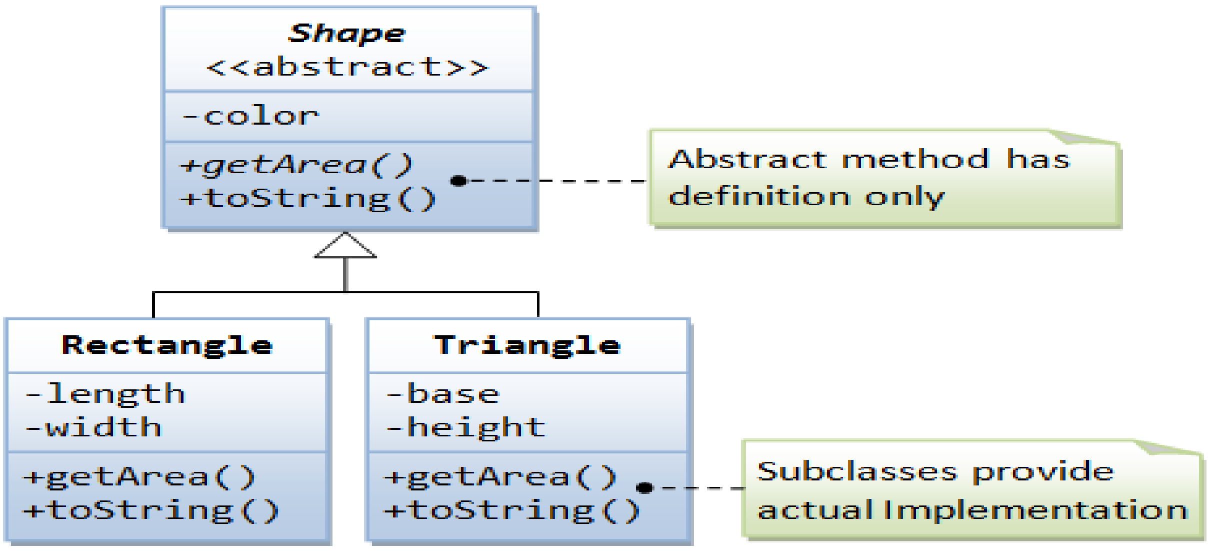
class Cylinder **extends** Circle {......}

An Example on Inheritance



The abstract Class and Inheritance

An abstract method is a method with only signature (i.e., the method name, the list of arguments and the return type) without implementation (i.e., the method's body). You use the keyword abstract to declare an abstract method



A class containing one or more abstract methods is called an abstract class. An abstract class must be declared with a class-modifier abstract. Let rewrite our Shape class as an abstract class, containing an abstract method getArea() as follows:

Shape.java

```
abstract public class Shape {
  // Private member variable
  private String color;
    // Constructor
  public Shape (String color) {
    this.color = color;
  }
    @Override
  public String toString() {
    return "Shape of color=\"" + color + "\"";
  }
    // All Shape subclasses must implement a method called getArea()
  abstract public double getArea();
}
```

An abstract class is *incomplete* in its definition, since the implementation of its abstract methods is missing. Therefore, an abstract class *cannot be instantiated*. In other words, you cannot create instances from an abstract class (otherwise, you will have an incomplete instance with missing method's body).

To use an abstract class, you have to derive a subclass from the abstract class. In the derived subclass, ***you have to override the abstract methods and provide implementation to all the abstract methods***. The subclass derived is now complete, and can be instantiated. (If a subclass does not provide implementation to all the abstract methods of the superclass, the subclass remains abstract.) We can then derive subclasses, such as Triangle and Rectangle, from the superclass Shape. Rectangle.java

```
// Define Rectangle, subclass of Shape
public class Rectangle extends Shape
 {
              // Private member variables
              private int length;
              private int width;

              // Constructor
              public Rectangle(String color, int length, int width)
            {
                super(color);
                this.length = length;
                this.width = width;
              }

              @Override
              public double getArea()
             {
                return length*width;
```

```
        }
}
```

Triangle.java

```
// Define Triangle, subclass of Shape
public class Triangle extends Shape
{
        // Private member variables
        private int base;
        private int height;

        // Constructor
        public Triangle(String color, int base, int height)
       {
          super(color);
          this.base = base;
          this.height = height;
        }

        @Override
        public double getArea()
       {
          return 0.5*base*height;
        }
}
```

The subclasses override the getArea() method inherited from the superclass, and provide the proper implementations for getArea().

For example,

```
public class TestShape
 {
  public static void main(String[] args)
 {
    Shape s1 = new Rectangle("red", 4, 5);
    System.out.println(s1);
    System.out.println("Area is " + s1.getArea());
    Shape s2 = new Triangle("blue", 4, 5);
    System.out.println(s2);
    System.out.println("Area is " + s2.getArea());

    // Cannot create instance of an abstract class
    Shape s3 = new Shape("green");   // Compilation Error!!
  }
}
```

In summary, an abstract class provides *a* ***template for further development***. For example, in the abstract class Shape, you can define abstract methods such as getArea().The subclasses could provide the proper implementations. Furthermore, your application can be extended easily to accommodate new shapes (such as Circle or Square) by deriving more subclasses.

Notes:

An abstract method cannot be declared final, as final method cannot be overridden. An abstract method, on the other hand, must be overridden in a descendent before it can be used.

An abstract method cannot be private (which generates a compilation error). This is because private method is not visible to the subclass and thus cannot be overridden.

5.3 Miscellaneous Issues: Adopt and Adapt vs. make; Versioning and version control

A version control system serves the following purposes, among others.

- Version control enables multiple people to simultaneously work on a single project. Each person edits his or her own copy of the files and chooses when to share those changes with the rest of the team. Thus, temporary or partial edits by one person do not interfere with another person's work.
- Version control also enables one person you to use multiple computers to work on a project, so it is valuable even if you are working by yourself.
- Version control integrates work done simultaneously by different team members. In most cases, edits to different files or even the same file can be combined without losing any work. In rare cases, when two people make *conflicting edits* to the same line of a file, then the version control system requests human assistance in deciding what to do.
- Version control gives access to historical versions of your project. This is insurance against computer crashes or data lossage. If you make a mistake, you can roll back to a previous version. You can reproduce and understand a bug report on a past version of your software. You can also undo specific edits without losing all the work that was done in the meanwhile. For any part of a file, you can determine when, why, and by whom it was ever edited.

Repositories and working copies

Version control uses a *repository* (a database of changes) and a *working copy* where you do your work.

Your *working copy* (sometimes called a *checkout*) is your personal copy of all the files in the project. You make arbitrary edits to this copy, without affecting your teammates. When you are happy with your edits, you commit your changes to a *repository*.

A repository is a database of all the edits to, and/or historical versions (snapshots) of, your project. It is possible for the repository to contain edits that have not yet been applied to your working copy. You can update your working copy to incorporate any new edits or versions that have been added to the repository since the last time you updated.

In the simplest case, the database contains a linear history: each change is made after the previous one. Another possibility is that different users made edits simultaneously (this is sometimes called "branching"). In that case, the version history splits and then merges again.

Distributed and centralized version control

There are two general varieties of version control: *centralized* and *distributed*. Distributed version control is more modern, runs faster, is less prone to errors, has more features, and is somewhat more complex to understand. You will need to decide whether the extra complexity is worthwhile for you.

Some popular version control systems are Mercurial (distributed), Git (distributed), and Subversion (centralized). The main difference between centralized and distributed version control is the number of repositories. In centralized version control, there is just one repository, and in distributed version control, there are multiple repositories.