Web Application Development

© Luca Dan Serbanati

2008

Introduction to Distributed Computing

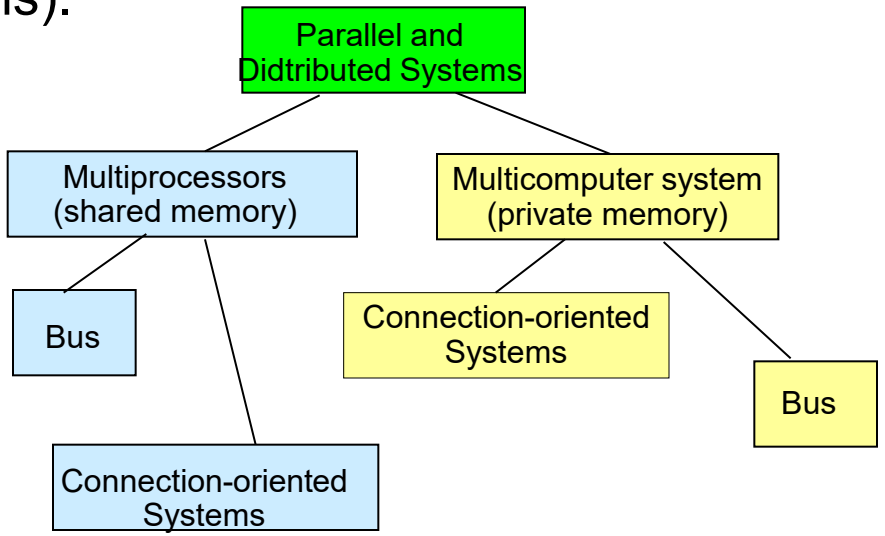
Cooperative Processing

Cooperative processing = Complex function processing by means of n cooperating sub-functions.

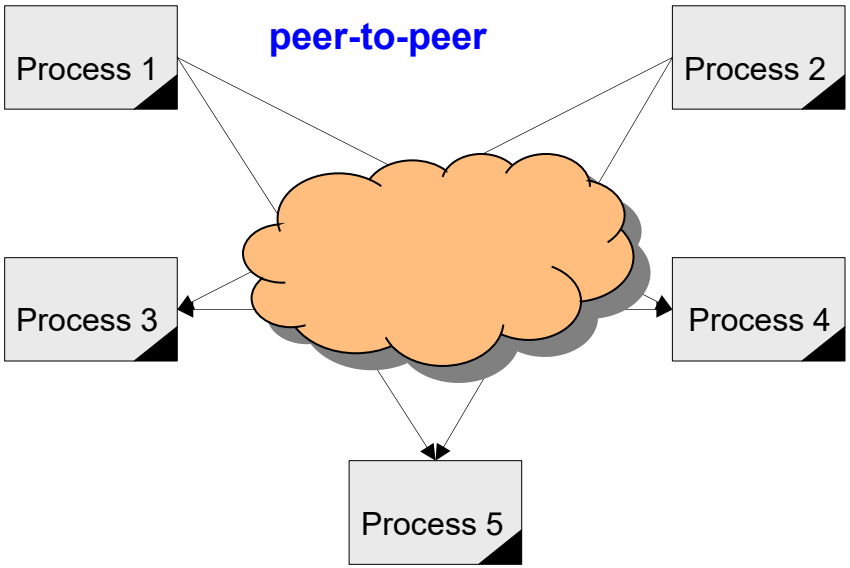
Solutions for cooperative processing

1. Sequential
   - A process for each function. The complex function process uses sequentially the processes of sub-functions.
2. Concurrent
   - A process for each function. Two possibilities:
     - Concurrency in the same multiprocessor system (parallel systems)
     - Execution of several processes in systems interconnected in a transport net (distributed systems).

Distributed system = Collection of independent computers that appear to the users of the system as a single computer.

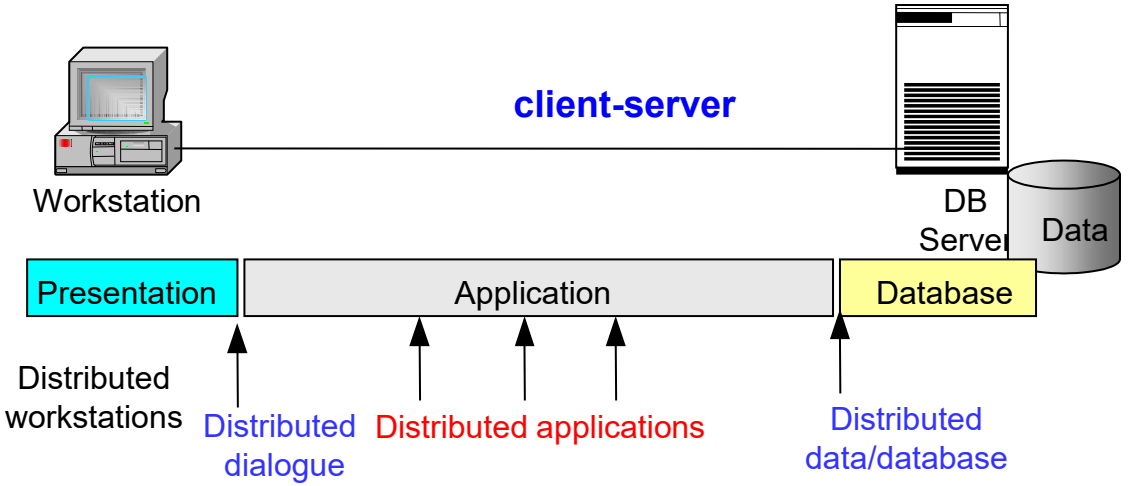


Process Interaction Models



- P2P - Peer-to-peer (tight connection, low transparency, mutual changes)

- C/S - Client-Server (autonomous partners, task separation, decoupled changes)



Glossary

- RESOURCE something that can be usefully shared in a networked computer system e.g. data, hardware, application,...
- SERVICE = part of a computer system which manages a collection of related resources and presents their functionality to users and applications; access is restricted to a well-defined set of operations.
- SERVER = a program or process on a networked computer which provides a service.
- CLIENT = a program or process generating a request to a server (invocation); requests are sent in messages from the client to the server and replies are sent in messages from the server to the client.
- Note that client and server apply only to the roles played within a single invocation.
- PEER PROCESSES = processes that cooperate and communicate in a symmetrical manner to perform a task.

Distributed Computing

- Distributed computing = Computations that own the resources in more than address space (an address space).
- Distributed application = Application that consists from more software parts that communicate between them cooperating for the realization of a common objective. Typically, these are distributed between more computers.
- Distribution in distributed computing can mean:
  - distributed user,
  - distributed information
  - distributed devices.
- Examples of applications that are distributed in natural way:
  - control systems of the traffic
  - control systems of the industrial processes.
- Advantages of the distribution:
  - Better control on the use and coordination of the resources of the system;
  - Smaller cost of the development of the distributed applications;
  - Possibility to share the total load of the system between more address spaces;
  - Independence of platform.

Transparency in distributed systems

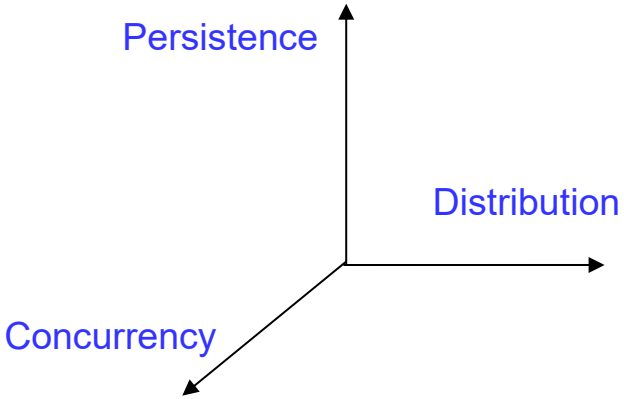
Distributed system = a collection of computers connected in a network with following characteristics:

- All the nodes are loosely coupled;
- Every node is an autonomous computer;
- The system can survive to the crash of the nodes or the net;
- The nodes can logically at the same time execute separated computations from the other;
- The system is asynchronous.

A true distributed system is transparent to:

- Access transparency = Accesses are uniform either to local or remote resources.
- Location transparency = The users cannot tell where are located.
- Migration transparency = resources can move at will without changing their names.
- Replication transparency = The users cannot tell how many copies exist.
- Concurrency transparency = Multiple users can share resources automatically.
- Parallelism transparency = Activities can happen in parallel without users knowing.
- Performance transparency.
- Failures transparency.

Dimensions in Distributed Computing



- Concurrency = appearing or real parallelism of the control flows.
- Distribution = Objects of the application are separated from each other by physical and/or logical distances.
- Persistence = Data and objects are stored in non- volatile memory.

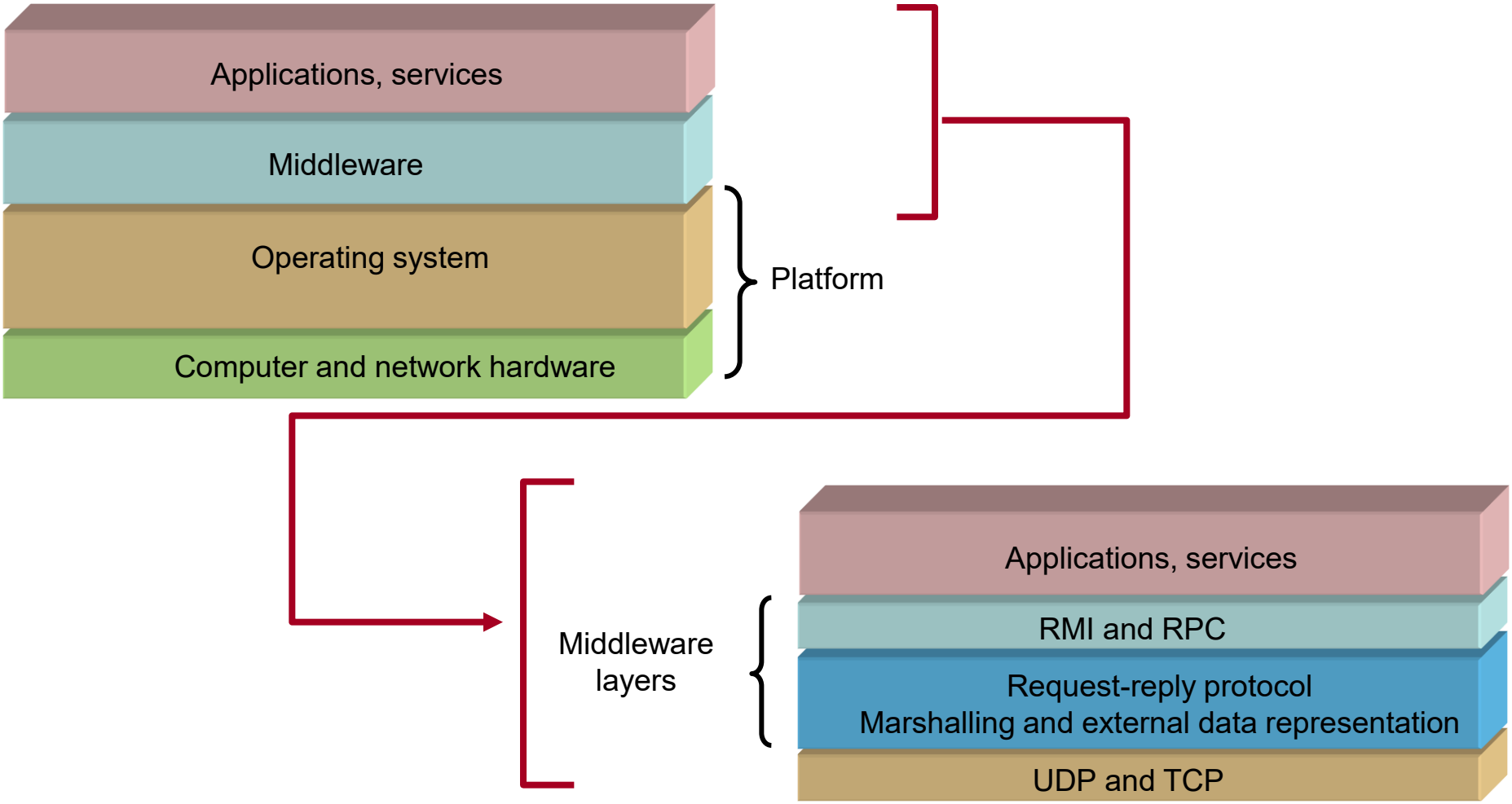
Fundamental for the realization of a distributed system is the communication infrastructure.

Cohabitation of the three conceptual dimensions that are often in conflict with each other and with the communication infrastructure represents the central problem of planning of the distributed systems.

Challanges for SD development

- Problems to face in the planning of the distributed systems and techniques or technologies you need to use in order to solve them:
  - Heterogeneity: standard and protocols, middleware, virtual machine;
  - Openness: publication of services, notification of the interfaces, independence from the vendors;
  - Security: firewalls, cryptography;
  - Scalability: replication, caching, multiple server;
  - Failure management : failure tolerance, recover/roll-back, redundancy;
  - Transparency: middleware, location transparent naming, anonymity;
  - Concurrency: control of concurrency to verify the consistency of the data.
- Exceeded the difficulties, the advantages of the distributed systems :
  - Parallelism. More computer can work simultaneously to the resolution of the same problem.
  - Scalability. There are no limits for the maximum dimensions of the system.
  - Extensibility. We can add without problems a new computer or new resources.
  - Availability. In the distributed systems there are redundant components.
  - Resource sharing. More devices deployed in the network can use the same expensive resources.
  - Fault tolerance. The robustness of the system can be increased through the duplication.

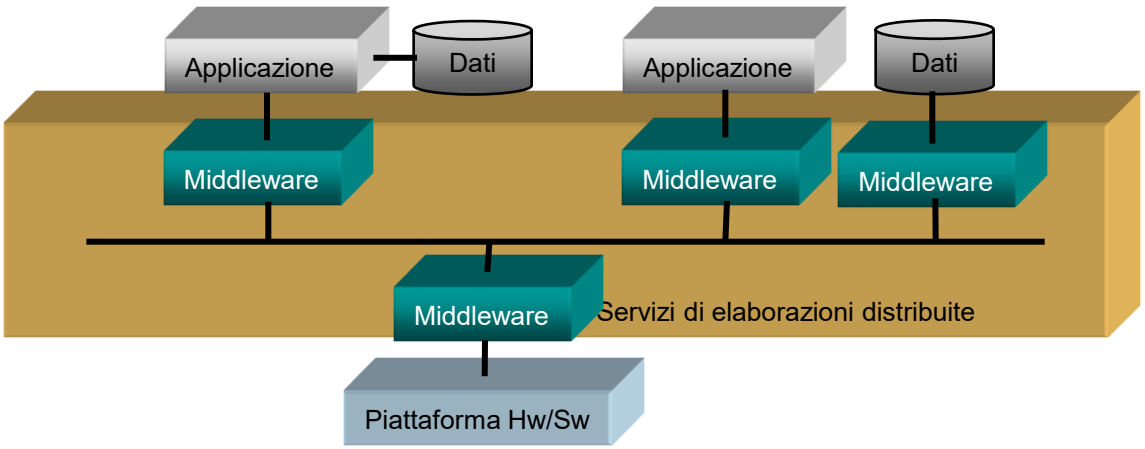
Layered Architecture of Distributed Systems



Services for distributed applications

- Beyond to the essential services of communication, a distributed application has the necessity to use some services that are always required in these applications.
- Needed services for Distributed Computing:
  - Naming Service
  - Security Service
  - Transaction Service
  - Persistence Service
  - Event Service
  - Queuing Service
- These services are implemented by from a software called middleware.
- Middleware is a general term for any programming that serves to "glue together" or mediate between two separate and often already existing programs. It is a software that functions as a conversion or translation layer. It is also a consolidator and integrator.

Traditional Middleware

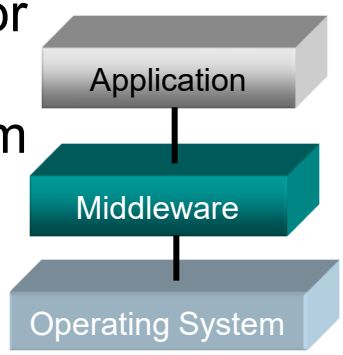


Custom-programmed middleware solutions have been developed for decades to enable one application to communicate with another that either runs on a different platform or comes from a different vendor or both.

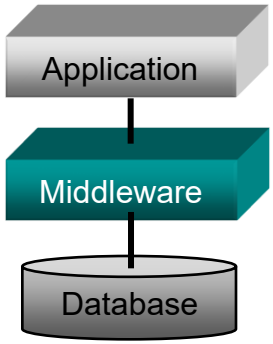
Tasks of middleware

1. it realizes common services for several applications.
2. it calls functions of the platform operating system in order to execute the common services:
   - communications in net,
   - file management,
   - task scheduling.

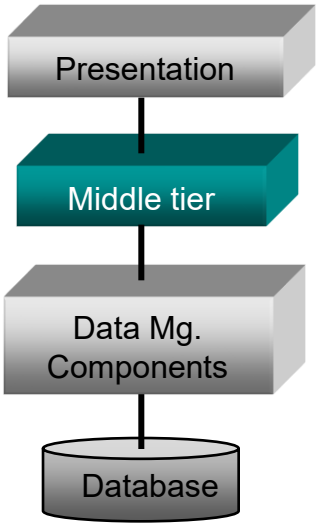
Exemples



Programming interface for a real-time application



Access interface to a database



Container for the business logic in a three tiers application

Distributed Objects

Distributed object = a server object of which the services can be called from a remote client. A distributed object can be called from:

- a client in the same process with the object (in-process),
- a client situated in an other process (out-process) on the same platform
- a client situated in an other place in net.

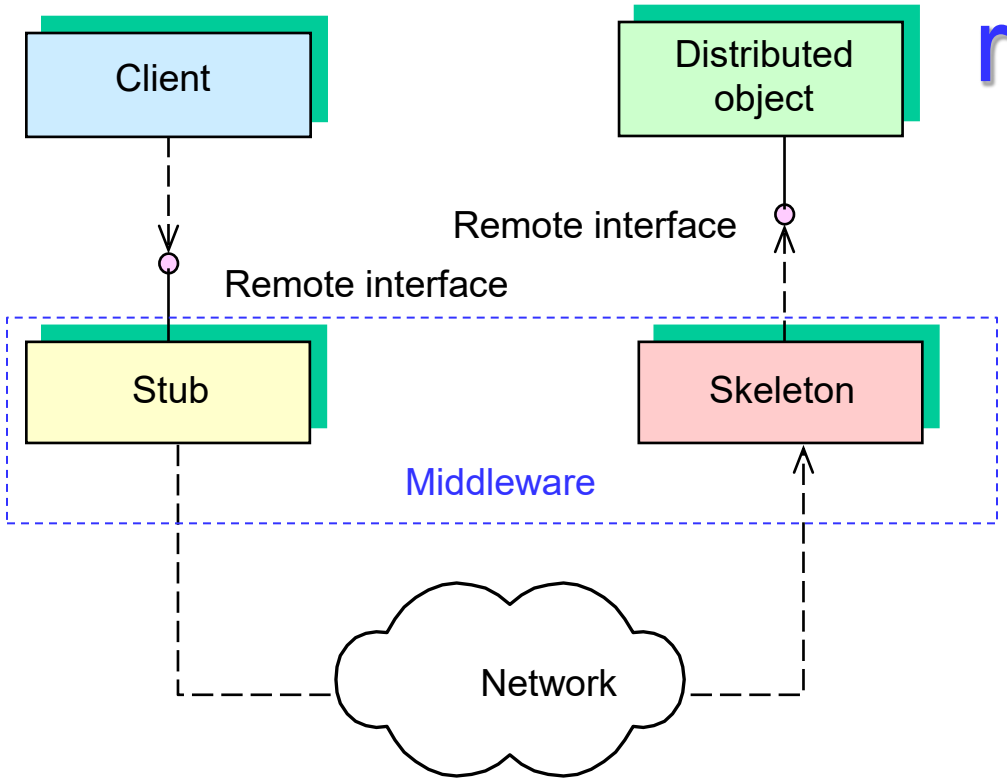
Scenario of the remote call is always the same one:

1. The client calls one stub, that is an object-proxy of the remote object placed in the client side client. The stub hides the communication in net for the client.
2. The stub calls through the net one skeleton, that is an object-proxy of the object distributed placed in the server-side. The skeleton hides the communication in net for the distributed object.
3. The skeleton delegates the call to the distributed object. The distributed object carries out its computation and returns the control to the skeleton. The skeleton sends the result to the stub and this one returns it to the client.

The fundamental aspect for remote call implementation:

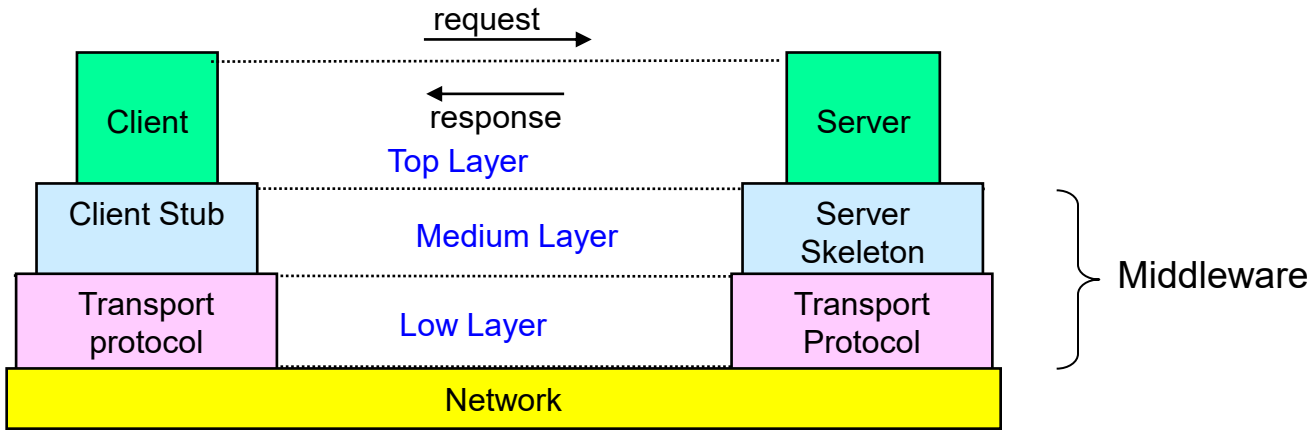
both the stub and the distributed object rigorously implement the same interface (called remote interface).

Communication with remote objects



Technologies:

- CORBA
- DOT NET
- RMI-IIOP
- web services



More about middleware

Characteristics of middleware

- It provides us all the critical services necessary us in the management of the execution of the applications in a client/server environment:
- It shields the applications from the complexity of the distributed environment.
- It is to the base of the distributing computing

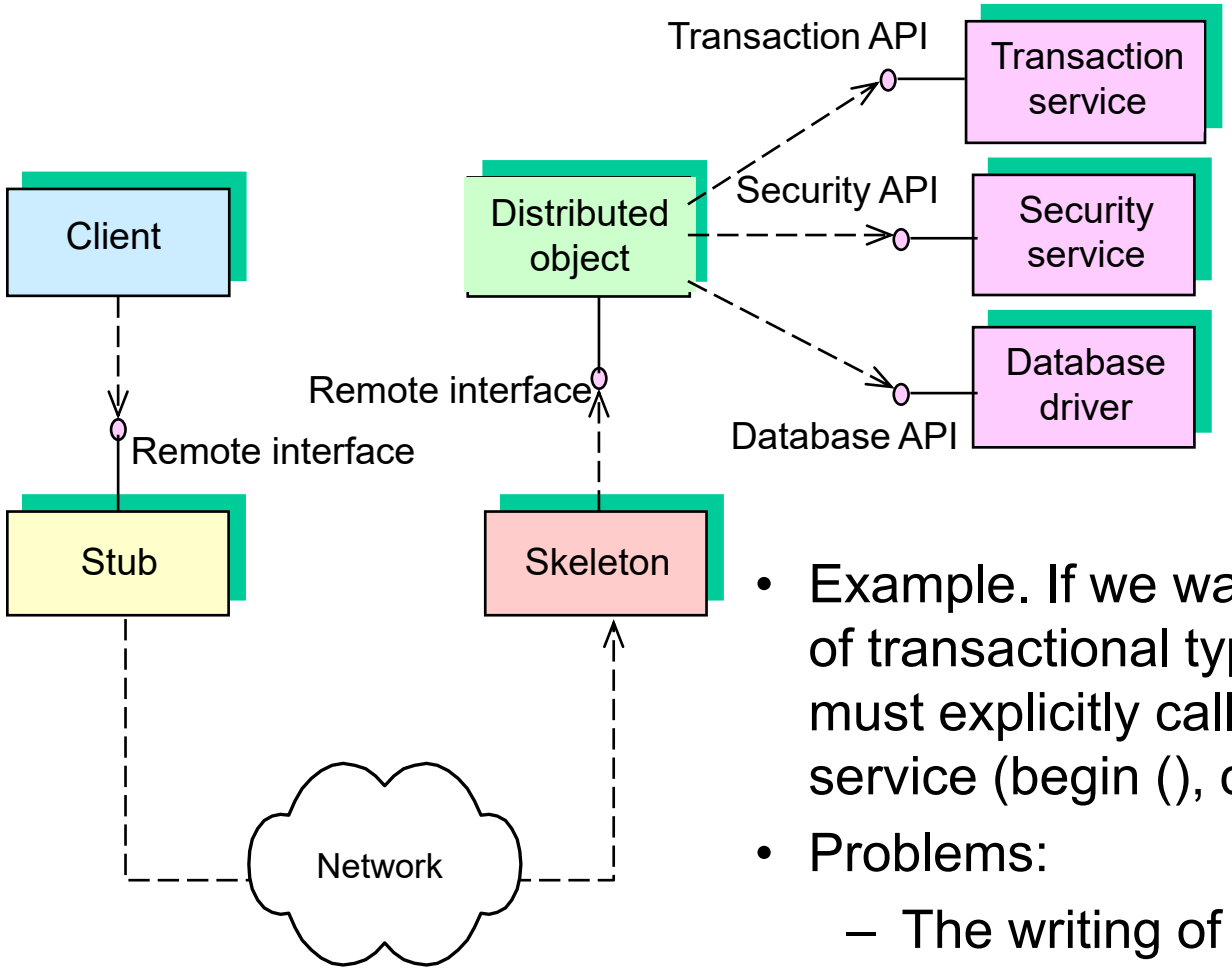
Advantages

- More applications can re-use the services.
- It is possible to modify the services without changes in the system.
- The business logic can use in transparent way data from several sources.
- Changes in the infrastructure (i.e. database) can be realized without changes in the client.
- The services can be distributed in several locations in order to increase the overall flexibility and facilitate the control of the performances.

Challenges

- Middleware systems rely on interception and indirection mechanisms, which induce performance penalties.
- As applications become more and more interconnected and interdependent, the number of objects, users and devices tends to increase. This poses the problem of the scalability of the communication and object management algorithms, and increases the complexity of administration.

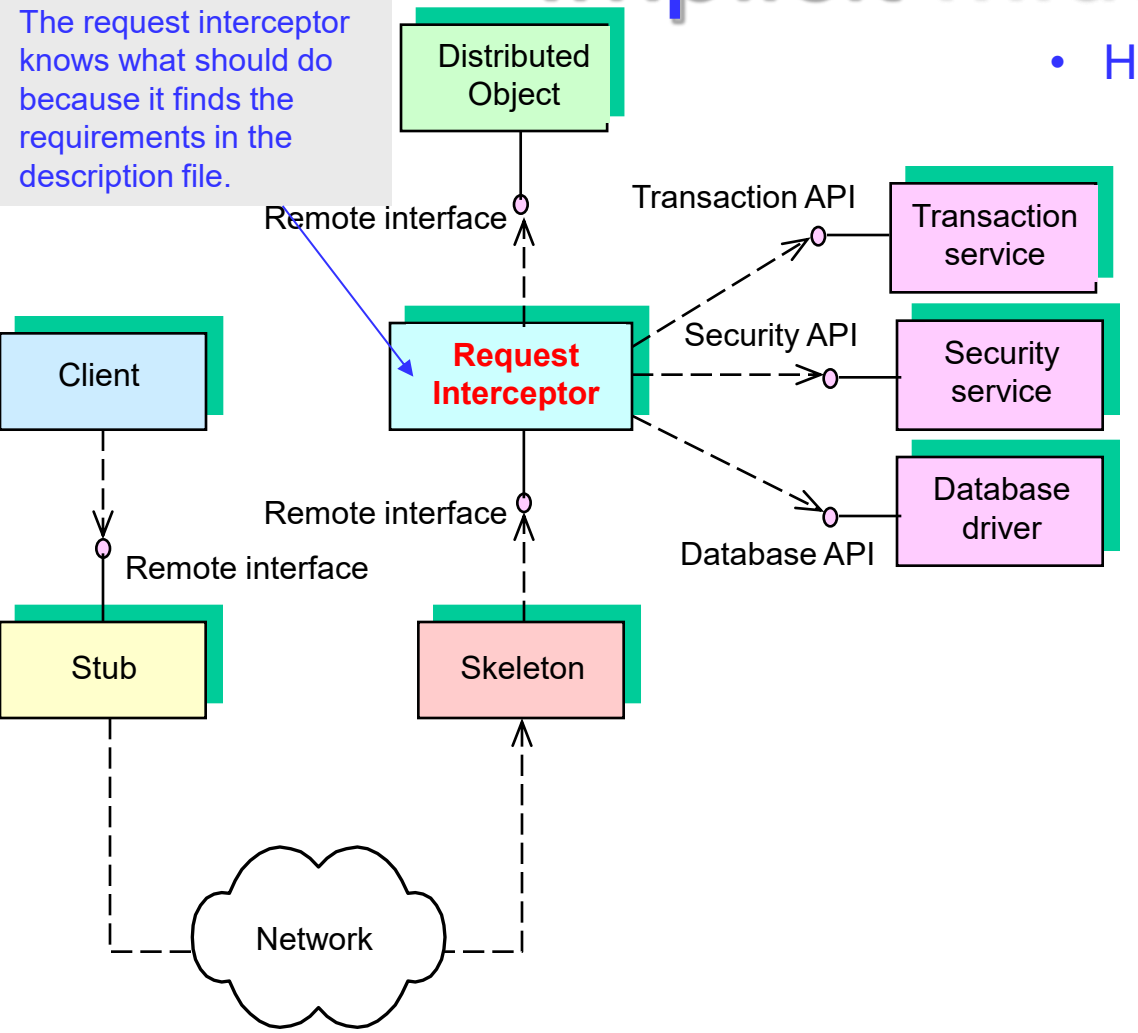
Explicit Middleware



- Example. If we want to have an application of transactional type, in our programs we must explicitly call the API of the transaction service (begin (), commit (), rollback ()).
- Problems:
  - The writing of the code is more difficult.
  - The maintenance is difficult.
  - It is difficult to improve the client-side.

Implicit Middleware

The request interceptor knows what should do because it finds the requirements in the description file.



- How to use the implicit middleware :
  1. Write the code of the distributed object containing the business logic only, without calls of specialized service APIs.
  2. Specify in the description file the middleware services that the object have to use.
  3. Carry out a middleware supplied tool that uses the description file in order to generate a “request interceptor”.
  4. Carry out the application: The request interceptor intercepts the requests coming from the client, asks to be executed all necessary middleware services specified in the description file and finally delegates the call to the distributed object.

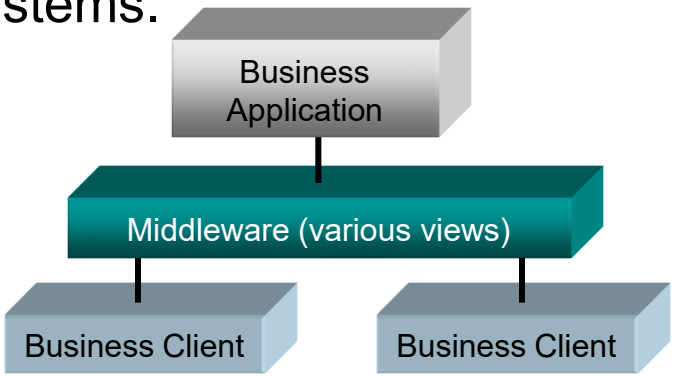
Advantages:

Easy to write,
Easy maintenance,
Easy modifiable if the requirements change (only the description file is changed).

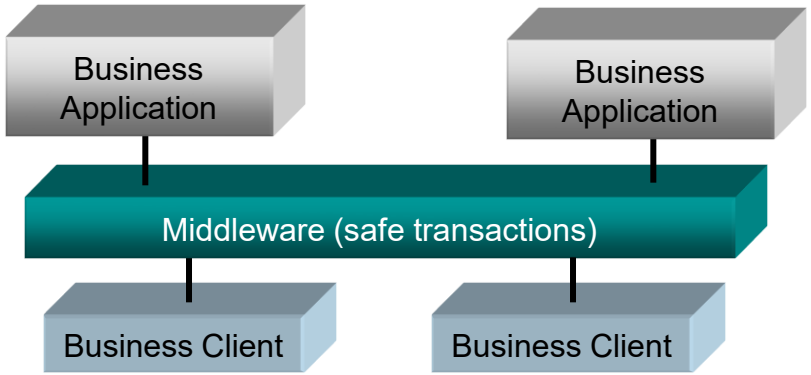
High Level Middleware

Internet, the private nets, e-commerce and the systems of resource planning added new characteristics to traditional middleware:

- middleware transforms the nets in connections between users and applications;
- middleware becomes an instrument of integration between products and systems.



Comunications



E-commerce

New middleware types :

- EAI - Enterprise Application Integration:
  - integration to the data level between users and heterogenous systems,
  - integration to the business model between users and heterogenous systems.
- Application Servers supply business logic for distributed applications distributed on heterogenous platforms.

Software Components

- Component = a piece of software that implements a set of well-defined interfaces. The components encapsulate a complex logic that exceeds that one of an object, but not sufficient for being a complete application. Once defined a component, it becomes an independent piece de software that can be distributed and used in other applications.
- A component is developed for a certain objective/function and not for a specific application!
- UML:
  - A component is a self contained unit that encapsulates the state and behavior of a number of classifiers.
  - A component specifies a formal contract of the services that it provides to its clients and those that it requires from other components or services in the system in terms of its provided and required interfaces.
  - A component is a substitutable unit that can be replaced at design time or run-time by a component that offers equivalent functionality based on compatibility of its interfaces. As long as the environment obeys the constraints expressed by the provided and required interfaces of a component, it will be able to interact with this environment. Similarly, a system can be extended by adding new component types that add new functionality.

Exemple of components

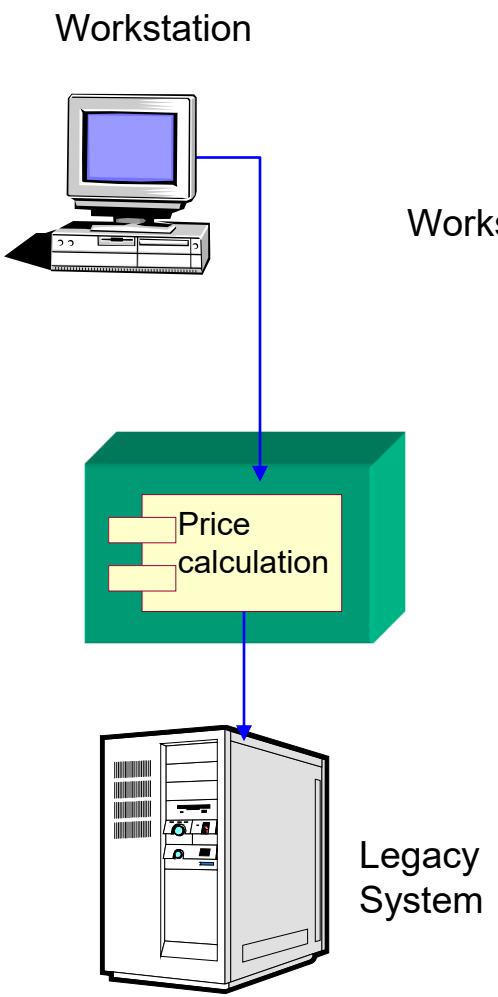
Price calculation. A component that calculates the total price of an order after it calculated the prices of product items. The component knows the rules how the prices of the products are calculated:

- the base price
- the reduction in price according to the number of products
- the reduction in price to the package of products
- the reduction in price of the customer
- the local or regional reduction in price
- adding costs as TVA etc.

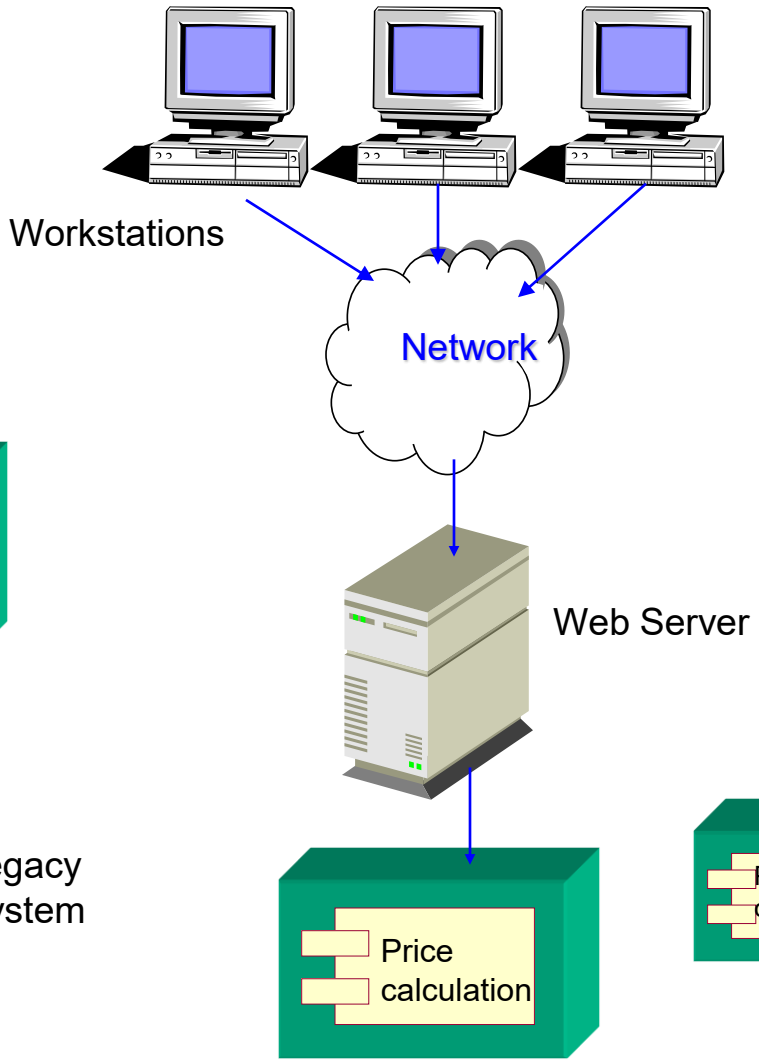
A such component can be used in many environments and situations: post offices where the prices of the packages must be calculated, companies that assemble computers where the total price of a computer is calculated according its configuration, chains of stores etc.

Scenarios

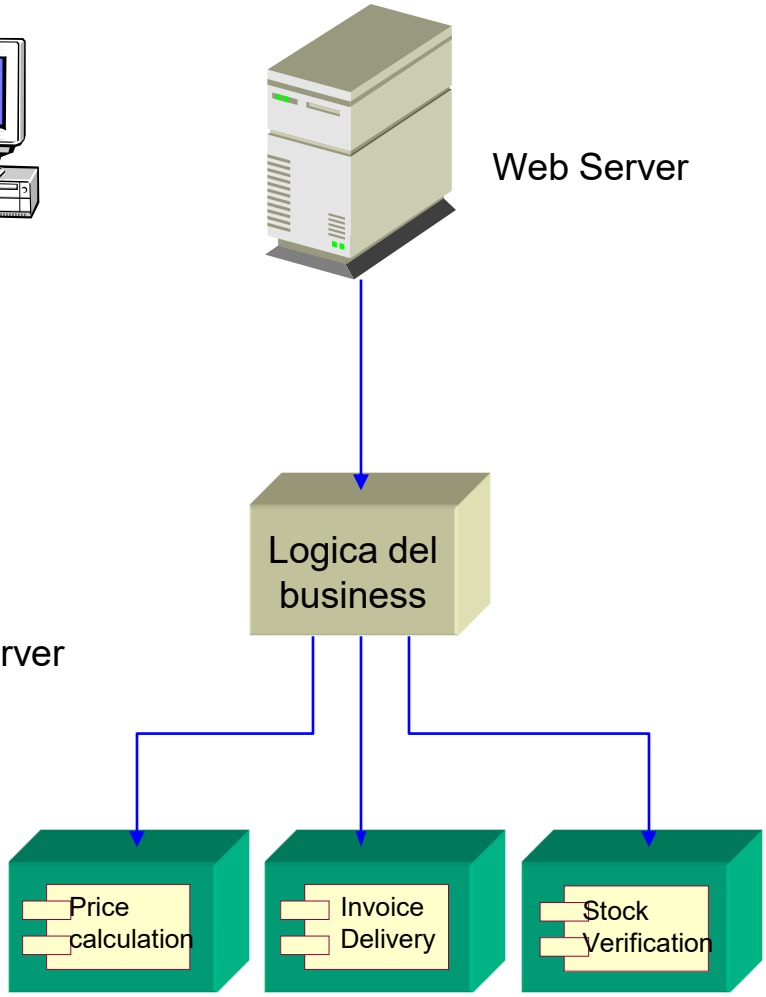
Scenario 1: Post Office



Scenario 2: Computer Provider

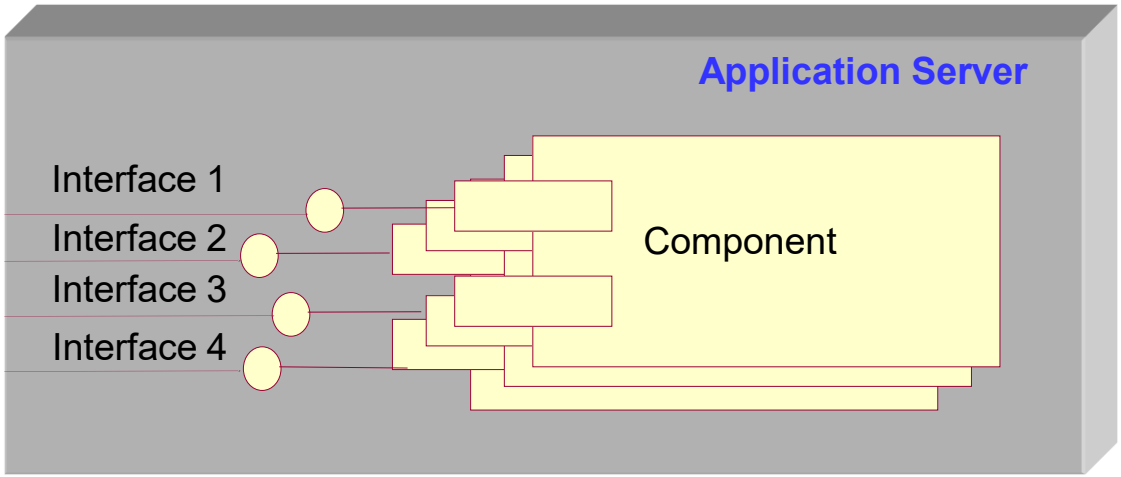


Scenario 3: Chain of stores



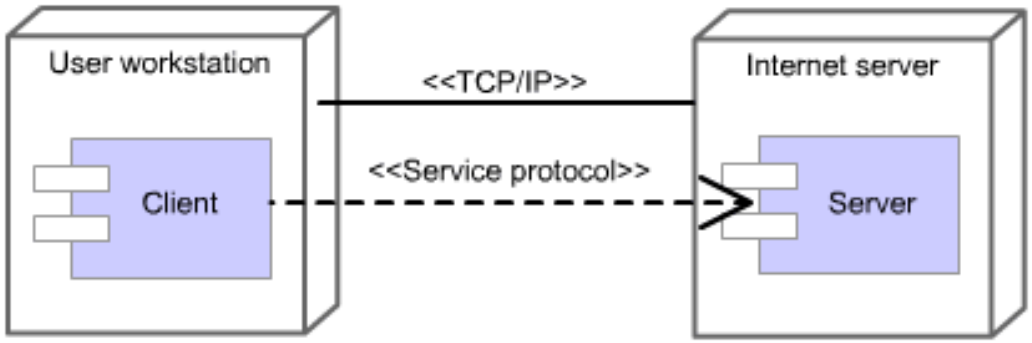
Component-Orientation

- Component-Oriented Model defines a set of contracts between the component and the system that will host the component. The contracts stipulate how the components must be developed and packed.
- Component-Oriented Architecture = A set of standard interfaces between the application server and components. The standard imposes that the same component can be used with whichever application server.



- Application Server = Middleware specialized in software components management. It supplies services to components according to the needs of the application that uses them.
- An application server is designed to manage many of the aspects of a system business. It has at least two members: a server for enterprise components, a HTTP/HTTPS server for web components.

Communication paradigms



Comunicazioni per applicazioni distribuite

Synchronous communications (client-server)

- Byte streams = two sockets with associated queues defined in the two communicating machines;
- Access to a database distributed or through a Transaction Processing (TP) Monitor, if the application allows. The transmitted data are constrained to the relational model;
- Remote Call Procedure (RPC) between applications. In this case the transmitted data, that is the call parameters and result, are in a flat format, a text to be interpreted at the arrival;
- Inter-Object Calls. The level of the interlocutors is lowered from that of the applications or processes to that of objects belonging to the same application: Java (RMI), CORBA (IIOP), Dot NET.
- Inter-Component Call. The fine granularity is recomposed and becomes coarser, at the conceptual level.
- Service Calls. Complete separation between access and implementation + the consequent process of discovery.

Asynchronous communications (broadcast and multicast)

- Point-to-point
- Publish-subscribe

Client-Server Communications

Client-Server System: a set o client-processes and a server-process that collaborate for realization of a computation task.

Server: passive entity that waits for service requests from clients for providing specialized services. According to this specialization we have several servers: database server, file server, e-mail server, application server, etc.

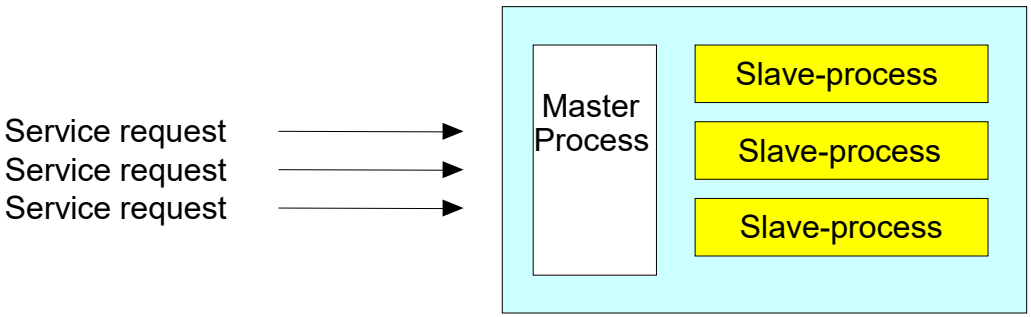
Client: active entity which obtains services from the server.

Clients and servers run usually on various machines.

Scenario:

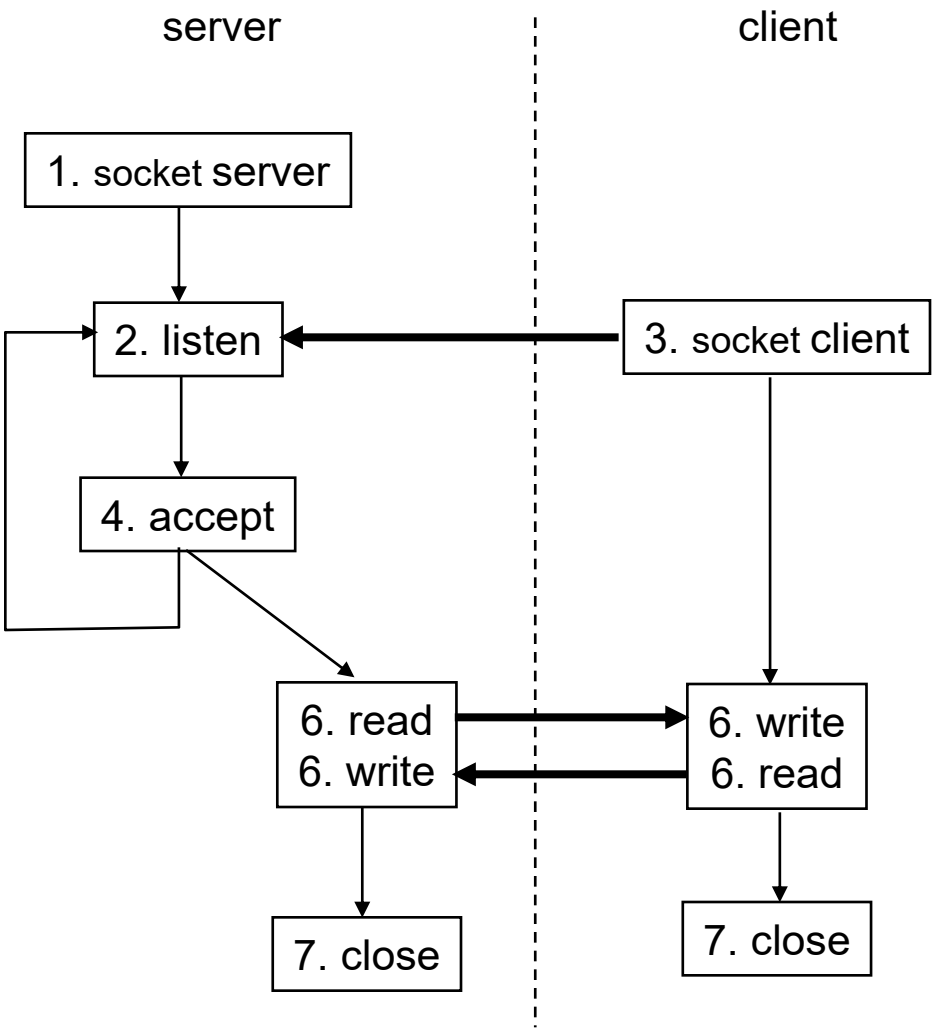
- The client sends a request to a server invia una richiesta ad un server across a network.
- The server responds wit the requested data.

• Theorically, n clients can send concurrently n service requests and the server has to respond to all of them.



Clinet-Server communications with sockets

1. The server registers the port number where it wants to use for communication.
2. The server “listens” to the specified port.
3. The client initiates a connection to the server using the specified port.
4. The server accepts the connection.
5. The server initiates a separated thread for serving the request.
6. The connection is established; now both applications can exchange data.
7. The client or the server closes the connection.



Client Implementation

0. Import the followings classes

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

1. Create a Socket object

```
Socket client = new Socket("hostname", portNumber);
```

2. Create an output stream that can be used to send info to the Socket

```
// Last arg of true means autoflush - flush stream when println is called
PrintWriter out = new PrintWriter(client.getOutputStream(), true);
```

3. Create an input stream to read the response from the server

```
BufferedReader in = new BufferedReader
(new InputStreamReader(client.getInputStream()));
```

4. Do I/O with the input and output Streams.

- For the output stream, PrintWriter, use print and println, similar to System.out.println

– For the input stream, BufferedReader, you can call read to get a single character or an array of characters, or call readLine to get a whole line

5. Close the socket when done

```
client.close();
```

Also closes the associated input and output streams

Server Implementation

0. Import the followings

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

1. Create a ServerSocket object

```
ServerSocket listenSocket = new ServerSocket(portNumber);
```

2. Create a Socket object from ServerSocket

```
while(someCondition) {
  Socket server = listenSocket.accept();
  doSomethingWith(server);
}
```

3. Create an input stream to read client input

```
BufferedReader in = new BufferedReader
(new InputStreamReader(server.getInputStream()));
```

4. Create an output stream that can be used to send info back to the client.

```
// Last arg of true means autoflush stream when println is called
PrintWriter out = new PrintWriter(server.getOutputStream(), true)
```

5. Do I/O with input and output Streams

– Most common input: `readLine`

– Most common output: `println`

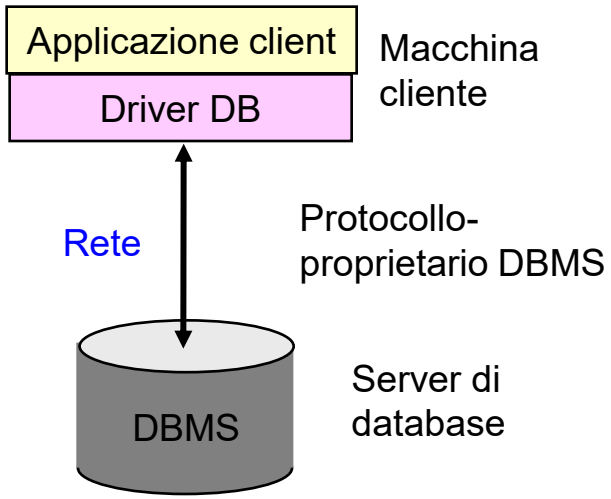
6. Close the socket when done

```
server.close();
```

This closes the associated input and output streams

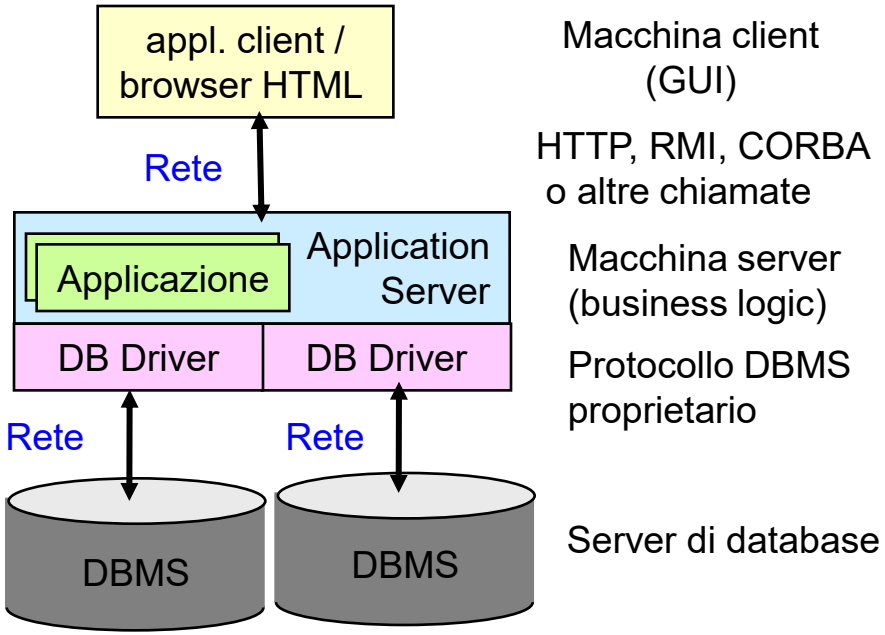
Client-Server Architectures

Two layers architecture



- The server manages data and processes complex applications.
- The client manages the user interface.

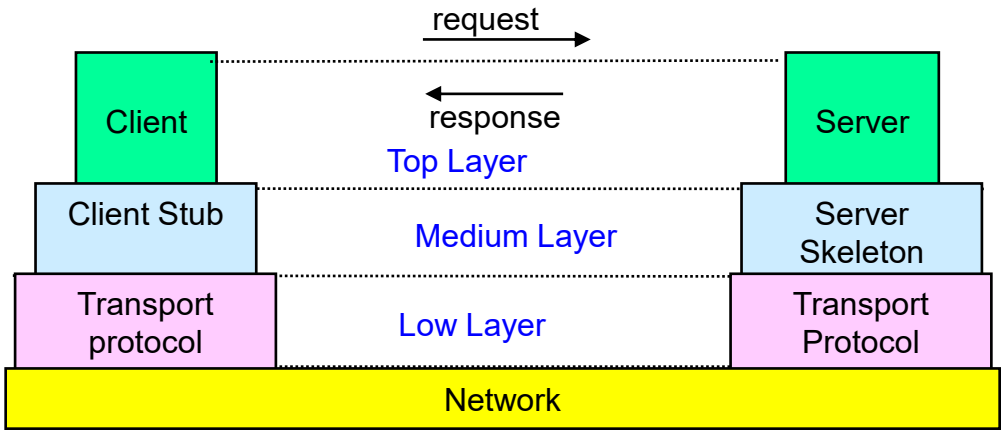
Three layers architecture



- Presentation layer
- Business logic layer
- Data layer

Remote Procedure Call

- Extension of the procedure call in UNIX that allows to a process that is executed in a machine client to call a function pertaining to a process that is executed in the remote server.
- RPC allows to pack the parameters of the call and send them from the local procedures to the remote ones. The result computed by the server procedures is also packed and transmitted to the client procedure.
- The RPC mechanism is planned for a heterogeneous environment.
- The RPC parameters must be send according to a specific protocol.
- The RPC passes parameters between procedures in various processes and not between objects.



ReMote Invocation (RMI)

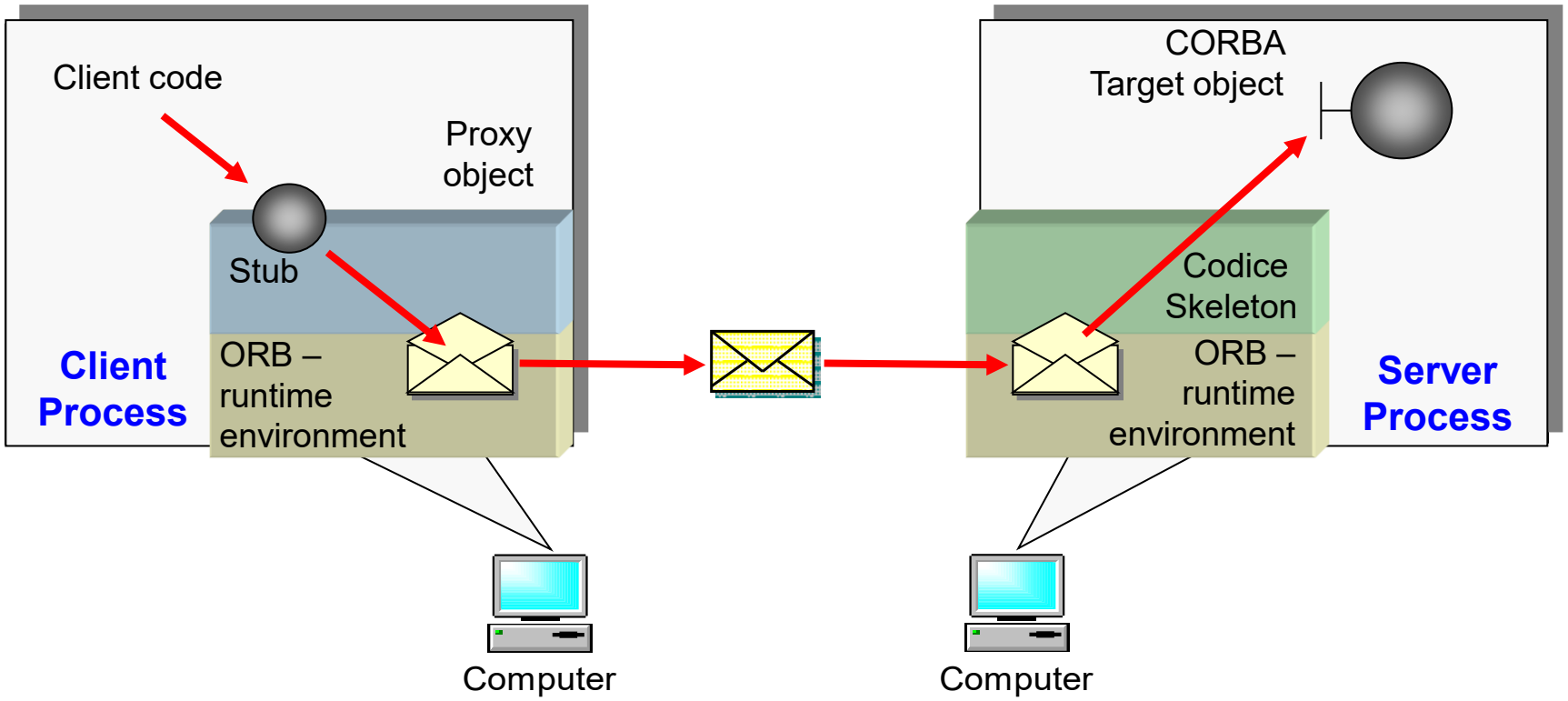
- RMI = a mechanism implemented in Java that allows to realize a method call defined in a various space of addresses (various JVMs).
- RMI supports not only the transfer of control between virtual machines, but also object transfer with “by-reference” or “copy-restore” parameter-passing methods.
- RMI supports communications between processes to a higher level of abstraction that the sockets.
- RMI allows support for: dynamic loading of data, call back from applets, and distributed objects model.
- Java/RMI is based on the serialization of objects (it codifies and transmits objects like data streams).
- The transmission protocol in RMI is aligned to the IIOP protocol used in CORBA (RMI-IIOP).

CORBA

- CORBA - Common Object Request Broker Architecture defines a specification for an heterogeneous object-oriented environment.
- An object request broker (ORB) is a middleware technology that manages communication and data exchange between objects. ORBs promote interoperability of distributed object systems because they enable users to build systems by piecing together objects- from different vendors- that communicate with each other via the ORB
- The relevant functions of an ORB technology are:
  - interface definition
  - location and possible activation of remote objects
  - communication between clients and object
- An ORB acts as a kind of telephone exchange. It provides a directory of services and helps establish connections between clients and these services.
- To communicate ORB uses a standard protocol for inter-ORB comunications (Internet Inter-ORB Protocol - IIOP).
- CORBA has IDL (Interface Definition Language) , a standard language for interface description. It separates the independent interfaces from the language they are implemented. IDL establishes a contract between clients and servers.

CORBA Invocation

- It allows the use of CORBA objects as they were normal elements in traditional programming languages .
- Proxy Object = local representative of a remote target object. It contains sufficient information in order to send requests to the remote object, encapsulating details on the network or port number. It supplies means in order to approach the object target in way “type-safe”, using of the standard types of the programming languages.

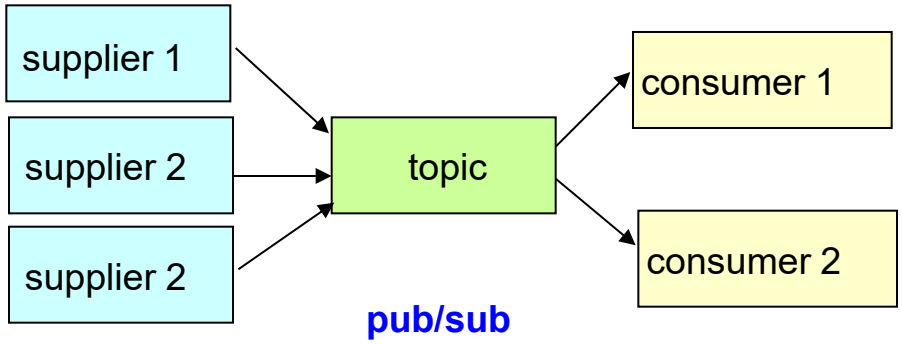


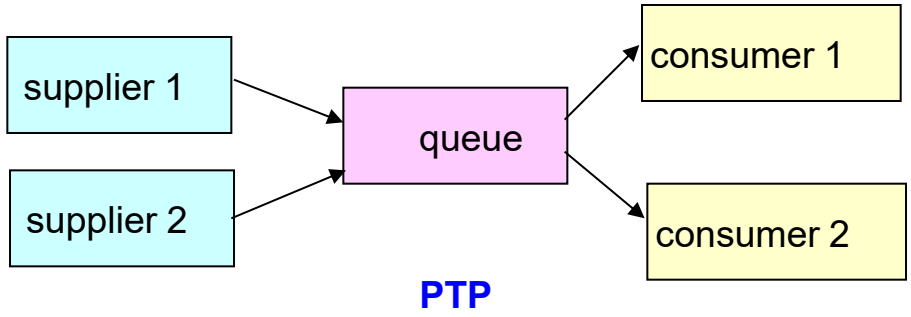
Event-Based Communication

- Client-server paradigm:
  - the sender selects the receiver,
  - there exist only one receiver.
- Event-Based Communication offers an alternative to this paradigm.
- Two event-based services:
  - Event Service
  - Notification Service.
- Event = Occurrence of whichever type, considered atomic, without inner structure.
- Event Notification (implicit invocation) = Dispatching of an event to all those interested in this event (event-consumers).
- Event Supplier = the entity that diffuses an event.
- Characteristics of the event-based communications:
  - they are many-to-many;
  - the event suppliers and consumers do not have themselves to be known between them, they are uncoupled and autonomous;
  - sending a notification is not blocking;
  - the notifications are self-describing. They cannot be verified statically like the invocations. Only when they are launched, after an analysis of the content, the consumer can elaborate to them.

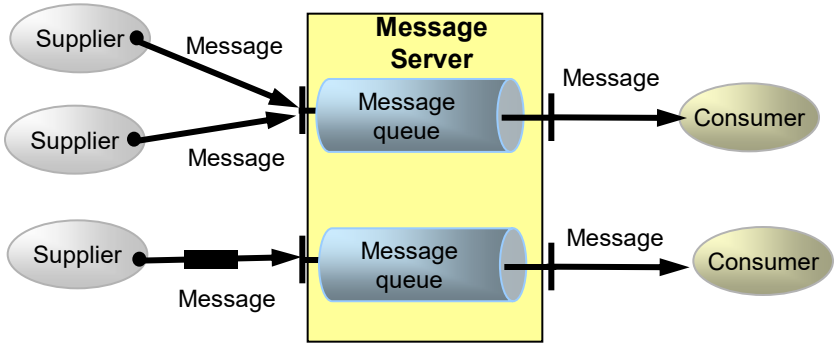
Asynchronous Communication

- Two models for sending/receiving messages:
  - pub/sub (publish/subscribe) – similar to the TV via cable. There are listeners record themselves near a channel of a event type or topic;
  - PTP (point-to-point) - similar to the voice mail that is accessed by persons interested in the messages stored by other persons. An event consumer consumes a message from a queue of messages.





Point-To-Point

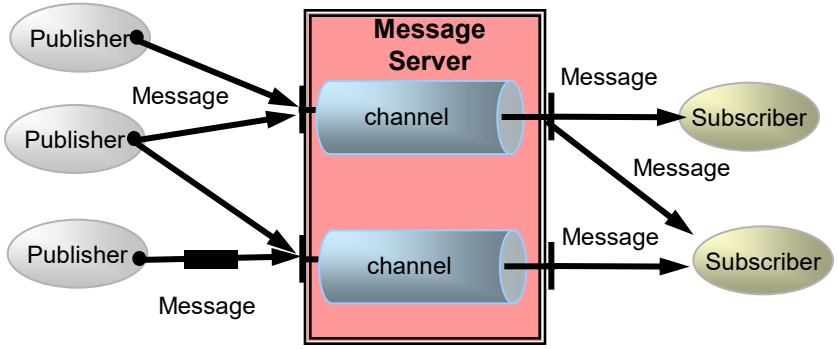


1. A message supplier sends a message to the desired queue.
2. A message consumer asks the server for registering it for a desired queue.
3. The server delivers a message to client registered as soon as the message is available.

- In a PTP (Point-to-Point) application every message is sent to a specific queue where is stored. The consumers extract their messages from this queue.
- The queues store all the messages until all have been consumed or expired.
- PTP messages have the following characteristics:
  - Every message has only a consumer.
  - Dependencies of time between a sender and a consumer do not exist. The consumer extracts its message independently of the fact if at the moment of the emission of the message was active or not.
  - The supplier confirms the message processing.
- PTP is used when every sent message must be processed by only one consumer.

Pub/Sub

- *Pub/Sub is used for applications that send a message more than one client.*
- In a product or an application pub/sub
  - the clients send messages to a channel. They are anonymous and can dynamically subscribe or publish in the hierarchy of contents. The system guarantees the distribution of the messages in arrival from the multiple producers of a channel to multiple subscribers of the same channel.
  - the topic sends the messages distributing them to the existing subscribers.

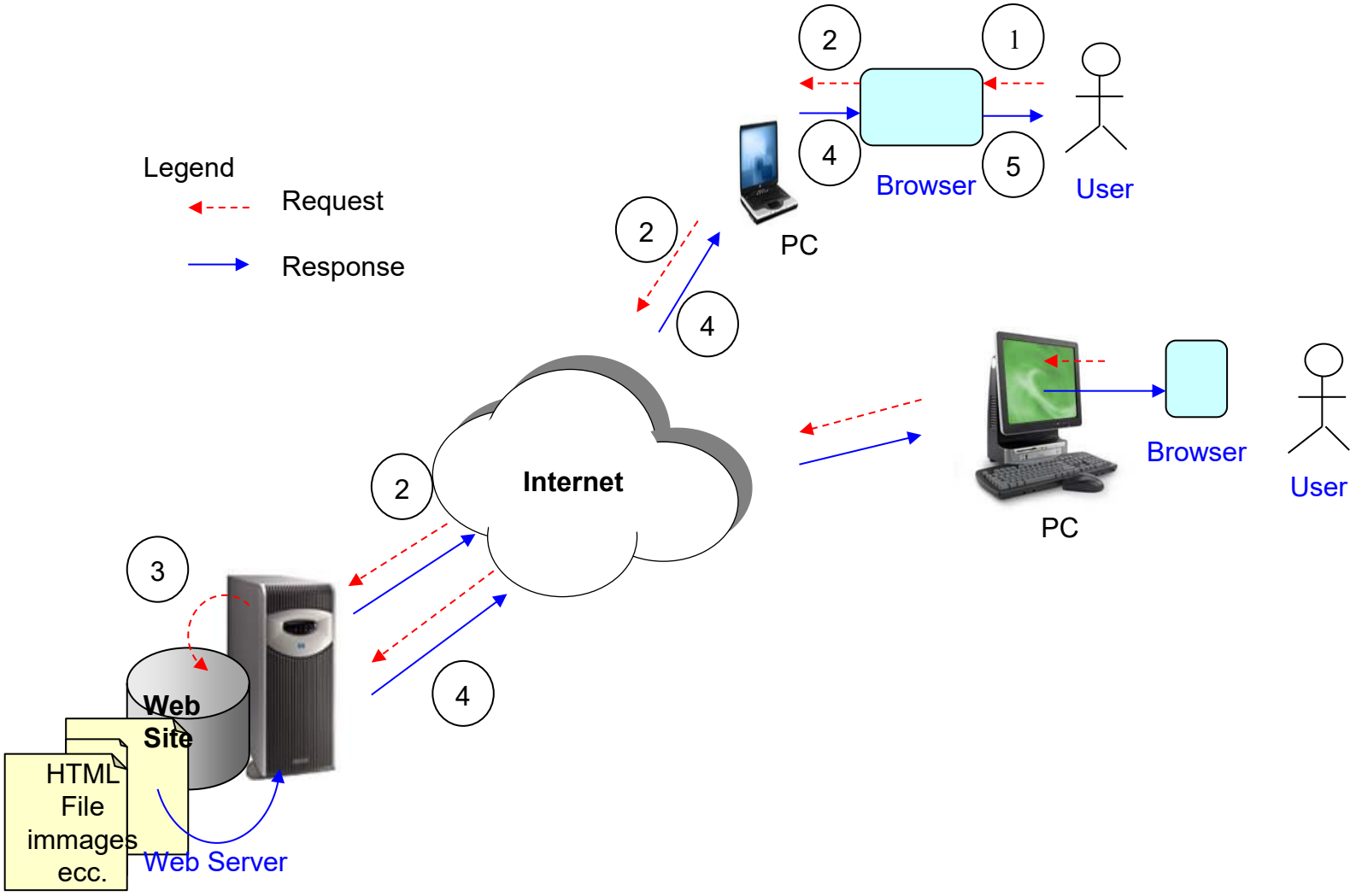


1. A message producer creates a message and publishes it in the wished channel.
2. A message consumer subscribes to a channel of interest.
3. A message server delivers messages to all the entities which subscribed to a specified channel.

- Pub/sub has the following characteristics:
  - Each message can have several consumers.
  - There is a temporal dependency between producers and consumers, because the consumer that subscribes to a channel it can only consume the messages publishes after the subscription and the subscriber must continue to be active if it wants to receive messages.

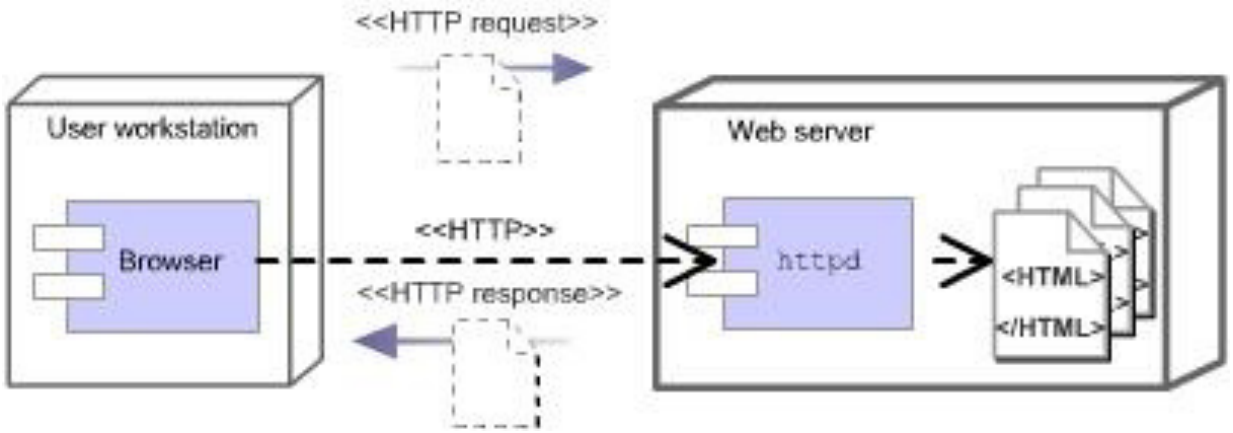
Web Programming

Communications in Web



The HTTP Protocol

- The Hypertext Transfer Protocol (HTTP) transfers files from the server to the client in conjunction with the related Hypertext Markup Language (HTML) standard.
- HTTP supports only one request per connection. This means that the client connects to the server to retrieve one document and then disconnects.
- HTML is a document display language that allows users to link from one document to another or images and other media objects to be embedded in an HTML document. The media objects are stored in files on a server. HTTP also retrieves these files. HTTP can, therefore, be used to transmit any file that conforms to the Multipurpose Internet Mail Extensions (MIME) specification.



The HTTP Protocol

Paradigm "Call-reply" for communications client-server in Web:

1. HTTP Request : the client Web, the browser, asks for information to the server with a message built in conformity with the protocol HTTP;
2. HTTP Response: the server grants the information requests in another message built in conformity with the same protocol.

- Often, the reply is not a page static HTML, but should be dynamically generated.
- There exist more possibility of to extend the functionality of the server to dynamically generate HTML pages:

1. To interface the server with applications in a position of to HTML pages HTML means I' interface CGI.
2. To use a server Web that admits some plug-ins that allow a direct access to the resources.
3. To interface the server Web with applications Java by means of servlets.
4. Active Server Page (ASP) - corresponding to the HTML pages on the server-side that use a version of the VBScript language.
5. Java Server Page (JSP) - corresponding to the HTML pages on the server-side that use a version of the Java language.

Methods of the HTTP protocol

Method =

"GET" | "HEAD" | "POST" | "OPTIONS" | "PUT" | "DELETE" | "TRACE" | "CONNECT"

GET

Meant: It asks for an internet resource (HTML pages, images etc.) identified with an URI address. It is used to obtain information like a document or the result of a database query. The request has not a body.

Exemple: `GET /servlet/proc?LastName=Smith&FirstName=John HTTP/1.0`

HEAD

Meant: It asks for meta-data (the date of the last change, the file dimension or type) of an entity of information identified with an address URI. The request has not a body.

Exemple: `HEAD /servlet/proc/index.htm HTTP/1.0`

POST

Meant: It sends to the server data encapsulated and eventually codified in the request body . It is more secure than the method GET. It is used to transmit to the server some sensible data of the client like the number of the credit card or information to update a database. It is used for HTTP requests with numerous and/or long parameters.

Exemple: `POST /servlet/proc HTTP/1.0`

The HTTP Request Structure

Client_Request= Method **Request_*URI*** ***HTTP_Version***
(general_header | request_header | ***entity_request*****)**
[*entity_body*]

GET Request

```
GET /servlet/Search?keywords=display+computer HTTP/1.1
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0)
Accept: image/gif, image/jpg, */*
Host: www.electronicstore.com
Accept-Encoding: gzip
Connection: Keep-Alive
Cookie: userID=id456578
Referer: http://www.electronicstore.com/findbooks.html
```

POST Request

```
POST /servlet/proc HTTP/1.0
User-Agent: Mozilla/4.04 [en] (WinNT; I)
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */*
Host: www.datamat.it
Content-type: application/x-www-form-urlencoded
Content-length: 102
Connection: Keep-Alive
Accept-Charset: iso-8859-1,*,utf-8
Referer: http://www.datamat.it/corsi.html

LastName=Smith&FirstName=John&day=28&month=02&year=1963&CNP=1630228400371&sex=M
```

HTTP Response

Server_Response* = *HTTP_Version* *State_Code* *State_Description
(general_header | request_header | *entity_request*)
[*entity_body*]

HTTP Response

```
HTTP/1.0 200 Document follows
Server: Apache-Go-WebServer/4.6
Date: Wed, 20 Apr 1999 20:09:34 GMT
Accept-Ranges: bytes
Content-type: text/html
Last-Modified: Wed, 20 Apr 1999 19:05:12 GMT
. . . . Response Body. . . .
```

Frequent Content-type :

text/html	HTML Document
text/css	CSS Style Sheet
image/gif	GIF image
image/jpeg	JPEG image
image/png	PNG image

State Codes and State Description
200: Everything is OK
301: Moved permanently
302: Found (the request was redirected to another URL/URI)
305: Use Proxy (the request must use a proxy to access the resource requested)
401: Unauthorized
403: Forbidden
404: Not Found

Web Client

- Client Web = Application that sends and receives information from a server HTTP.
- Browser Web is the commonest type of client Web which:
  - Uses the protocol HTTP for:
    - Transforming of the coming requests of the user across a GUI in HTTP requests;
    - Transforming of the HTTP responses coming from a Web server in content visualized in a GUI;
  - Visualizes the resources codified in HTML;
  - Supplies support for some mechanisms of scripting;
  - Supplies the environment for Java applets execution (JVM);
  - Supports plug-in;
- HTTP requests content URL addresses that identify remote internet resources.
- HTTP responses are documents called Web pages with a multimedia content or simple HTML pages with text, static or animated images, hyperlinks, GUI components, audio or video clips etc.

Web Server

Web Server = Application on the server side of which the main task it is to receive, manage or delegate to others HTTP requests and to generate or to dispatch the responses.

In the simpler scenarios, a Web server: 1) receives GET and POST requests, 2) reads data from a local file indicated with its URL address by the Web client and 3) sends the data in the file back to the web client.

The more advanced Web servers :

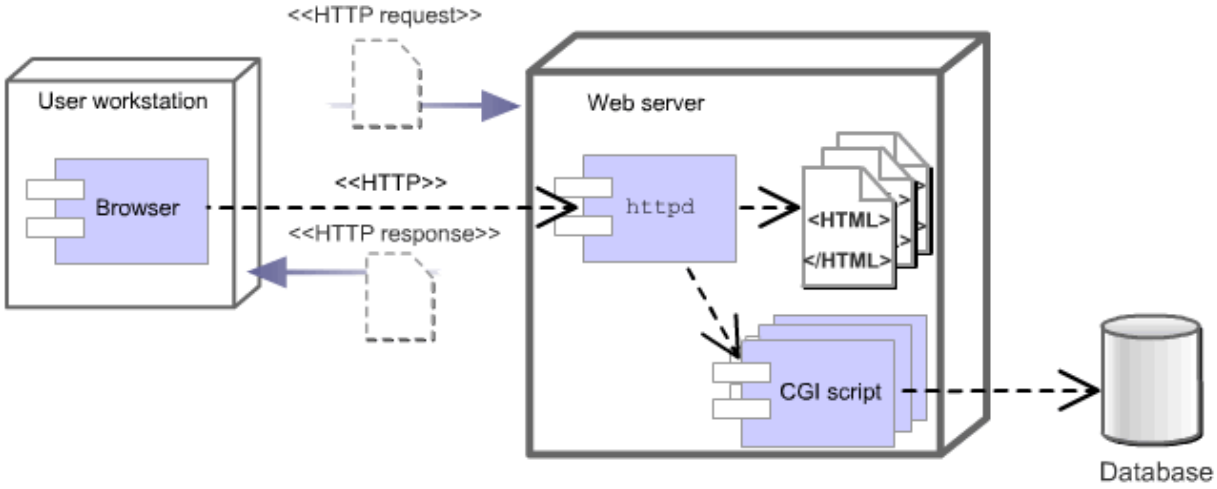
- support concurrent requests from a scalable number for clients. For this they manage a pool of threads that it is used in order to handle responses from clients when these arrive;
- implement several kinds of access control and
- support several APIs to extend the functionality of the server Web and to dynamically generate Web documents in specific ways to the application; they manage information regarding the running session between successive requests arrived from a client, correcting the stateless characteristic of protocol HTTP;
- manage the cache of the responses in order to supply the data already elaborated to requests that repeat a previous demand;
- control the environment configuration and
- supply a log mechanism.

Web Server Extensions

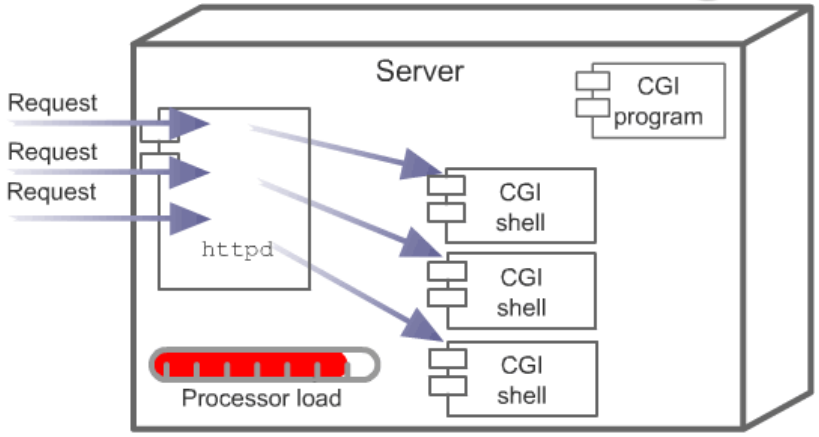
- The Web page is based on data submitted by the user
  - E.g., results page from search engines and order confirmation pages at on-line stores
- The Web page is derived from data that changes frequently
  - E.g., a weather report or news headlines page
- The Web page uses information from databases or other server-side sources
  - E.g., an e-commerce site could use an extension of the Web server to build a Web page that lists the current price and availability of each item that is for sale.

Execution of CGI Programs

- Very early in the development of HTML, the designers created a mechanism to permit a user to invoke a program on the Web server. This was called the Common Gateway Interface (CGI).
- When a Web site includes CGI processing, this is called a Web application.
- The CGI specification defines how the data is packaged and sent in the HTTP request to the server.
- At runtime, a CGI program is launched by the Web server as a separate operating system (OS) shell.
- Each new CGI request launches a new operating system shell on the server. The shell includes an OS environment and process to execute the code of the CGI program, which resides within the server's file system.



Advantages and Disadvantages of CGI Programs



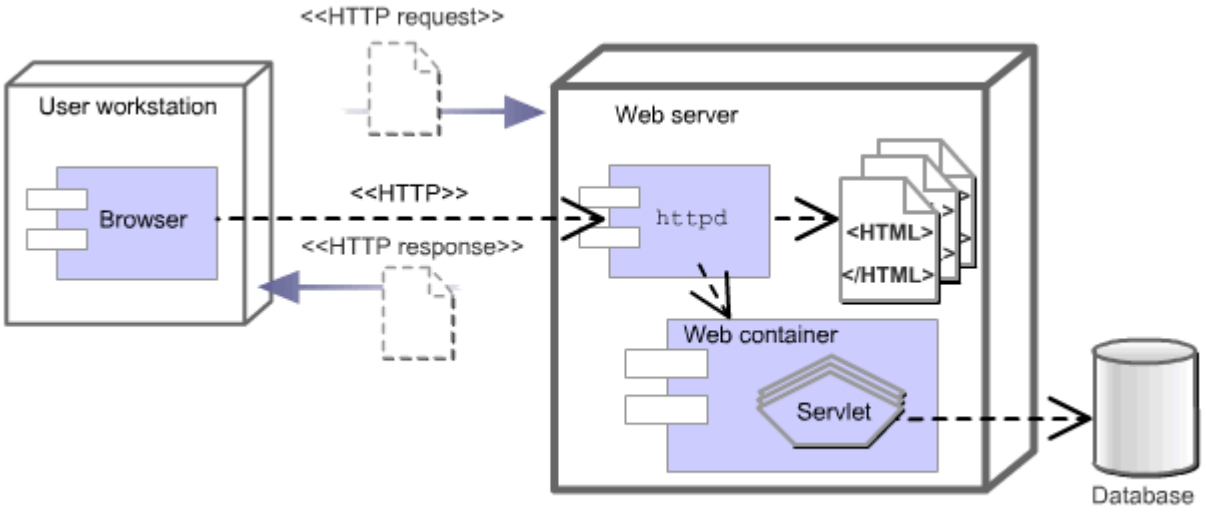
Advantages:

- Programs can be written in a variety of languages, (often Perl).
- The CGI programs do not have concurrency conflicts with other HTTP requests executing the same CGI program.
- All Web service providers support CGI programs.

Disadvantages:

- The response time of CGI programs is very high, because CGI programs execute in their own OS shell.
- CGI is not scalableThere is a limit on the number of separate operating system processes a computer can run.
- The CGI script has to generate an HTML response, so the CGI code is mingled with HTML. This is not a good separation of presentation and business logic.
- CGI programs are often platform dependent.

Servlets on the Web Server



- Unlike CGI programs, servlets run within a component container architecture.
- This container is called the Web container (the previous servlet engine). It is a Java™ virtual machine (JVM™) that supplies an implementation of the servlet API and manage them in order to respond to HTTP requests.
- The Jakarta Tomcat project is an open source implementation of the Servlet and JSP specifications.

Client-side Programming

Standard Markup Languages

- SGML (Standard Generalized Markup Language)
  - Specifies rules for creation of markup languages
  - An extensible language used for structured language specification (a meta-language).
- HTML (HyperText Markup Language).
  - The most popular SGML, used for Web page specification.
  - Includes *tags* with well-defined semantics
  - Very simple
  - Includes hyper-links for hyper-text creation.
  - Provides a sufficient support for building forms
  - Separates the document presentation of the document structure.
  - Limitations:
    - The page has no internal structure
    - The HTML pages are not reusable
    - The tags are predefined, so the language is not extensible
- XML (eXtensible Markup Language).
  - A W3C standard which eliminates the limitations of HTML.

Pagine HTML

- Plain text documents with .html or .htm extension.
- Reside in the web server machine
- Composed from text interrupted by HTML elements . Each element is defined by a HTML tag.
- A tag is a keyword delimited by < >. Exemples: <html> <title> <body>.
- A tag can have a closing tag to delimit a text where the tag is applied.

Web page structure= Header + Body

```
<HTML>
  <HEAD>
      Header
  </HEAD>
  <BODY attributes>
      Body
  </BODY>
</HTML>
```

Header Structure:

```
<HEAD>
<TITLE>text </TITLE>
</HEAD>
```

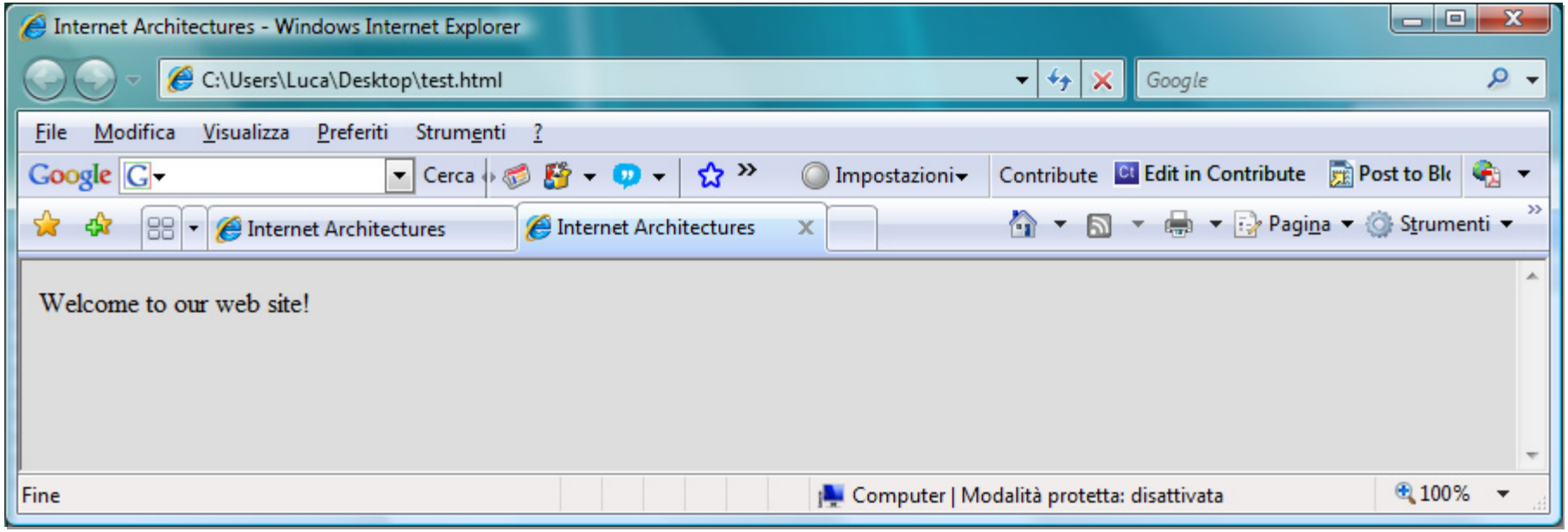
Body Structure:

```
<BODY TEXT=".." LINK=".." ALINK=".."
VLINK=".." BACKGROUND="..">
   tags + text
</BODY>
```

The tags may include optional or compulsory *attributes*. An attribute has a name end a value domain. An attribute value is a string delimited by “”.

A simple HTML file

```
<BODY BGCOLOR="yellow" LINK="#FFFF00">
<HTML>
<!– This is a comment  -->
<HEAD>
<TITLE>Internet Architectures</TITLE>
</HEAD>
<BODY BACKGROUND="" BGCOLOR="#DDDDDDf" TEXT="#000000" LINK="#0000ff" VLINK="#800080"
    ALINK="#ff0000">
<P>Benvenuto al nostro sito!</P>
</BODY>
</HTML>
```



HTML Tags (I)

- **titles**
  Every HTML document needs a title. Here is what you need to type:
  `<title>My first HTML document</title>`
- **add headings and paragraphs:**
  `<H1>text</H1>, <H2>text</H2>,. . ., <H6>text</H6>`
  `<P>text</P>` or with attributes: `<P ALIGN=LEFT>`
- **allignments:**
  `<CENTER>text</CENTER>` or with an attribute: `<H1 ALIGN=CENTER>text</H1>`
- **to force a line break:**
  `<BR>`
- **preformatted text:**
  `<pre>text</pre>`
- **unordered list**

```
<UL>                          introduces a list
   <LI> text                  introduces an item in the list
   <LI> text
 </UL>                        termina la lista
```

- **ordered list** (variants: `<ol type = "I">`, `<ol type = "a">`):

```
<OL>
      <LI> text               introduces an item in the list
      <LI> text
 </OL>                        ends the list
```

- **separation line**
  `<HR>`
- **division** (similar to <p> , but it can be nested):
  `<DIV>`

HTML Tags (II)

- **Adding a bit of emphasis**

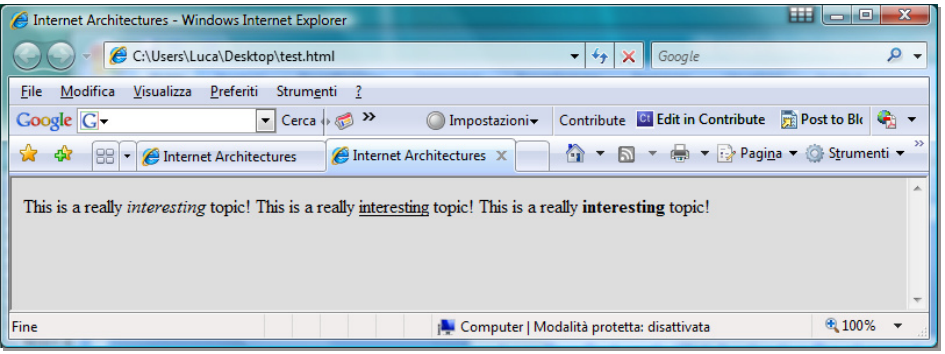
```
This is a really <em>interesting</em> topic!
This is a really <u>interesting</u> topic!
This is a really <strong>interesting</strong> topic!
```

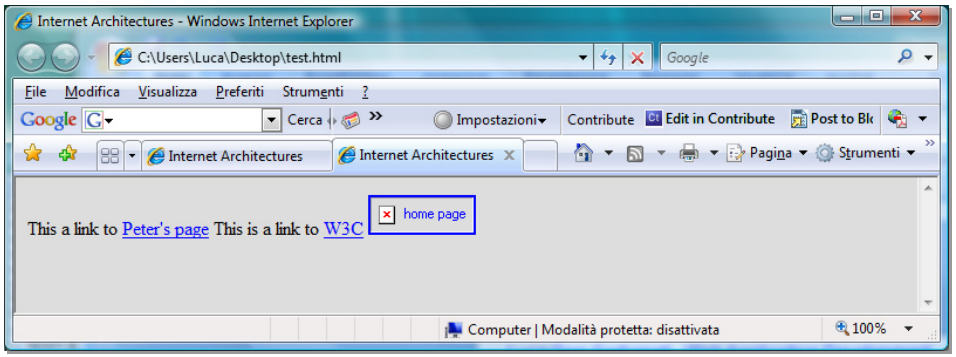
- **Adding images**

```
<img src="peter.jpg" width="200" height="150">
```

- **Adding links to other pages**

```
This a link to <a href="peter.html">Peter's page</a>
This is a link to <a href="http://www.w3.org/">W3C</a>
<a href="/"><img src="logo.gif" alt="home page"></a>
```





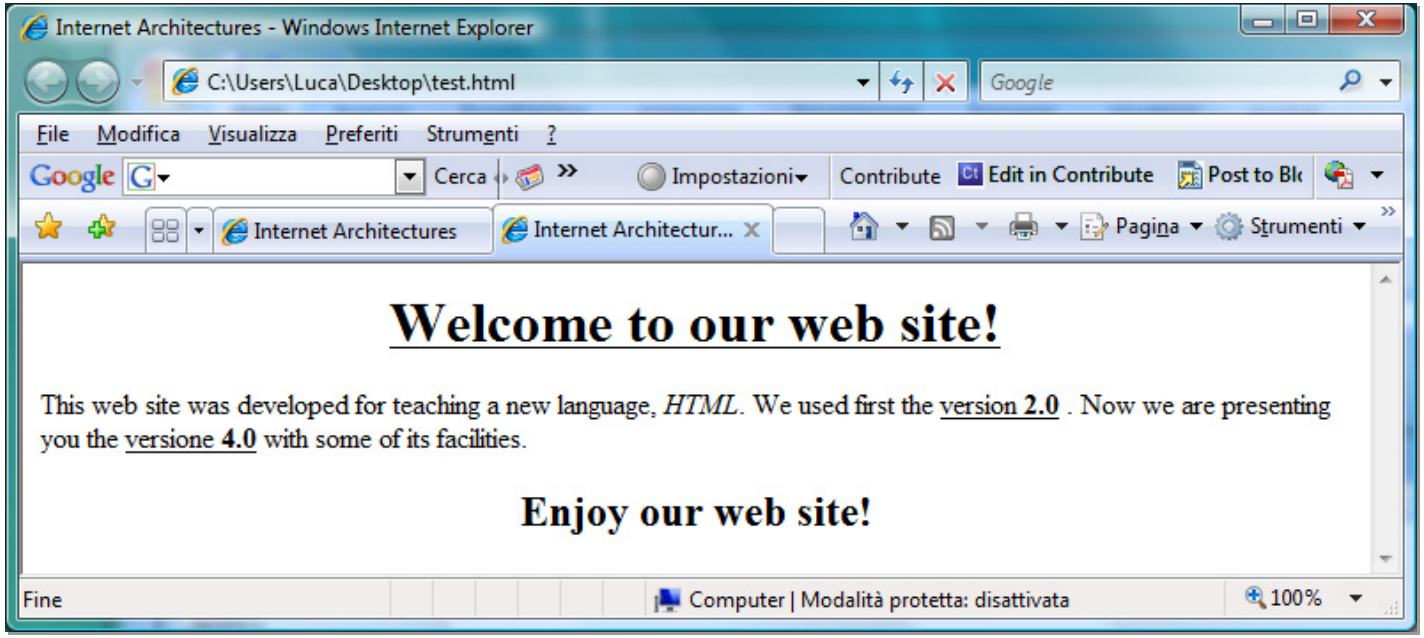
- **How to introduce non-breaking spaces or special characters**

```
Beverages, such as Coca Cola,have attained world-wide popularity.
```

Copyright sign	`©`	Copyright © 1999 W3C
Registered trademark	`®`	MagiCo ®
Trademark	`™`	Webfarer™
Less than	`<`	<
Greater than	`>`	>
Ampersand	`&`	&
em dash	`—`	—
quotation mark	`"`	"

Text Formatting

```
<HTML>
<HEAD>
<TITLE>Internet Architectures</TITLE>
</HEAD>
<BODY>
<H1 ALIGN = "center" > <U>Welcome to our web site!</U></H1>
<P>This web site was developed for teaching a new language, <EM>HTML</EM>. We used first the
    <U>version<STRONG> 2.0</STRONG></U> .
Now we are presenting you the <U>versione<STRONG> 4.0</STRONG></U> with some of its facilities.</P>
<H2 ALIGN = "center"> Enjoy our web site!</H2>
</BODY>
</HTML>
```



Using hyper-links

```
<HTML>
<HEAD>
<TITLE>Commerce</TITLE>
</HEAD>
<BODY>
<H1 ALIGN = "center" > <U>Welcome to Romania Plaza!</U></H1>
 <OL>
    <LI><A HREF="#html">HTML</A></LI>
    <LI><A HREF="#version">VERSION</A></LI>
    <LI><A HREF="#email">EMAIL</A></LI>
 </OL>
 <UL>
    <LI><A HREF="apparel.html"></A>Apparel Online Store</LI>
    <LI><A HREF="pizza.html">Pizzeria Napoli</A></LI>
    <LI><A HREF="bar.html">Bar Plaza</A></LI>
 </UL>
<H2 ALIGN = "center">Enjoy!</H2>
<P>This  web site was developed for teaching a new language, <A NAME="html"> <U> HTML</U></A>. We used
    HTML beginning with its <A NAME="version"> <U> version  2.0 </U><A>. Now we present you the
    <U>version<STRONG> 4.0</STRONG></U> with some of its facilities.</P>
<P>Call us at: <A NAME="email" HREF="mailto:contact@plaza.ro"> contact@plaza.ro</A>
</BODY>
</HTML>
```

Images in HTML Pages

```
<IMG WIDTH=... BORDER=... WIDTH=... HEIGHT=... ALT=... SRC=... ALIGN=...>
```

- **Flowing text around images**

```
<p><img src="sun.jpg" alt="sunburst graphic" width="32" height="21"
  align="left">This text will beflowed around the right side of the
  graphic.</p>
<p><img src="sun.jpg" alt="sunburst graphic" width="32" height="21"
  align="right">This text will beflowed around the left side of the
  graphic.</p>
<p><img src="sun.jpg" alt="sunburst graphic" width="32" height="21"
  align="left">This text will beflowed around the right side of the
  graphic.<br clear="all">This starts a new line below the floated
  image.</p>
```

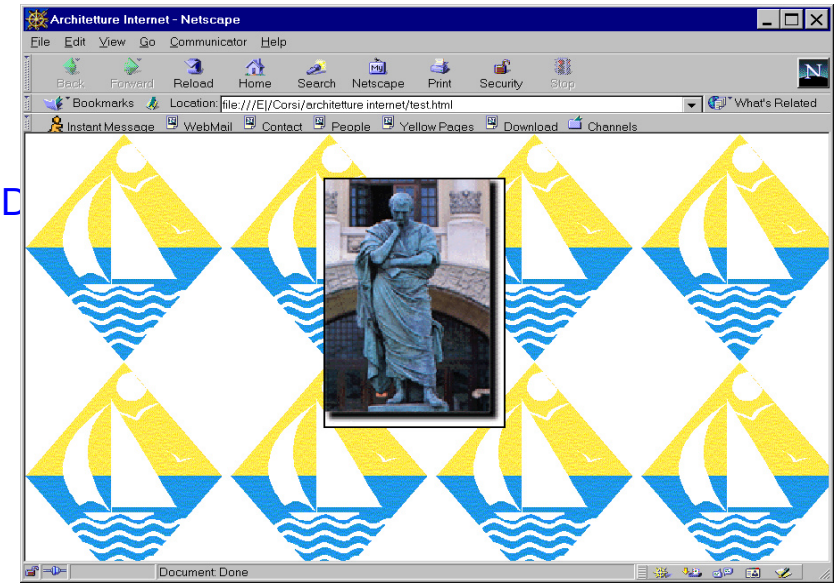
- **Clickable regions within images**

```
<A HREF="principale.html">
<IMG SRC="button/imag.jpg" WIDTH="120" HEIGHT="80" BORDER="0"
ALT="Pagina principale"></A>

<MAP NAME="europa">
 <AREA SHAPE="rect" COORDS="10, 80, 20, 100" HREF="spagna.html">
 <AREA SHAPE="poly" COORDS="35,100,35,120,50,105,45,100"
HREF="italia.html">
 . . .
</MAP>
<IMG ISMAP SRC="europa.gif" USEMAP="#europa# ALIGN="middle">
```

Immagini “hot-link”

```
<HTML>
<HEAD>
<TITLE>Architetture Internet</TITLE>
</HEAD>
<BODY BACKGROUND = "vela.gif"  BGCOLOR = "#CDCDCD
<BR>
<BR>
<CENTER>
<A HREF = “ovidio.html”>
<IMG SRC="ovidio.gif" BORDER = "2"
  HEIGHT = "290" WIDTH = "200" >
</CENTER>
</BODY>
</HTML>
```

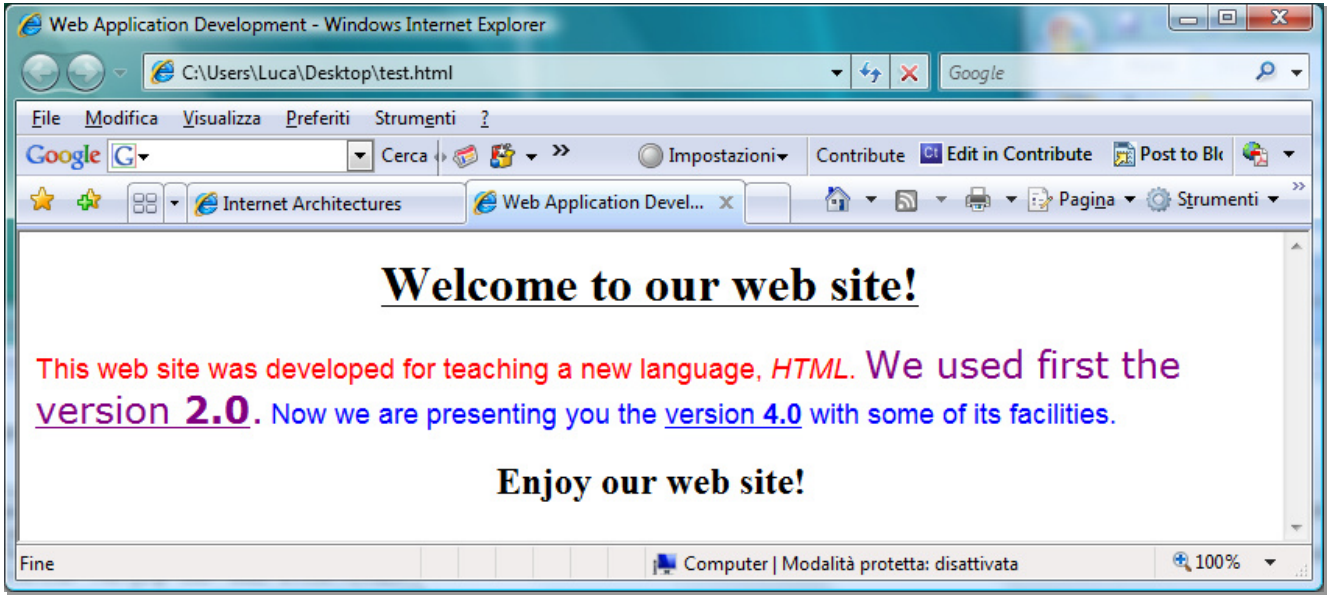


Fonts and colors in HTML:

```
<BODY TEXT=“#000000” LINK=“#0000EE” VLINK=“#502090” ALINK=“#00FF00#”
        BACKGROUND=“logo.jpg”>
. . . .
<FONT COLOR=“blue”  FACE=“Arial” SIZE=“+1”> testo </FONT>
. . . .
```

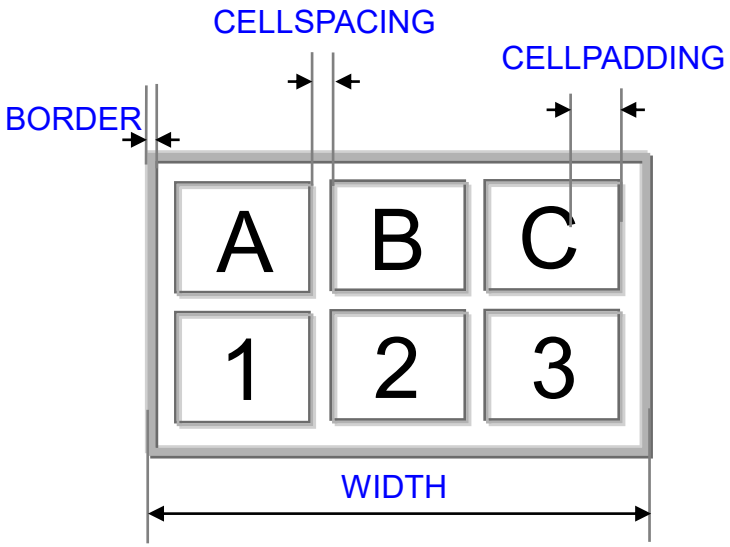
Fonts and Colors

```
<HTML>
<HEAD>
<TITLE>Web Application Development</TITLE>
</HEAD>
<BODY>
<H1 ALIGN = "center" > <U>Welcome to our web site!U></H1>
<P><FONT COLOR="red" SIZE="+1" FACE="Arial">This web site was developed for teaching a new language,
    <EM>HTML</EM>. </FONT>
<FONT COLOR="purple" SIZE="+2" FACE="Verdana">We used first the <U>version<STRONG>
    2.0</STRONG></U>.</FONT>
<FONT COLOR="blue" SIZE="+1" FACE="Helvetica">Now we are presenting you the <U>version<STRONG>
    4.0</STRONG></U> with some of its facilities.</FONT></P>
<H2 ALIGN = "center">Enjoy our web site!</H2>
</BODY>
</HTML>
```



TABLE

```
<TABLE> introduce una tabella:
 <TABLE CELLPADDING=.. CELLSPACING=.. BORDER=1 ALIGN=.. WIDTH=.. BGCOLOR=..>
  <THEAD ALIGN= . . .>                    heading introduction
     <TR ALIGN=…>
       <TH ALIGN= . . .>. . . </TH>       table column title
       <TH ALIGN= . . .>. . . </TH>       table column title
     </TR>
  </THEAD>
  <TR ALIGN=…>                            introduces a row
    <TD> testo </TD>                      table column
    <TD> testo </TD>
  </TR>                                   terminates a row
  <TFOOT ALIGN= . . .>                    table closing
    <TR ALIGN=…>
       <TH ALIGN= . . .>. . . </TH>
       <TH ALIGN= . . .>. . . </TH>
     </TR>
  </TFOOT>
</TABLE>
```




```
<table border="1">
<tr><th>Year</th><th>Sales</th></tr>
<tr><td>2000</td><td>$18M</td></tr>
<tr><td>2001</td><td>$25M</td></tr>
<tr><td>2002</td><td>$36M</td></tr>
</table>
```

Year	Sales
2000	$18M
2001	$25M
2002	$36M

```
<table border="1"
cellpadding="10">
```

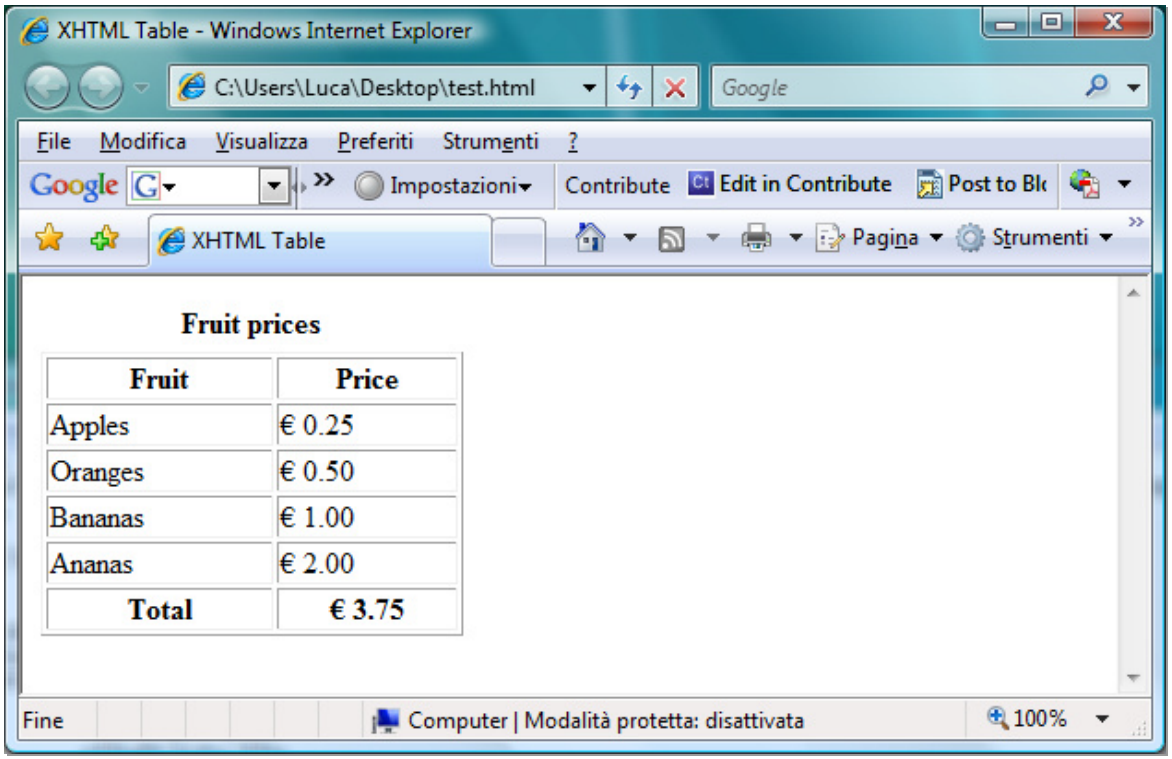
Year	Sales
2000	$18M
2001	$25M
2002	$36M

```
<table border="1" cellpadding="10"
cellspacing="10">
```

Year	Sales
2000	$18M
2001	$25M
2002	$36M

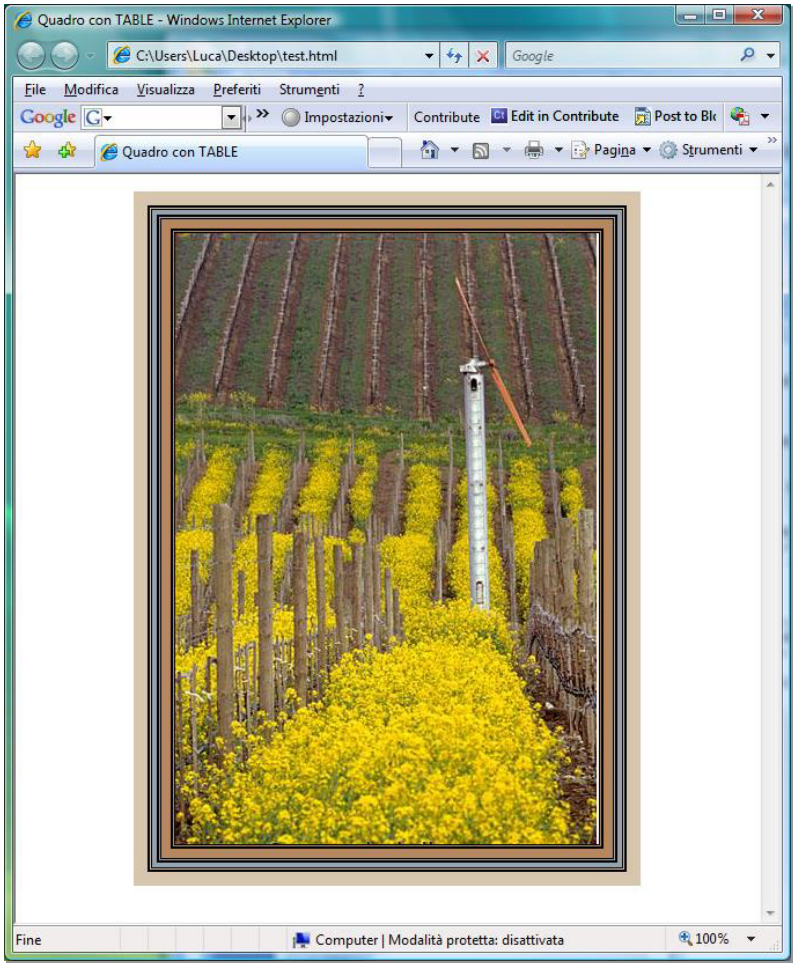
```
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" http://www.w3.org/TR/xhtml1/DTD/xhtml1-
   strict.dtd>
<html xmlns = "http://www.w3.org/1999/xhtml">
     <head>
         <title>Una tabela XHTML</title>
     </head>
  <body>
   <table border = "1" width = "40%" summary = "La tabella presenta i prezzi della frutta">
       <caption><strong>Prezzi della frutta</strong></caption>
            <thead>
               <tr>
                  <th>Frutta</th>
                  <th>Prezzo</th>
               </tr>
            </thead>
            <tbody>
               <tr>
                  <td>Mele</td>
                  <td> € 0.25</td>
               </tr>
               <tr>
                  <td>Aranci</td>
                  <td> € 0.50</td>
               </tr>
               <tr>
                  <td>Banane</td>
                 <td> € 1.00</td>
               </tr>
                 <tr>
                  <td>Ananas</td>
                  <td> € 2.00</td>
               </tr>
            </tbody>
            <tfoot>
               <tr>
                  <th>Totale</th>
                  <th> € 3.75</th>
               </tr>
            </tfoot>
     </table>
   </body>
 </html>
```

Building a table



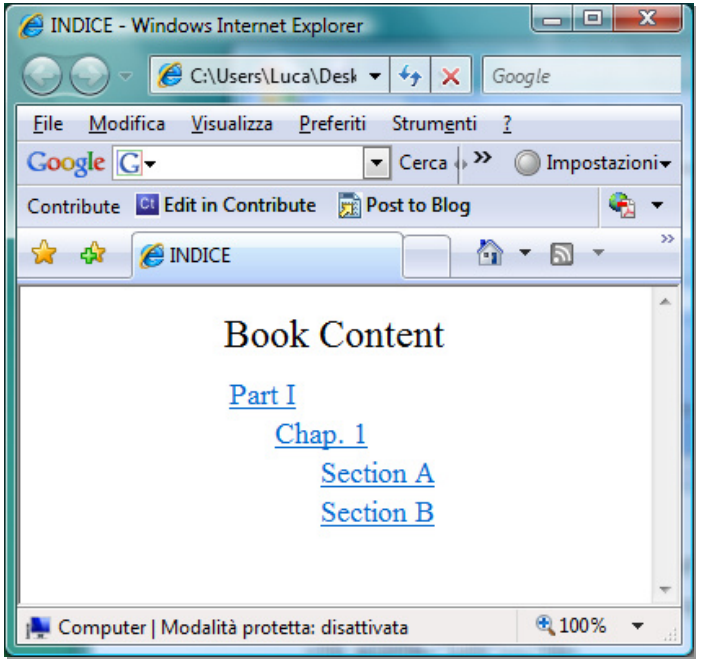
Frame with TABLE

```
<HTML>
<HEAD>
<TITLE>Quadro con TABLE</TITLE>
</HEAD>
<BODY>
<CENTER>
 <TABLE BORDER="12" BORDERCOLOR="#D7C5AD" BGCOLOR="#000000">
  <TR>
   <TD>
    <TABLE BORDER="4" BORDERCOLOR="#97A4AD">
     <TR>
      <TD>
       <TABLE BORDER="8" BORDERCOLOR="#B58861">
          <TR>
           <TD>
            <IMG SRC="windmill.jpg" WIDTH="267"
       HEIGHT="154‟ ALT=”Landscape with windmill.">
           </TD>
          </TR>
       </TABLE>
      </TD>
     </TR>
    </TABLE>
   </TD>
  </TR>
 </TABLE>
</CENTER>
</BODY>
</HTML>
```



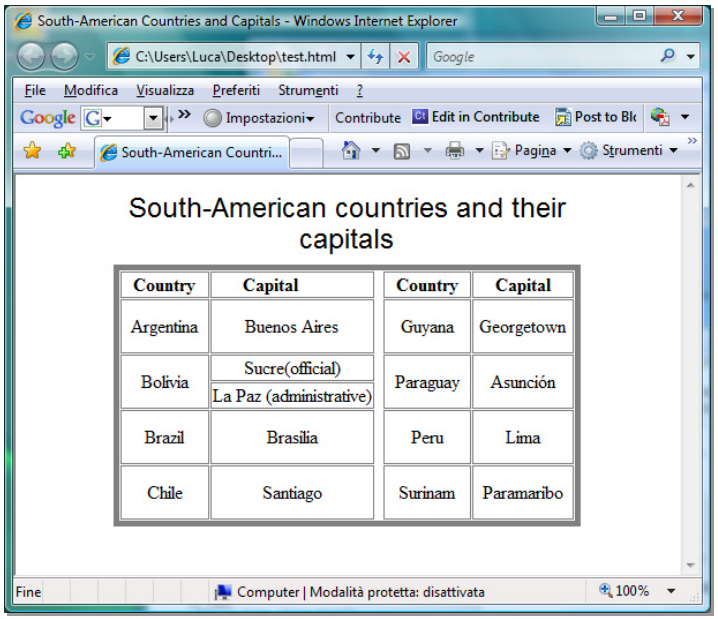
Book Contents

```
<HTML><HEAD><TITLE>INDICE</TITLE></HEAD>
<BODY><CENTER>
<TABLE WIDTH="200">
    <CAPTION><FONT SIZE="5">Book Content</FONT>
    </CAPTION>
    <TR>
        <TD HEIGHT="3" WIDTH="16%"></TD>
        <TD WIDTH="14%"></TD>
        <TD WIDTH="14%"></TD>
        <TD WIDTH="44%"></TD>
        <TD WIDTH="12%"></TD>
    </TR>
    <TR>
        <TD></TD> <TD COLSPAN="3"><FONT SIZE="4">
        <A HREF="P-I">Part I</A></FONT></TD>
        <TD></TD>
    </TR>
    <TR>
        <TD COLSPAN="2"></TD>
        <TD COLSPAN="2"><FONT SIZE="4"><A HREF="P-I_C-1">Chap. 1</A></FONT></TD>
        <TD></TD>
    </TR>
    <TR>
        <TD COLSPAN="3"></TD>
        <TD><FONT SIZE="4"><A HREF="P-I_C-1_S-A">Section A</A></FONT></TD>
        <TD></TD>
    </TR>
    <TR>
        <TD COLSPAN="3"></TD>
        <TD><FONT SIZE="4"><A HREF="P-I_C-1_S-B">Section B</A></FONT></TD>
        <TD></TD>
    </TR>
</TABLE>
</CENTER>
</BODY>
</HTML>
```



ROWSPAN and NOWRAP

```
<HTML><HEAD><TITLE>SOUTH-AMERICAN COUNTRIES AND CAPITALS</TITLE></HEAD>
<BODY><CENTER>
<TABLE BORDER="5" BORDERCOLOR="#808080" CELLSPACING="1" WIDTH="405">
<CAPTION><FONT FACE="ARIAL" SIZE="5">South-American countries and their
   capitals</FONT></CAPTION>
     <TH WIDTH="100">Country</TH><TH WIDTH="100">Capital</TH>
     <TH WIDTH="5"></TH>
     <TH WIDTH="100">Country</TH><TH WIDTH="100">Capital</TH>
     <TR ALIGN="CENTER">
         <TD HEIGHT="47">Argentina</TD><TD>Buenos Aires</TD>
         <TD></TD>
         <TD>Guyana</TD><TD>Georgetown</TD>
     </TR>
     <TR ALIGN="CENTER">
         <TD ROWSPAN="2">Bolivia</TD><TD>Sucre(official)</TD>
         <TD ROWSPAN="2"></TD>
         <TD ROWSPAN="2">Paraguay</TD><TD ROWSPAN="2">Asunción</TD>
     </TR>
     <TR ALIGN="CENTER">
         <TD NOWRAP >La Paz (administrative)</TD>
     </TR>
     <TR ALIGN="CENTER">
         <TD HEIGHT="47">Brazil</TD><TD>Brasília</TD>
         <TD></TD>
         <TD>Peru</TD><TD>Lima</TD>
     </TR>
     <TR ALIGN="CENTER">
         <TD HEIGHT="47">Chile</TD><TD>Santiago</TD>
         <TD></TD>
         <TD>Surinam</TD><TD>Paramaribo</TD>
     </TR>
</TABLE> </CENTER>
</BODY>
</HTML>
```



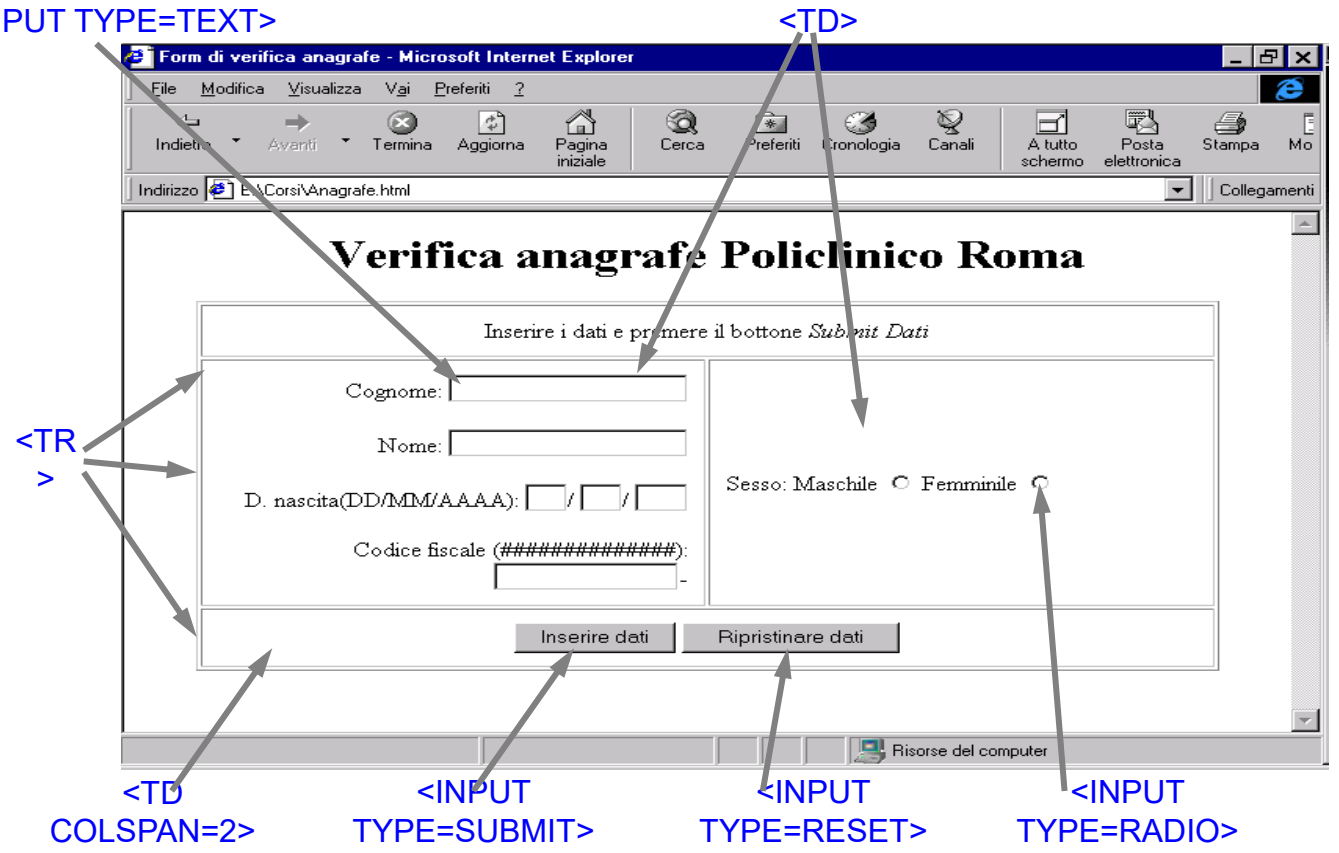
FORM

```
<FORM atributes> form tags </FORM>
<FORM NAME=verify METHOD=post ACTION=http://jf0.pol/servlet/procSQL>
. . . .
</FORM>
```

- A form is composed from INPUT elements.

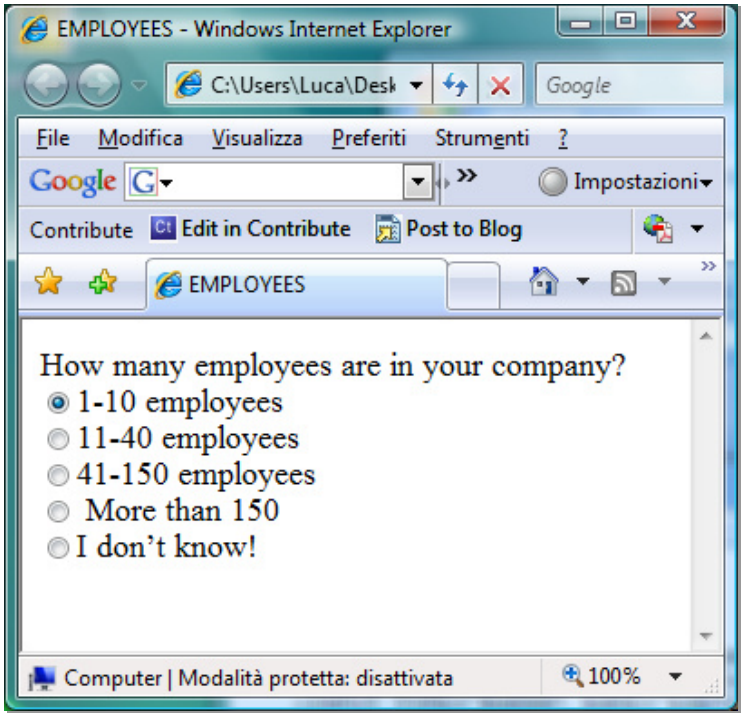
```
<INPUT TYPE=input_type NAME=field_name VALUE=value other-atributes>
input_type = CHECKBOX | FILE | HIDDEN | IMAGE | PASSWORD | RADIO | RESET
   | SUBMIT | TEXT
```

- INPUT values are sent to server



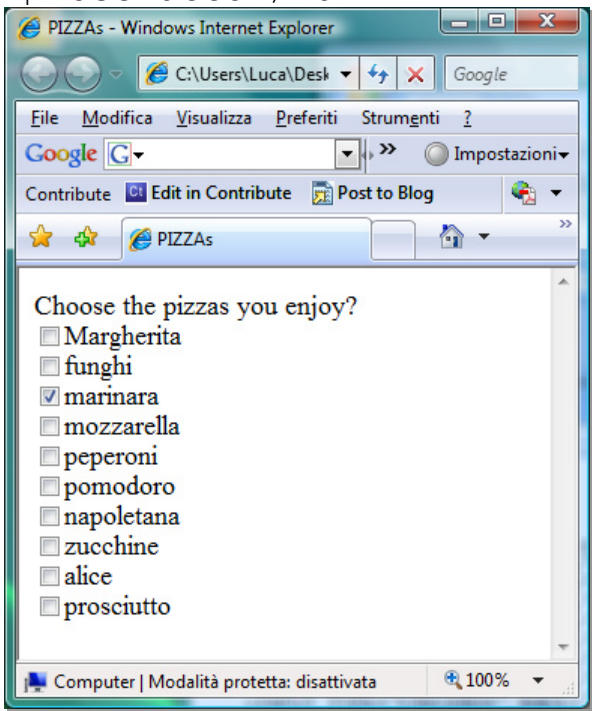
RADIO Buttons

```
<HTML>
<HEAD>
<TITLE>EMPLOYEES</TITLE></HEAD>
<BODY><LEFT>
   <FORM><FONT SIZE="4">How many employees are in your company?<BR>
    <INPUT TYPE="RADIO" NAME="Employees" VALUE="1-10" CHECKED>1-10 employees <BR>
    <INPUT TYPE="RADIO" NAME="Employees" VALUE="11-40">11-40 employees <BR>
    <INPUT TYPE="RADIO" NAME="Employees" VALUE="41-150">41-150 employees <BR>
    <INPUT TYPE="RADIO" NAME="Employees" VALUE="More than 150"> More than 150 <BR>
    <INPUT TYPE="RADIO" NAME="Employees" VALUE="I dont know">I don't know!</FONT>
    </FORM>
</LEFT>
</BODY>
</HTML>
```



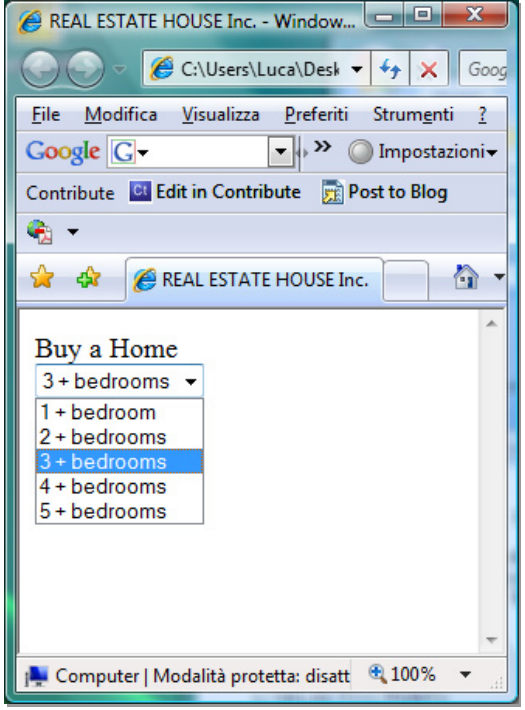
CHECKBOX

```
<HTML>
<HEAD>
<TITLE>PIZZAs</TITLE></HEAD>
<BODY><LEFT>
    <FORM><FONT SIZE="4">Choose the pizzas you enjoy?<BR>
    <INPUT TYPE="CHECKBOX" NAME="Pizza" VALUE="Margherita">Margherita<BR>
    <INPUT TYPE="CHECKBOX" NAME="Pizza" VALUE="Funghi">funghi<BR>
    <INPUT TYPE="CHECKBOX" NAME="Pizza" VALUE="Marinara" CHECKED>marinara<BR>
    <INPUT TYPE="CHECKBOX" NAME="Pizza" VALUE="mozzarella">mozzarella<BR>
    <INPUT TYPE="CHECKBOX" NAME="Pizza" VALUE="peperoni">peperoni<BR>
    <INPUT TYPE="CHECKBOX" NAME="Pizza" VALUE="pomodoro">pomodoro<BR>
    <INPUT TYPE="CHECKBOX" NAME="Pizza" VALUE="napoletana">napoletana<BR>
    <INPUT TYPE="CHECKBOX" NAME="Pizza" VALUE="zucchine">zucchine<BR>
    <INPUT TYPE="CHECKBOX" NAME="Pizza" VALUE="alice">alice<BR>
    <INPUT TYPE="CHECKBOX" NAME="Pizza" VALUE="prosciuto">prosciutto</FONT>
    </FORM>
</LEFT>
</BODY>
</HTML>
```

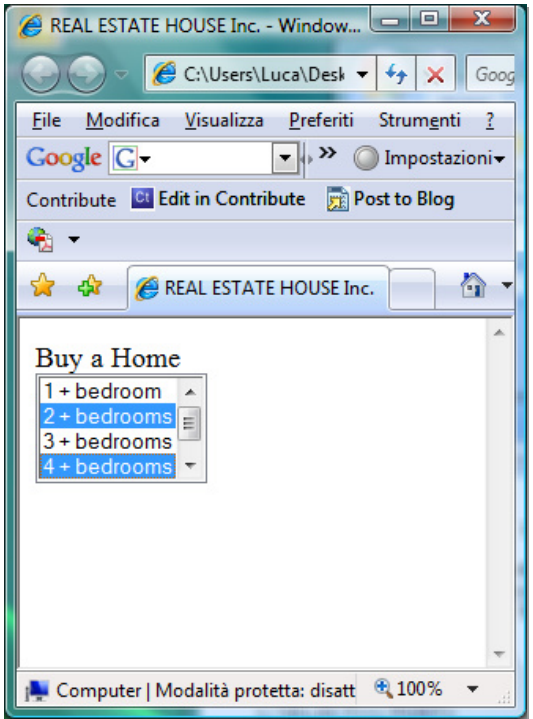


SELECT

```
<HTML>
<HEAD>
<TITLE>REAL ESTATE HOUSE Inc.</TITLE></HEAD>
<BODY><LEFT>
    <FORM><FONT SIZE="4">Buy a Home<BR>
    <SELECT NAME="TipoCasa" SIZE="1">
    <OPTION SELECTED>1 + bedroom
    <OPTION>2 + bedrooms
    <OPTION>3 + bedrooms
    <OPTION>4 + bedrooms
    <OPTION>5 + bedrooms
    </SELECT></FONT>
    </FORM>
</LEFT></BODY>
</HTML>
```




```
<HTML>
<HEAD>
<TITLE>REAL ESTATE HOUSE Inc.</TITLE></HEAD>
<BODY><LEFT>
    <FORM><FONT SIZE="4">Buy a Home<BR>
    <SELECT NAME="TipoCasa" MULTIPLE SIZE="4">
    <OPTION SELECTED>1 + bedroom
    <OPTION>2 + bedrooms
    <OPTION>3 + bedrooms
    <OPTION>4 + bedrooms
    <OPTION>5 + bedrooms
    </SELECT></FONT>
    </FORM>
</LEFT></BODY>
</HTML>
```



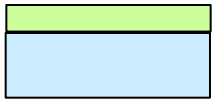
FRAMESET

- Frames can divide the screen into separate windows. Each of these windows can contain an HTML document.
- Frameset= A file that specifies how the screen is divided into frames.
  If you want to make a homepage that uses frames you should:
  - make an HTML document with the frameset .
  - make the normal HTML documents that should be loaded into each of these frames.
- When a frameset page is loaded, the browser automatically loads each of the pages associated with the frames.

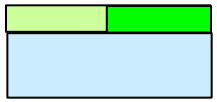
```
 <html>
<head>
<title>My Frames Page</title>
</head>
<frameset cols="120,*‟ frameborder="0" border="0" framespacing="0" >
<frame src="menupage.htm" name="menu‟ noresize scrolling=no >
<frameset rows="*,50">
<frame src="welcomepage.htm" name="main‟ scrolling=auto >
<frame src="bottombanner.htm" name="bottom">
</frameset>
</frameset>
</html>
```

Frameset - Exemples

```
<frameset rows="16%,84%">
<frame src="top.htm" name="top">
<frame src="bottom.htm" name="bottom">
</frameset>
```



```
<frameset rows="16%,84%">
<frameset cols="50%,50%">
<frame src="tl.htm" name="tl">
<frame src="tr.htm" name="tr">
</frameset>
<frame src="bottom.htm" name="bottom">
</frameset>
```



```
<frameset rows="50%,50%" cols="50%,50%">
<frame src="topleft.htm" name="topleft">
<frame src="topright.htm" name="topright">
<frame src="botleft.htm" name="botleft">
<frameset rows="50%,50%">
<frameset cols="50%,50%">
<frame src="brtl.htm" name="brtl">
<frame src="brtr.htm" name="brtr">
</frameset>
<frame src="botrbot.htm" name="botrbot">
</frameset>
</frameset>
```



```
<frameset rows="30%,*" cols="120,*">
<frame src="topleft.htm" name="topleft">
<frame src="topright.htm" name="topright">
<frame src="botleft.htm" name="botleft">
<frame src="botright.htm" name="botright">
</frameset>
```



JavaScript Programming

JavaScript

Characteristics

- Source: a text interpreted by the browser
- Case-sensitive
- Object-based (there are not classes and inheritance)
- Code integrated with and contained in HTML
- Variables, but without declarations
- Dynamic binding (verification to the run-Time of references to objects)
- Event-driven .

The Javascript client-side

- The browser has a Javascript interpreter.
- The interpreter is integrated with DOM (Document Object Model)

The Javascript server-side

- An alternative to CGI
- The server executes the scripts found in the document and sends the result- document to the client.
- The HTML files that contain Javascript code in the server-side are precompiled.
- Dynamic creation of the Web pages
- Use of the same objects used by the Web server Web: Rows, Database, Request, Client.

Structuring the Javascript Code

- Statements, statement blocks, comments.
- A statement can use variabiles, constants (strings, numbers, expressions).
- A code line contains one or more statements. In the later case the statements are separated by “;” :

```
unBird = “Robin"; var now = new Date();
```

- Statement block = Javascript statements enclosed by { }.
- They are used in conditional statements or functions.

```
function convert(inches) {
      feet = inches / 12;
      miles = feet / 5280;
      nauticalMiles = feet / 6080;
      cm = inches * 2,54;
      meters = inches / 39,37;
  }
```

With Javascript we can write directly in the HTML page:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<!- This is a test page -->
<HEAD>
<TITLE>Write two rows in Javascript</TITLE>
<SCRIPT LANGUAGE = "JavaScript">
 document.write("<CENTER><FONT COLOR='magenta'><H1>Welcome to " );
 document.writeln(“Javascript Programming!</H1></FONT></CENTER>" );
</SCRIPT>
</HEAD><BODY>
<H1 ALIGN="CENTER">Programming in JavaScript!</H1>
</HTML>
```

Window object

In JavaScript, we use objects by giving each one a distinctive name. Some objects, such as the `window`, have a built-in name.

The `window` object has a property called `status` that describes the text that is currently displayed on the status bar of the user's browser.

`window.status` or `self.status`

refer to the contents of the status bar, which we can either read from or write to.

Ex:

```
window.status = "Click here to download the Mac version of Java."
window.status = "The old status was " + window.defaultStatus
```

The `window` object has also associated behavior, built-in functions:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<!- This is a test page -->
<HEAD>
<TITLE>Display a DialogBox</TITLE>
<SCRIPT LANGUAGE = "JavaScript">
  window.alert ("Welcome to JavaScript programming!" );
</SCRIPT>
</HEAD>
<BODY>
<P>Click on Refresh or Reload for executing the script again!</P>
</BODY>
</HTML>
```

Javascript Characteristics

Ignores spaces, tab and newline

Comments: `// e /* . . . */`

Letterals: `12  1.2 “hello world” ‘hello’ true null`

Identifiers: `i  v13 $str variable_name`

Java Script keywords: `do else if new return switch while var ...`

Java keywords : `char double final implements native static ...`

Numbers: `3 026 0xff 3.14 .3333 6.02e+24 Number.MAX_VALUE`

Special numbers: `Number.MIN_VALUE Number.NaN Number.POSITIVE_INFINITY`

Strings: `‘test’ “3.14” ‘nome=“forma”’ \b \t \n \’ \” \\ \XXX \uXXXX`

Functions:

```
function square(x) {return x*x;};
var square = new Function(“x”, “return x*x;”);
var square = function(x) {return x*x;};
```

Objects:

```
image.width  image[“width”]  document.unform.button
var o = new Object(); o.x = 2.3; o.y = -1.2;
var o = {x : 2.3; y : -1.2};
```

Array:

```
document.images[1].width
var a=new Array();a[0]=1.2;a[1]=“JS”;a[2]=true;a[3]={x:1};
var a = [1.2,“Java”, true, {x:1}];
```

Date:

```
adesso = new Date(); natale = new Date(99,11,25);
natale.setYear(natale.getYear()+1);adesso.toLocaleString()
```

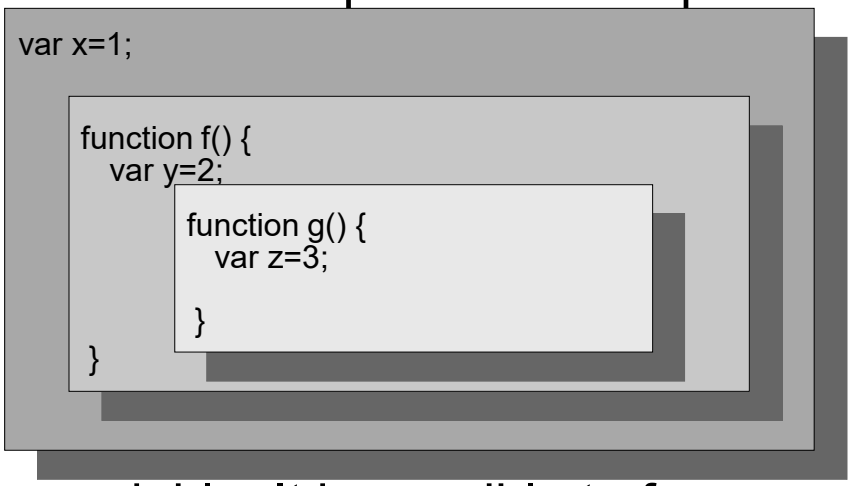
Wrapper-objects:

```
s=“This is a test”;
s=new String (“This is a test”); msg = s + ‘!’;
```

Variables in JavaScript

- We use variables in JavaScript for storing and retrieve values by using their names.
- Variable visibility domain:
  - global
  - local, inside functions.
- Each program is represented by a global object having the global variables as a property.
- For each function an execution context is created containing the function local variables.
- Variables and object properties are identical concepts in Javascript

Variables' static context ===→



- Variable declarations:

```
var i;
var sum, diff;
var j=0, k=1;
for(var i=0; i<10; i++) ...
```

- JavaScript variables are typeless, their current type is provided by the type of the value actually stored in the variable. It is possible to force automatic conversion (numbers->strings) or use explicit conversion (`parseInt()` or `parseFloat()`)

Sum of two numbers

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<!-- This is a test page -->
<HEAD>
<TITLE>Sum</TITLE>

<SCRIPT LANGUAGE = "JavaScript">
  var first, second, nr1, nr2, sum;
  first = window.prompt("Input the first number", "0");
  second = window.prompt("Input the second number", "0");
  nr1 = parseInt(first);
  nr2 = parseInt(second);
  sum = nr1 + nr2;
  document.writeln("<H1>Sum is " + sum + "</H1");
</SCRIPT>

</HEAD>
<BODY>

<P>Click on Refresh or Reload for executing the script again!</P>
</BODY>
</HTML>
```

Data Types

Strings

- In JavaScript a string is an object with special properties, as length:

```
"Hello, world!"
'"Come back!" said the boss.'
"42"
```

- The string "" has the length 0.

Numeric types

- Javascript supports only integer numbaer.
- Esistono inoltre alcuni valori numerici particolari:
  - **NaN** is not a number
  - Positive Infinito
  - Negative Infinite
  - Positive 0
  - Negative 0

Boolean Type

- **true** and **false**

Not defined type

- It is a value assigned to a variable at the declaration time .

Null

- **null** is a meaningless value .

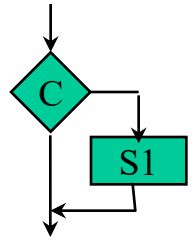
Operatori and their precedence

. [ ] ( )	Field access, array indexing, and function calls
++ -- - ~ ! typeof new void delete	Unary operators, return data type, object creation, undefined values
* / %	Multiplication, division, modulo division
+ - +	Addition, subtraction, string concatenation
<< >> >>>	Bit shifting
< <= > >=	Less than, less than or equal to, greater than, greater than or equal to
== != === !==	Equality, inequality, identity, nonidentity
&	Bitwise AND
^	Bitwise XOR
|	Bitwise OR
&&	Logical AND
||	Logical OR
?:	Conditional
= *op*=	Assignment, assignment with operation
,	Multiple evaluation

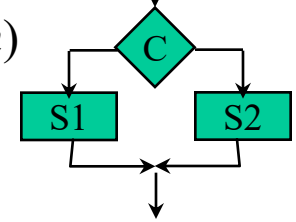
Controlling Program Flow

Conditional Stm. (if-else)

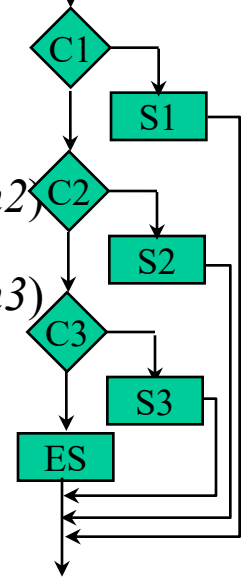
```
if (Condition)
  Statement1
```




```
if (Condition)
  Statement1
else
  Statement2
```

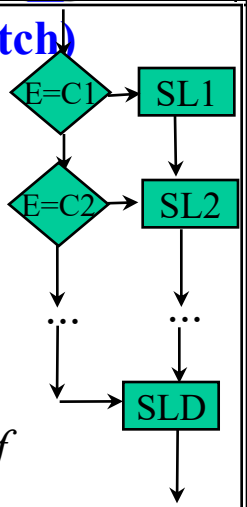



```
if (Condition1)
  Statement1
else if (Condition2)
  Statement2
else if (Condition3)
  Statement3
else
  ElseStatement
```



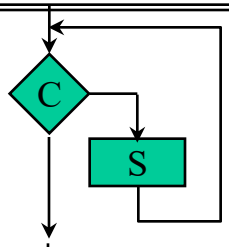
Selection Stm. (switch)

```
switch (E) {
  case C1:
    StatementList1
  case C2:
    StatementList2
  .......
  default:
    StatementListDef
  }
```



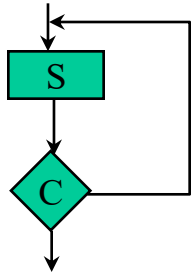
Loop (while)

```
while (Condition)
  Statement
```



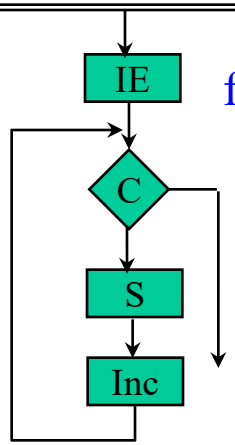
Loop (do-while)

```
do
  Statement
while (Condition)
```

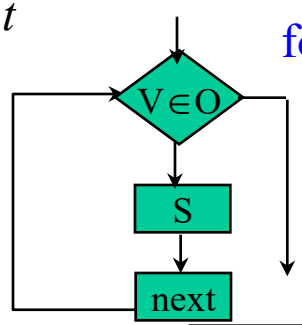


Loop (for)

```
for (InitExp; Condition; Increm)
  Statement
```




```
for (variable in object)
  Statement
```



Current Iteration Interruption

```
continue;
```

Loop/switch interruption

```
break;
break label;
```

Statement return

```
return expression;
```

Conditional Statement

```
// The smash() function is defined elsewhere in the code.
if (newShip) smash(champagneBottle,bow); // Boolean test of whether
  newShip is true.
// In this example, the test fails unless both conditions are true.
if(rind.color=="deep yellow"&&rind.texture=="large and small wrinkles‟){
  theResponse = ("Is it a Crenshaw melon? <br> ");
}
// In this example, the test succeeds if either condition is true.
var theReaction = "";
if ((lbsWeight > 15) || (lbsWeight > 45)) {
  theReaction = ("Oh, what a cute kitty! <br>");
 } else
  theReaction = ("That's one huge cat you've got there! <br>");
```

Conditional Operator

```
var hours = "";
// Code specifying that hours contains either the contents of
// theHour, or theHour - 12.
hours += (theHour >= 12) ? " PM" : " AM";
```

Loops

For Loop

```
var howMany = 11;
var sum = new Array(howMany); // Creates an array sum with 11 members,0 through 10.
var theSum = 0;
sum[0] = 0;
for (var icount = 1; icount < howMany; icount++) {    // Counts from 1 to 10.
   theSum += icount;
   sum[icount] = theSum;
}

var newSum = 0;
for(var icount = 1; icount > howMany; icount++) {
   newSum += icount;                          // This line is not executed.
}

var sum = 0;
for(var icount = 1; icount > 0; icount++) {     // Infinite loop.
   sum += icount;
}

for(j in house){    //house is an object with several properties.
     // Various JScript code statements.
}
```

While Loop

```
var theMoments = "";
var theCount = 42; // Initialize the counter variable.
while (theCount >= 1) { if (theCount > 1) {
  theMoments = "Only " + theCount + " moments left!";
}
else {
  theMoments = "Only one moment left!";
 }
theCount--; // Update the counter variable.
}
theMoments = "BLASTOFF!";
```

Javascript Functions

- JavaScript Function = enclosed group of operations, under a global name .
- You can write out a set of statements, name it, and then execute the entire set any time you want to, just by calling it and passing to it any information it needs.
- You pass information to a function by enclosing the information in parentheses after the name of the function. Pieces of information that are being passed to a function are called arguments or parameters. Some functions don't take any arguments at all; some functions take one argument; some take several. There are even functions for which the number of arguments depends on how you are using the function.
- JavaScript supports two kinds of functions:
  - those that are built into the language, and
  - those you create yourself.
- **Special Built-in Functions**
- eval() - evaluates any valid mathematical expression that is presented in string form. The **eval()** function takes one argument, the expression to be evaluated.

```
var anExpression = "6 * 9 % 7";
var total = eval(anExpression); // Assigns the value 5 to the variable total.
var yetAnotherExpression = "6 * (9 % 7)";
total = eval(yetAnotherExpression) // Assigns the value 12 to the variable total.
var totality = eval("...surrounded by acres of clams."); // Generates an error.
```

- parseInt() – receives a string that represents an integer and computes the represented number.

User Functions

```
function print(message) {
  document.write(message, "<BR>");
}
function distance(x, y, z) {
  var dx = x2 - x1;
  var dy = y2 - y1;
  return Math.sqrt(dx*dx + dy*dy);
}
function factorial(x) {
  if (x <= 1) return;
  return x*factorial(x-1);
}
```

Function Calls

```
print("Hello, " + nome);
print("Welcome in our web site!");
total_dist = distance(3, 4.6, 8) + distance(3.1, 8.3);
print("A probability of: " + factorial(32)/factorial(63);
```

Nested Functions

```
function ipo(a, b) {
  function sq(x) { return x*x; }
  return Math.sqrt(sq(a) + sq(b));
}
```

Constructors of Functions

```
var f = new Function("x", "y", "return x*y;");
```

Javascript Objects

- Creation of objects

```
var circle ={x:0, y:0, radius:2}
var person =    {
                name: "John Smith",
                age: 34,
                married: true,
                email: "pippo@datamat.com"
                };
```

Javascript supports three kinds of objects:
1. intrinsic objects,
2. objects you create, and
3. browser objects

- Accessing properties

```
var book = new Object();
book.title = "JavaScript";
book.chap1 = new Object();
book.chap1.title = "Introduction";
book.chap1.pages = 25;
book.chap2 = { title:"Basic components", pages:10 };
```

- Constructors

```
function Rect_area() { return this.width * this.height; }
function Rectangle(w,h){this.width=w; this.height=h;
  this.area=Rect_area }
var r1 = new Rectangle (2, 4);
var r2 = new Rectangle (7.5, 10);
var a = r1.area();
```

Using Arrays

- Arrays in Javascript are *sparse.*

```
a[0]=88;
a[1]=5;
a[2]=10;
a[50]=100;
```

you can create element 50 without worrying about elements 3 through 49. The array's length variable is set to 51, rather than to 4.

- There are two main ways to address the members of an array:
  - *indexing* an array. We can address arrays by using brackets:

```
theListing = addressBook[entryNum];
theFirstLine = theListing[1];
```

Subscript numbering in Javascript begins with 0.

  - *associative* addressing of the array . We can address an array that contains properties that are numbered, and then generate the numbers in a loop. The following example generates two arrays, one for the name and one for the address, from a listing in `addressBook`.

```
theListing = addressBook[entryNum];
for (i = 1; i < 4; i++) {
   theName[i] = theListing["Name" + i];
   theAddress[i] = theListing["Address" + i];
}
```

Sometimes the particular property may not exist until run time, or there may be no way to know which one it will be in advance.

Objects as Arrays

- You can create arrays in Javascript with the operator `new` as for objects :

```
var aTable = new Array(17);
var myArray = new Array("Athens", "Belgrade", "Cairo");
var MyArray = {"a":"Athens", "b":"Belgrade", "c":"Cairo" }; //associative array
```

- In Javascript, objects and arrays are handled identically, because objects in Javascript are associative arrays.
- You can refer to any of the members of an object (its properties and methods) either by name (using the period) or by its array subscript index.
- Thus, a property can be referred to in several ways. All of the following statements are equivalent:

```
theWidth = room.width;
theWidth = room[3]; // [3] is the "width" index.
theWidth = room["width"];
```

- Don't use the dot (.) convention with index numbers. The following statement generates an error.

```
theWidth = room.3;
```

- Objects can have other objects as properties . In this case the naming convention extends in a straightforward way.

```
var init4 = toDoToday.shoppingList[3].substring(0,1);
// shoppingList, an array, is a property of toDoToday.
```

- Multidimensional arrays:

```
var multTable = new Array(17);
var aRow = new Array(17);
aRow[0]=34;
aRow[1]="34";
multTable[5] = aRow;
```

Objects, functions, methods

- function – function constructor
- a functione can be stored in a variable, passed to an other variable etc.

```
a[0]=function(x) { return x*x; };
a.sort(function(x) { return x*x; });
var ten=(function(x) { return x*x; })(10);
```

- The functions can be processed as data

```
a = square(4);    // a contains 16
b = square;       // b rifers the same function as square
c = b(5);         // c contains 25

o = new Object();// an object
o.square = new Function("x", "return x*x"); // stores a function as a method
y = o.square(16);

a = new Array(10);
a[0] = function(x) { return x*x; } // use of functions as letterals
a[1] = 20;
a[2] = a[0](a[1]);

function add(x, y) { return x+y; }
function mul(x, y) { return x*y; }
function operate(op, opd1, opd2) { return op(opd1, opd2); }
var i = operate(add, operate(add, 2, 3), operate(mul, 4, 5); };
```

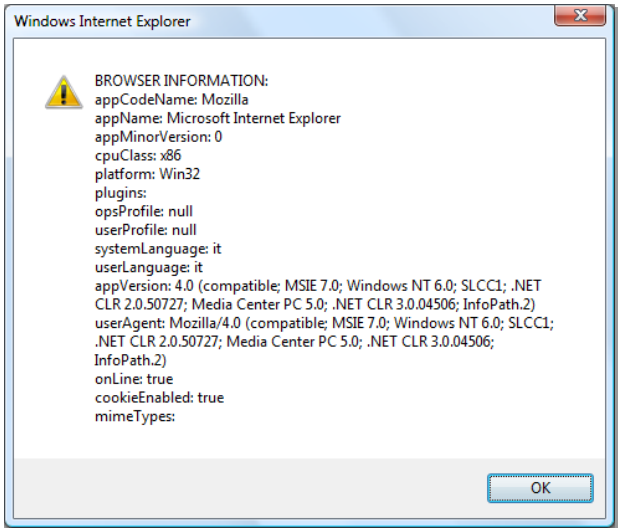
Navigator. Detecting your viewer's browser

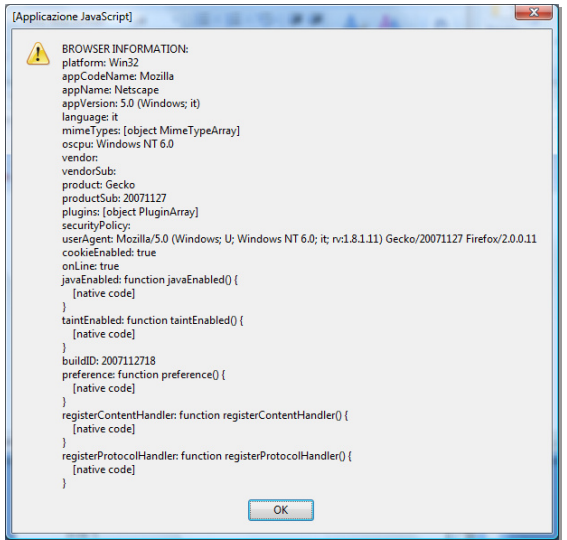
Navigator = an object containing information on your browser.

Navigator properties:

appName	the browser name
appVersion	the browser version number
userAgent	represents the browser
appCodeName	the coded name of the browser
platform	the hardware platform
language	the language version supported

```
var browser = "BROWSER INFORMATION:\n";
for (var nomeprop in navigator) {
  browser += nomeprop + ": " + navigator[nomeprop] + "\n";
}
alert(browser);
```



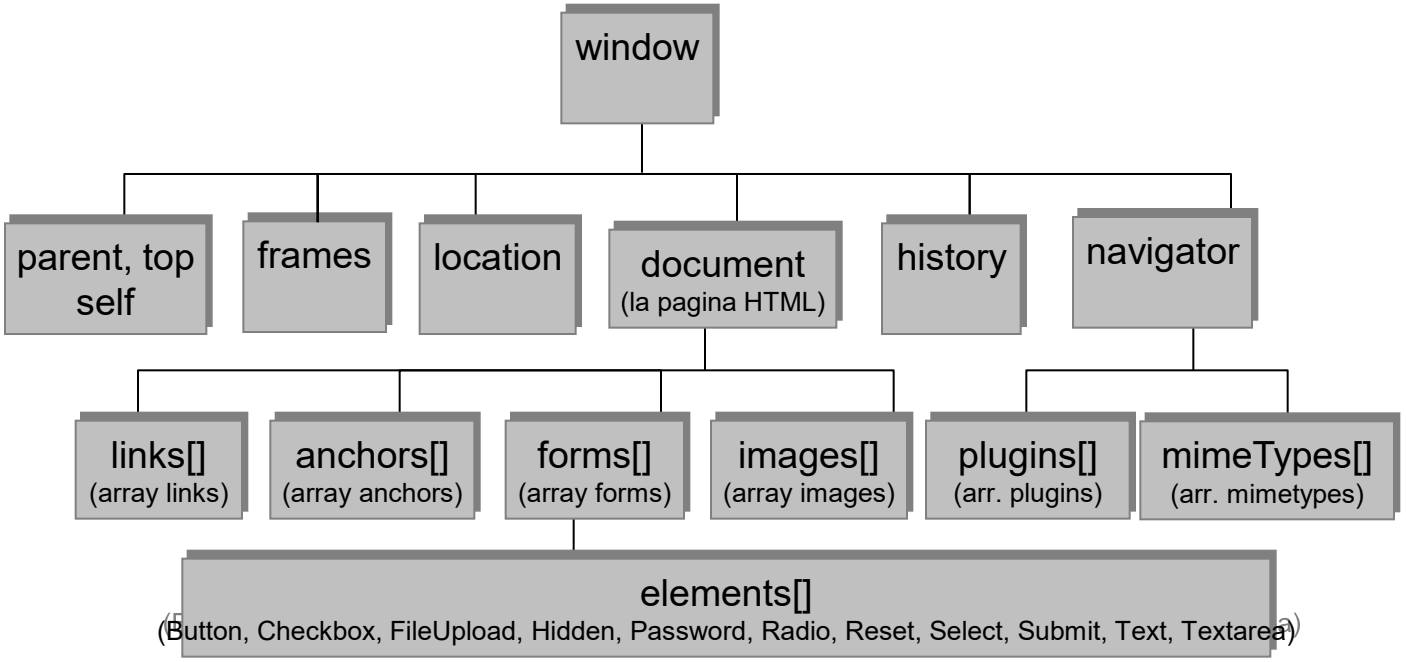


Browser Objects

The browser objects:

- are created when the page is downloaded. They are : `window`, `frames` `location` , `history`, `document` etc.
- are included in a hierarchy based on relations as: `top`, `parent`, `self` .

- In the same hierarchy are included the Javascript objects: *forms*[*i*], *links*[*i*], *image*[*i*]
- The browser's algorithm builds its hierarchy in a top-down manner.
- The Javascript interpreter 's algorithm builds its object hierarchy also in top-down manner starting with `document`.



Window Object

The Window object is the top level object in the JavaScript hierarchy.
The Window object represents a browser window.
A Window object is created automatically with every instance of a `<body>` or `<frameset>` tag.

Method	Description
alert()	Displays an alert box with a message and an OK button
blur()	Removes focus from the current window
clearInterval()	Cancels a timeout set with setInterval()
clearTimeout()	Cancels a timeout set with setTimeout()
close()	Closes the current window
confirm()	Displays a dialog box with a message and an OK and a Cancel button
createPopup()	Creates a pop-up window
focus()	Sets focus to the current window
moveBy()	Moves a window relative to its current position
moveTo()	Moves a window to the specified position
open()	Opens a new browser window
print()	Prints the contents of the current window
prompt()	Displays a dialog box that prompts the user for input
resizeBy()	Resizes a window by the specified pixels
resizeTo()	Resizes a window to the specified width and height
scrollBy()	Scrolls the content by the specified number of pixels
scrollTo()	Scrolls the content to the specified coordinates
setInterval()	Evaluates an expression at specified intervals
setTimeout()	Evaluates an expression after a specified number of milliseconds

Location Object

- The Location object is actually a JavaScript object, not an HTML DOM object.
- The Location object is automatically created by the JavaScript runtime engine and contains information about the current URL. Example:

Send a user to a new location

```
<html>
<head>
<script type="text/javascript">
function currLocation(){
alert(window.location);
}
function newLocation(){
window.location="http://www.w3schools.com";
}
</script>
</head>
<body>
<input type="button" onclick="currLocation()" value="Show current URL">
<input type="button" onclick="newLocation()" value="Change URL">
</body>
</html>
```

- The Location object is part of the Window object and is accessed through the `window.location` property.

History Object

- The History object is actually a JavaScript object, not an HTML DOM object.
- The History object is automatically created by the JavaScript runtime engine and consists of an array of URLs. These URLs are the URLs the user has visited within a browser window.
- The History object is part of the Window object and is accessed through the `window.history` property.

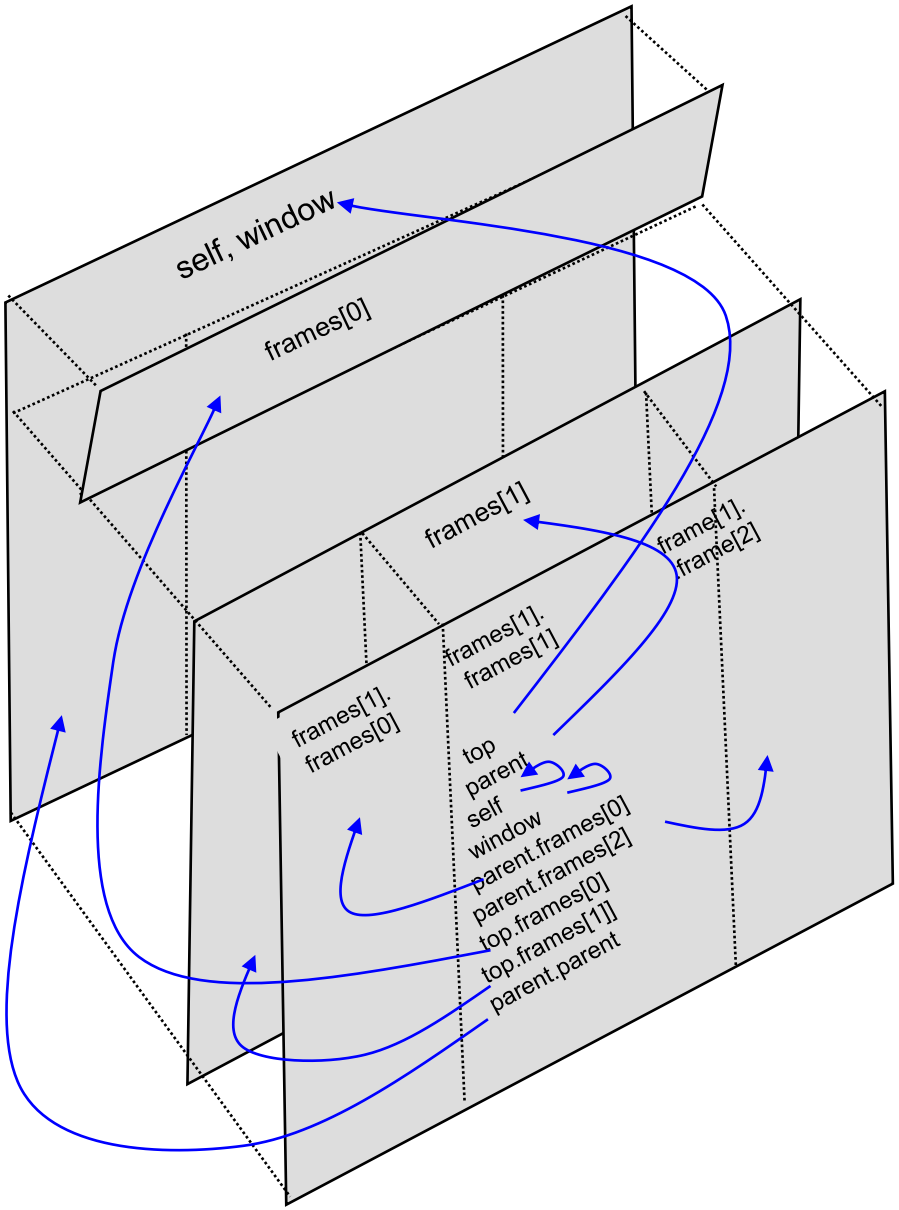
Method	Description
`back()`	Loads the previous URL in the history list
`forward()`	Loads the next URL in the history list
`go()`	Loads a specific page in the history list

Document Object

- The Document object represents the entire HTML document and can be used to access all elements in a page.
- The Document object is part of the Window object and is accessed through the window.document property.

Method	Description
close()	Closes an output stream opened with the document.open() method, and displays the collected data
getElementById()	Returns a reference to the first object with the specified id
getElementsByName()	Returns a collection of objects with the specified name
getElementsByTagName()	Returns a collection of objects with the specified tagname
open()	Opens a stream to collect the output from any document.write() or document.writeln() methods
write()	Writes HTML expressions or JavaScript code to a document
writeln()	Identical to the write() method, with the addition of writing a new line character after each expression

Inter-frames relationships



References in Javascript

Two types of references to objects:

1. Full paths in the hierarchy:

```
document.links[0]
document.forms[0].elements[0]
document.forms[0].elements[1].value
```

2. Object names defined in the HTML page:

```
<FORM NAME=verif METHOD=post ACTION=http://jf0.pol/servlet/procSQL>
  . . . .
  <P>Last name:<INPUT TYPE=TEXT NAME=lastname SIZE=20 MAXLENGTH=40></P>
. . . . document.verif.lastname.value . . . .
```

`window` object

Other windows than the pages's one can be also managed in JavaScript: *location* object.

```
<HEAD>
<SCRIPT language="JavaScript">
<!--
 function getgoing()
  {    top.location=http://someplace.com;  }
   if (top.frames.length > 0) {
     alert("The frames will be broken in ten seconds.");
     setTimeout('getgoing()',10000);
     }
//-->
</SCRIPT>
</HEAD>
```

Displaying Information In the Browser

- The most common way to display information are `write()` and `writeln()` methods of the document object.

```
document.write("Pi is approximately equal to " + Math.PI);
document.write( );
```

- When you call the `write()` method, it opens and clears the document if the document is not in the process of being opened and parsed when the `write()` method is called.

```
<HTML> <HEAD>
<SCRIPT LANGUAGE="JScript">
 function singOut() {
 var theMoment = new Date();
 var theHour = theMoment.getHours();
 var theMinute = theMoment.getMinutes();
 var theDisplacement = (theMoment.getTimezoneOffset() / 60);
 theHour -= theDisplacement;
 if (theHour > 23) {
 theHour -= 24
 }
 document.write(theHour+" hours,"+theMinute+" minutes,Coordinated Universal Time.");
 window.setTimeout("singOut();", 60000);
 }
 </SCRIPT>
 </HEAD>
 <BODY>
 <SCRIPT>
 singOut();
 </SCRIPT>
 </BODY> </HTML>
```

a script intends to display the time once a minute, but fails to do so after the first time because it clears itself in the process.

It should be better to use window.alert()

```
window.alert(theHour + " hours, " + theMinute + "
minutes, Coordinated Universal Time.");
window.setTimeout("singOut();", 60000);
```

Exemples (I)

Write text to the output

```
<html>
<body>
<script type="text/javascript">
document.write("Hello World!");
</script>
</body>
</html>
```

Return the URL of a document

```
<html>
<body>
The URL of this document is:
<script type="text/javascript">
document.write(document.URL);
</script>
</body>
</html>
```

Open a new document, specify MIME type and add some text

```
<html>
<head>
<script type="text/javascript">
function createNewDoc(){
  var newDoc=document.open("text/html","replace");
  var txt="<html><body>Learning about the DOM is FUN!</body></html>";
  newDoc.write(txt);
  newDoc.close();
}
</script>
</head>
<body>
<input type="button" value="Open and write to a new document"
   onclick="createNewDoc()">
</body>
</html>
```

Exemples (II)

Access an item in a document

```
<html>
<body>
<form id="Form1" name="Form1">
Your name: <input type="text">
</form>
<form id="Form2" name="Form2">
Your car: <input type="text">
</form>
<p>
To access an item in a collection you can either use the number or the name of the
   item:
</p>
<script type="text/javascript">
document.write("<p>The first form's name is: " + document.forms[0].name + "</p>");
document.write("<p>The first form's name is: " +
   document.getElementById("Form1").name + "</p>");
</script>
</body>
</html>
```

Using Message Boxes

- Use *alert*, *confirm*, and *prompt* message boxes to obtain input from your user. The boxes are methods of the interface `window` object.

```
window.alert("Welcome! Press OK to continue.");

var truthBeTold = window.confirm("Click OK to continue. Click Cancel to
  stop.");
if (truthBeTold) {
   window.alert("Welcome to our Web page!");
 } else
   window.alert("Bye for now!");

var theResponse = window.prompt("Welcome?","Enter your name here.");
```

Javascript Events

Event	Description	handler syntax
Blur	Causes the element to lose focus and fires the onblur event.	onBlur
Click	The user clicks a form element or a link	onClick
Change	The user changes the text in a text area	onChange
Error	An error during an image or form download occured	onError
Focus	The user grants the focus on a form component	onFocus
Load	The user downloads a page in the browser	onLoad
MouseOver*	The user moves the mouse over an element	onMouseOver
MouseOut*	The user moves the mouse out of an element	onMouseOut
MouseMove*	The user moves the mouse	onMouseMove
Move	The user moves the window or the frame	onMove
Reset	The user resets the form	onReset
Resize	The user resizes the window or the frame	onResize
Select	The user selects an input field of a form	onSelect
Submit	The user submits a form (he pressed the Submit button)	onSubmit
Unload	The user leaves the current pasge	onUnload

Exemple:

```
<INPUT
  TYPE="checkbox"
  NAME="opts"
  VALUE="ignore-case"
  onClick="ignore_case = this.checked;"
>
```

- For managing events we have to write an "event handler" in Javascript.

Interactions with HTML (I)

```
<HTML>
<HEAD>
<SCRIPT LANGUAGE=JavaScript>
Function window_loading()
{
   var numeroForms = document.forms.length;
   var formIndex;
   for (formIndex = 0; formIndex < numeroForms; formIndex++) {
      alert(document.forms[formIndex].name);
   }
}
</SCRIPT>
</HEAD>
<BODY LANGUAGE=JavaScript onLoad="window_loading()">
<FORM NAME="form1">
<P>We are in form1.</P>
</FORM>
<FORM NAME="form2">
<P>We are in form2.</P>
</FORM>
<FORM NAME="form3">
<P>We are in form3.</P>
</FORM>
</BODY>
</HTML>
```

Event handler

Call of the event handler

Interactions with HTML (II)

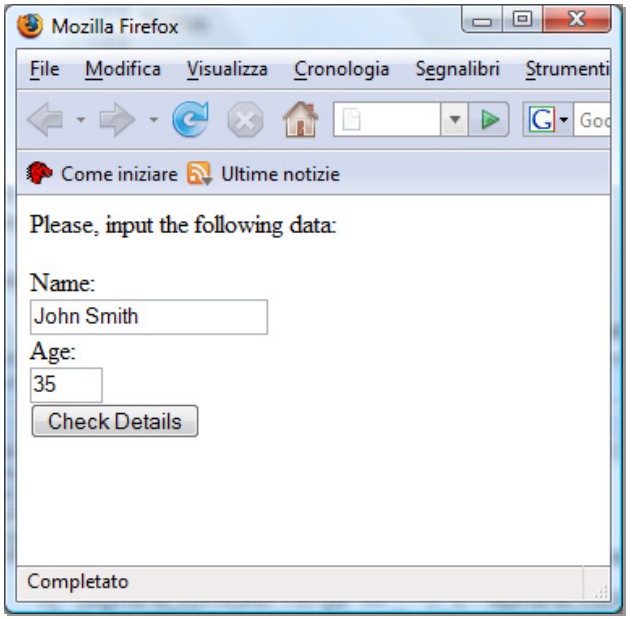
```
<HTML>
<HEAD>
<SCRIPT LANGUAGE=JavaScript>
function myButton_down()
{
    document.form1.myButton.value = “The mouse is pressed."
}
function myButton_up()
{
   document.form1.myButton.value = “The mouse is released."
}
</SCRIPT>
</HEAD>
<BODY>
<FORM NAME=form1>
    <INPUT TYPE='button' NAME='myButton' VALUE='The mouse is released.'
   onMouseUp="myButton_up()" onMouseDown="myButton_down()"
  onMouseOver="status='click here';return true;" onMouseOut="status='';">
</FORM>
</BODY>
</HTML>
```

Event handler

Call of the event handler

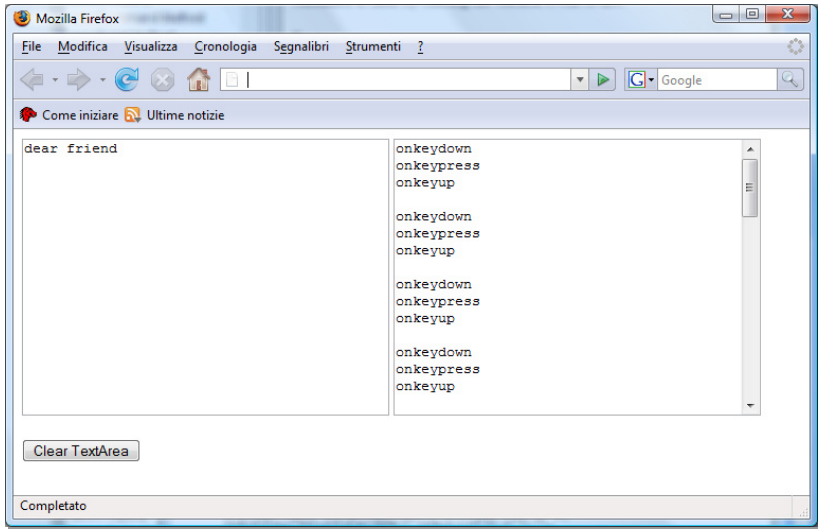
Interactions with HTML (III)

```
<HTML> <HEAD>
<SCRIPT LANGUAGE=JavaScript>
function butCheckForm_onclick() {
   var myForm = document.form1;
   if (mioForm.txtAge.value == "" || mioForm.txtName.value == "") {
      alert("Fill in the form, please.");
      if (myForm.txtName.value == "") {  myForm.txtName.focus(); }
      else {  myForm.txtAge.focus(); }
   }
   else { alert("Thank for filling in, " + mioForm.txtName.value); }
}
function txtAge_onblur() {
   var txtAge = document.form1.txtAge;
   if (isNaN(txtAge.value) == true) {
      alert("Please, input the correct age.");
      txtEta.focus();
      txtEta.select();
   }
}
function txtName_onchange() {  window.status = "Hello, "
                     + document.form1.txtName.value; }
</SCRIPT></HEAD>
<BODY>
<FORM NAME=form1>
   Please, input the following data:
   <P>   Name:   <BR>
   <INPUT TYPE="text" NAME=txtName onchange="txtName_onchange()"> <BR>
   Age:   <BR>
   <INPUT TYPE="text" NAME=txtAge onblur="txtAge_onblur()" SIZE=3 MAXLENGTH=3> <BR>
   <INPUT TYPE="button" VALUE="Check Details" NAME="butCheckForm"
    onclick="butCheckForm_onclick()">
</FORM></BODY></HTML>
```



Interactions with HTML (IV)

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 //EN"> <HTML>
<HEAD>
<SCRIPT LANGUAGE=JavaScript>
function DisplayEvent(eventName){
   var myMessage = window.document.form1.textarea2.value;
   mioMessaggio = myMessage + eventName;
   window.document.form1.textarea2.value = myMessage;
}
</SCRIPT></HEAD>
<BODY>
<FORM NAME=form1 ACTION = "…">
   <TEXTAREA ROWS=15 COLS=40 NAME=textarea1
      onchange="DisplayEvent('onchange\n');"
      onkeydown="DisplayEvent('onkeydown\n');"
      onkeypress="DisplayEvent('onkeypress\n');"
      onkeyup="DisplayEvent('onkeyup\n\n');">
   </TEXTAREA>
   <TEXTAREA ROWS=15 COLS=40 NAME=textarea2>
   </TEXTAREA>
   <BR><BR>
   <INPUT TYPE="button" VALUE="Clear TextArea"
      NAME=button1 onclick="window.document.form1.textarea2.value=''">
</FORM>
</BODY>
</HTML>
```



DHTML Programming

DHTML Components

Dynamic HTML = HTML + a set of components that can be processed, included in a HTML page.

Why DHTML?

- Style change = dynamic modification of the layout of the page content.
- Component position = position of the content blocks in the page, eventually moving them from a [point to an other.
- Event Handling = relate user events with style or position changes.

CSSP (Cascading Style Sheet Positioning) - Microsoft - IIE 4 e Netscape

- A DHTML page is composed from <DIV> blocks.
- A <DIV> block has a name that allows us to handle the whole block.

```
<DIV ID="mailblock">
<A HREF="mailto:axiom@axiom.com">Send us your commands!</A><BR>
<IMG SRC="posta.gif" WIDTH=30 HEIGHT=15 ALT=Mailbox">
</DIV>
```

Characteristics of DHTML pages

Positioning components

- The <DIV> blocks can be positioned absolutely or relatively.
- The <DIV> blocks can be nested.

Father

Son

```
<DIV ID="mailblock_father" ALIGN="right">
 <DIV ID="mailblock_son" STYLE="position:relative; width:auto;
 height:auto left:10px; top:10px; background-color:pink">
     <A HREF="mailto:axiom@axion.com">Send us your commands!</A><BR>
     <IMG SRC="posta.gif" WIDTH=30 HEIGHT=15 ALT=Mailbox">
 </DIV>
</DIV>
```

- The <DIV> blocks are programmed with STYLE properties.

STYLE

The STYLE tag syntax:

```
STYLE="property1:value1; property2:value2; property3:value3; . . ."
```

Property	Values	Description
position	absolute, relative, static	Position in the page
width	NNpx, NN%	Width where the component can expand
height	NNpx, NN%	Height where the component can expand
left	±NNpx	The left-side offset
top	±NNpx	The top-side offset
clip	rect(0px 5px 10px 10px)	Rectangle area where the block is visible
visibility	inherit, visible hidden	Block visibility
z-index	int	Order of overlaying blocks
background-color	green FF8F00	Color of the background block
background-image	green FF8F00	Image of the background block

Style Approaches in DHTML

```
<DIV ID="mailblock" STYLE="position:relative; width:auto;
 height:auto left:10px; top:10px; background-color:pink">
      <A HREF="mailto:axiom@axiom.com">Send us your commands!</A><BR>
      <IMG SRC="mail;.gif" WIDTH=30 HEIGHT=15 ALT="Mailbox">
</DIV>
```

- Netscape Dom Model :

  Each <DIV> block is a Layer object.

```
document.layers["mailblock"]
document.layers["mailblock"].bgColor = "red";
document.layers["mailblock"].left = 0 ;
```

- Microsoft Model:

  Each <DIV> block is a DIV object that owns a STYLE object with STYLE (and not DIV) properties.

```
document.all["mailblock]
document.all["mailblock].style.backgroundColor = "red";
document.all["mailblock].style.left = "0px";
document.all["mailblock].style.pixelLeft = 0;
```

Technics for dynamicity

- Animazione = mages in a block can change the position together with the block according to a given path.
- Drop-down menu = Modifying the clipping areas can be made to see or hide portions of the block according to the choice in the menu.
- Content exchange = more blocks of contents can be overlapped in a given position, but only one is visible. The visibility property can be changed when an event occurs. Or, maintaining the overlapped blocks visible, the clipping areas can be modified with a sure effect that produces the “transition” from a block to other.

CSS - Style Sheet

- With Cascading Style Sheet (CSS) we have a style and layout model for HTML documents. CSS allowed developers to control the style and layout of multiple Web pages all at once. You can define a style for each HTML element and apply it to as many Web pages as you want. To make a global change, simply change the style, and all elements in the Web are updated automatically.
- Style Sheet = Style description that can be used for one or more elements or for a HTML block.
- How a Style Sheet can be used?
  - Define a style that is applied to all the HTML elements of the same type.

```
<STYLE ID="sheet1">
H3 { color:blue; font-family:"Arial"; font-size:large; }
</STYLE>
```

  - Define a style "class" which represents predefined styles applied to an element.

```
<STYLE ID="sheet1">
.redbold { color:red; font-weight:bold; }
</STYLE>
. . . .
<BLOCKQUOTE CLASS=redbold>
```

  - Apply styles dynamically using the STYLE attribute of a tag. This procedure si applied to only the tag where the attribute occurs.

```
<H4 STYLE="color:red; font-weight:bold;">
```

Using the classes of style (I)

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Style class</TITLE>
<STYLE TYPE = "text/css">
   P { font-size: 14pt; margin-left: 2em; margin-right: 2em }
   P.narrow { color: red; margin-left: 5em; margin-right: 5em }
   BODY { font-family: Arial, serif }
</STYLE>
</HEAD>
<BODY>
<P> This is a normal paragraph. This is a normal paragraph. </P>
<P CLASS="narrow"> This is a paragraph con greater margins and of red color.
   This is a paragraph con greater margins and of red color. </P>
<P> This is a normal paragraph. This is a normal paragraph. </P>
<P CLASS="narrow"> This is a paragraph con greater margins and of red color.
   This is a paragraph con greater margins and of red color. </P>
</BODY>
</HTML>
```

Using the classes of style (II)

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Free style class</TITLE>
<STYLE TYPE = "text/css">
   P { font-size: 14pt; margin-left: 2em; margin-right: 2em }
   P.narrow { color: red; margin-left: 5em; margin-right: 5em }
   .hot { color: red; text-decoration: underline }
   BODY { font-family: Arial, serif }
</STYLE>
</HEAD>
<BODY>
<H1 CLASS=hot>This is a title</H1>
<P> This is a normal paragraph. This is a normal paragraph. </P>
<DIV CLASS="hot">
<P CLASS="narrow"> This is a paragraph con greater margins and of red color. This is
   a paragraph con greater margins and of red color. </P>
</DIV>
<P>This is a normal paragraph. <SPAN CLASS="hot">This is a normal paragraph.
   </SPAN></P>
<P CLASS="narrow"> This is a paragraph con greater margins and of red color. This is
   a paragraph con greater margins and of red color. </P>
</BODY>
</HTML>
```

Style Change

```
<P ID="selection" ALIGN="left" STYLE="color:blue; font-weight:normal;">
Select the item. </P>
```

The style can be changed with a simple code JavaScript:

```
document.all["selection"].style.color="red";
document.all["selectione"].style.fontWeight="bold";
```

- A style sheet can contain more definitions called rules:

```
<STYLE ID="sheet1">
   H3 { color:blue; font-family:"Arial"; font-size:large; } // first rule
   .redbold { color:red; font-weight:bold; }  // second rule
</STYLE>
```

- The access to the style can be done with styleSheets[] and then for the rules with rules[]:

```
document.styleSheets[0]
document.styleSheets[0].rules[0]
document.styleSheets[0].rules[0].style.color = "pink";
document.styleSheets[0].rules[0].style.fontFamily = "courier";
```

Event Handling

- The properties of elements in a DHML page can be changed dynamically when an event occurs.
- The events are applied for any element, including images <IMG> and paragraphs <P>.

  Exemples:

```
<A HREF="page.html"
onMouseOver="document.styleSheets[0].rules[0].color='blue'"> Click
here!</A>


<A HREF="page.html" TARGET="mainframe" STYLE="color: blue; font-weight:
normal; font-family: Arial"<BR> onMouseOver="this.style.color='blue';
this.style.fontWeight='normal'"> Daily menu</A>
```

- Nescape acquires mouse events only when they are applied to hyperlinks <A>, areas <AREA> or layers <LAYER>.

Filters

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Filter Flip</TITLE>
<STYLE TYPE = "text/css">
   BODY { background-color:#CCCCCC }
   TABLE { font-size: 3em;
           font-family: Arial, sans-serif; }
           background-color: #CCFFCC;
           border-style: ridge;
           border-collapse: collapse }
   TD     { border-style: groove;
           padding: lex; }
</STYLE>
</HEAD>
<BODY>
<TABLE BORDER=1>
   <TR>
           <TD STYLE="filter: fliph">Testo</TD>
           <TD>Testo</TD>
   </TR>
   <TR>
           <TD STYLE="filter: flipv fliph">Testo</TD>
           <TD STYLE="filter: flipv">Testo</TD>
   </TR>
</TABLE>
</BODY>
</HTML>
```



Filtrer Chroma

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Filter Chroma</TITLE>
<SCRIPT LANGUAGE="JavaScript">
   function changeColor() {
   if (selectedColor.value) {
      chromaImg.filters("chroma").color =
         parseInt(selectedColor.value, 16);
      chromaImg.filters("chroma").enabled = true;
         }
         else
      chromaImg.filters("chroma").enabled = false;
    }
</SCRIPT>
</HEAD>

<BODY>
<H1>Filter Chroma:</H1>
<IMG ID="chromaImg" SRC="travel\flag.jpg"
         STYLE="position: absolute; filter: chroma">
<SELECT ID="selectedColor" onChange="changeColor()">
   <OPTION VALUE= "">None
   <OPTION VALUE= "00FFFF">Cyan
   <OPTION VALUE= "FFFF00">Yellow
   <OPTION VALUE= "FF00FF">Magenta
   <OPTION VALUE= "000000" SELECTED>Black
</SELECT>
</BODY>
</HTML>
```

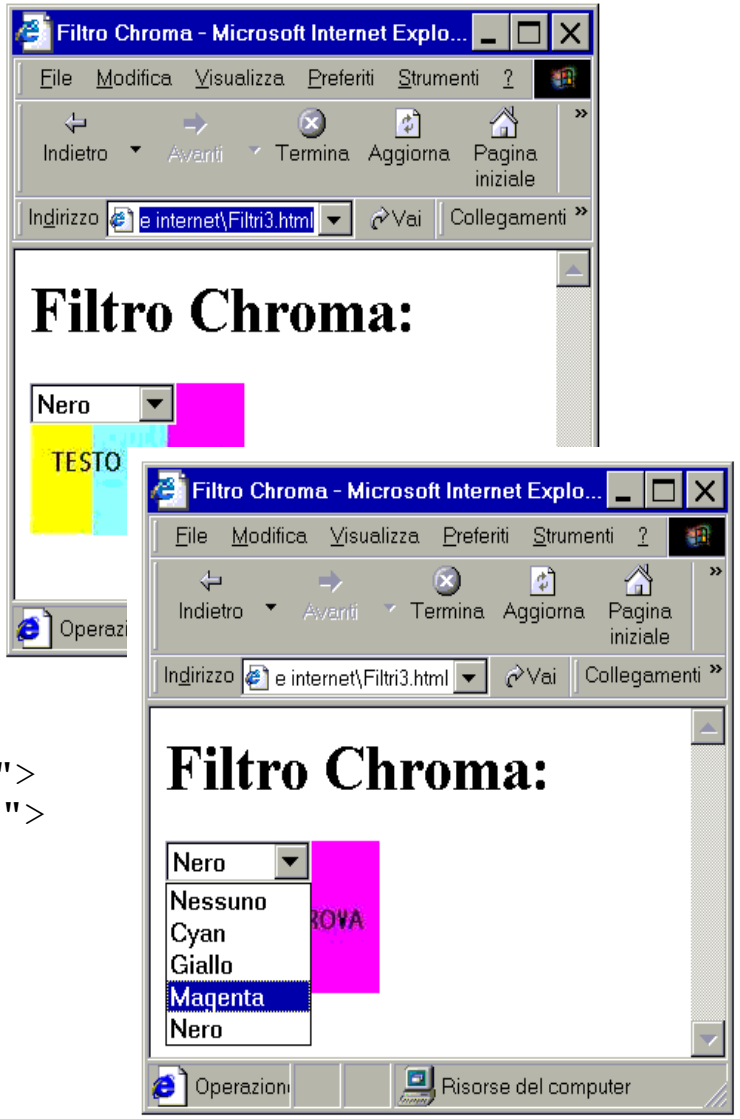
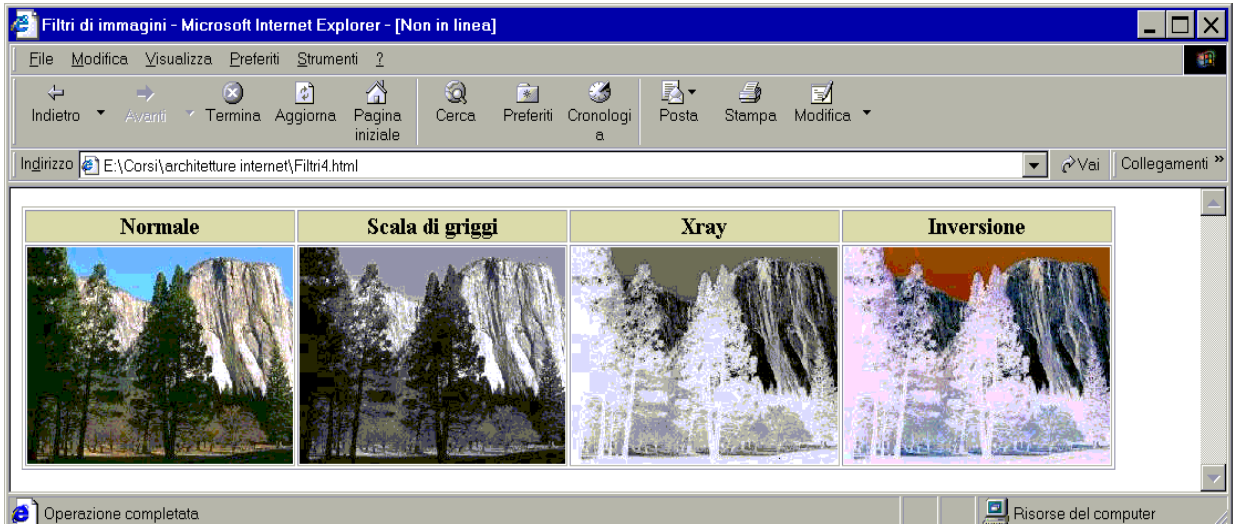


Image Filters

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML><HEAD>
<TITLE>Image Filters</TITLE>
<STYLE TYPE = "text/css">
   .cap { font-weight:bold;
          background-color: #DDDDAA;
          text-align: center;}
</STYLE></HEAD>
```




```
<BODY>
<TABLE BORDER=1>
   <TR CLASS= "cap">
         <TD>Normal</TD>
         <TD>Gray Scale</TD>
         <TD>Xray</TD>
         <TD>Inversion</TD>
   </TR>
   <TR>
         <TD><IMG SRC= "mountain.jpg"></TD>
         <TD><IMG SRC= "mountain.jpg" STYLE= "filter: gray"></TD>
         <TD><IMG SRC= "mountain.jpg" STYLE= "filter: xray"></TD>
         <TD><IMG SRC= "mountain.jpg" STYLE= "filter: invert"></TD>
   </TR>
</TABLE>
</BODY></HTML>
```

Server-Side Programming with Servlets

- Introduction to servlets
- Reading explicit data from clients (form data)
- Reading implicit data from clients(request headers)
- Risults generation
- Sending explicit data to clients (HTML)
- Sending implicit data to clients



Introduction to servlets

- Benefits of servlets
- Using servlets in web applications
- Basic structure of servlets
- A servlet that generates text
- A servlet that generates a HTML page
- Servlets and packages
- Auxiliary tools for building HTML pages
- Servlet life cycle

Advantages and Disadvantages of Java Servlets

Advantages of servlets over "traditional" CGI

Efficient: Each request is run in a separate thread, so servlet request processing is significantly faster than traditional CGI processing.

Servlets are scalable. Many more requests can be executed because the Web container uses a thread rather than an operating system process, which is a limited system resource.

Convenient: Lots of high-level utilities

Powerful: Sharing data, pooling, persistence

Portable: Run on virtually all operating systems and servers. Servlets are platform independent, because they are written in the Java programming language.

Inexpensive: There are plenty of free and low-cost servers.

Servlets have access to logging capabilities. Most CGI programs do not.

Secure: The Web container provides additional services to the servlets, such as error handling and security. No shell escapes, no buffer overflows.

The main disadvantage:

The lack of separation of business logic and presentation logic which makes them difficult to use in generating Web responses.

Presentation logic is anything that controls how the application presents information to the user.

Business logic is anything that manipulates data in order to accomplish something, such as storing data.

Extending the Power of Servlets: JavaServer Pages (JSP)

- Java technology code fragments are embedded in an HTML-like file. This code is executed at runtime to create dynamic content.
- Use regular HTML for most of page
- Mark dynamic content with special tags

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD><TITLE>Welcome to Our Store</TITLE></HEAD>
<BODY>
<H1>Welcome to Our Store</H1>
<SMALL>Welcome,
<!-- User name is "New User" for first-time visitors -->
<%= course.Utils.getUserNameFromCookie(request) %>
To access your account settings, click
<A HREF="Account-Settings.html">here.</A></SMALL>
<P>
Regular HTML for rest of on-line store's Web page
</BODY></HTML>
```

Template Page Technologies

There are three mainstream technologies available for creating HTML with embedded code:

- PHP from Apache,
- Active Server Pages (ASP) from Microsoft, and
- JSP from Sun Microsystems.

PHP and ASP, however, work only with proprietary Web servers.

PHP (PHP Hypertext Preprocessor):

```
<? for ( $i=0; $i<10; $i++ ) { ?>
  <TR><TD><? echo $i ?></TD>
<TD><? echo ($i * $i) ?></TD></TR>
<? } ?>
```

ASP (Active Server Pages):

```
<% FOR I = 0 TO 10 %>
  <TR><TD><%= I %></TD>
<TD><%= (I * I) %></TD></TR>
<% NEXT %>
```

JSP (JavaServer Pages):

```
<% for ( int i=0; i<10; i++ ) { %>
  <TR><TD><%= i %></TD>
<TD><%= (i * i) %></TD></TR>
<% } %>
```

A Servlet's Job

- Read explicit data sent by client (form data)
- Read implicit data sent by client (request headers)
- Generate the results
- Send the explicit data back to client (HTML)
- Send the implicit data to client (status codes and response headers)

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class HelloWorld extends HttpServlet {
  public void doGet(HttpServletRequest request, HttpServletResponse response) throws
   ServletException, IOException {
     PrintWriter out = response.getWriter();
     out.println("Hello World");
   }
}
```

- A servlet is an object instance of a class that extends one of the classes: `javax.servlet.Servlet` or `javax.servlet.http.HTTPServlet`.
- The ovverriden methods correspond to HTTP methods: `doGet()` and `doPost()`
- These methods:
  - have 2 parameters `javax.servlet.HttpServletRequest` and `javax.servlet.HttpServletResponse`. They represent the logical connections needed to send data strems to or from the servlet.
  - launch 2 exceptions: `java.io.IOException`, `javax.servlet.ServletException`.

Web applications

Servlets, JSP pages, HTML files, utility classes, beans, tag libraries, etc. are usually bundled together in a single directory hierarchy or file

Access to content in the Web app is always through a URL that has a common prefix:

```
http://host/webAppPrefix/blah/blah
```

• Many aspects of Web application behavior controlled through deployment descriptor (web.xml)

Organization

Related files grouped together in a single file or directory hierarchy: HTML files, JSP pages, servlets, beans, images, etc.

Portability

– All compliant servers support Web apps.

– Can redeploy on new server by moving a *single file.*

Separation

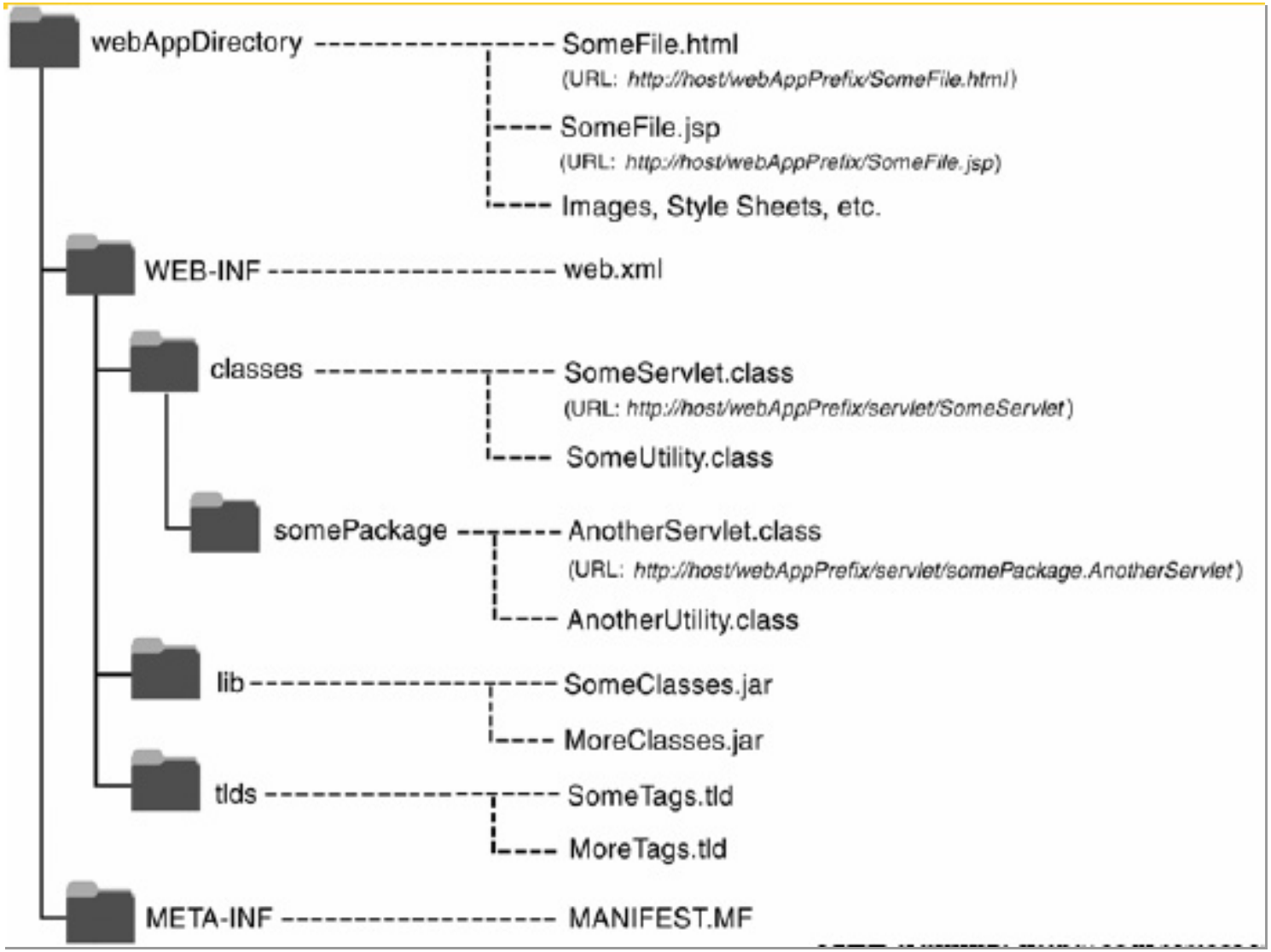
– Each Web app has its own:

- ServletContext
- Class loader
- Sessions
- URL prefix
- Directory structure

Structure of a Web Application

JSP and regular Web content (HTML, style sheets, images, etc.):
Main directory or a subdirectory thereof.
Servlets:
– WEB-INF/classes (if servlet is unpackaged – i.e. in default package)
– A subdirectory thereof that matches the package name.
Unjarred beans and utility classes:
– Same place as servlets (but *always use packages!)*
JAR files:
– WEB-INF/lib.
web.xml:
– WEB-INF
Tag Library Descriptor files:
– WEB-INF or subdirectory thereof
Files in WEB-INF not directly accessible to clients
– Server can use RequestDispatcher to forward to pages in WEB-INF

Exemple Structure

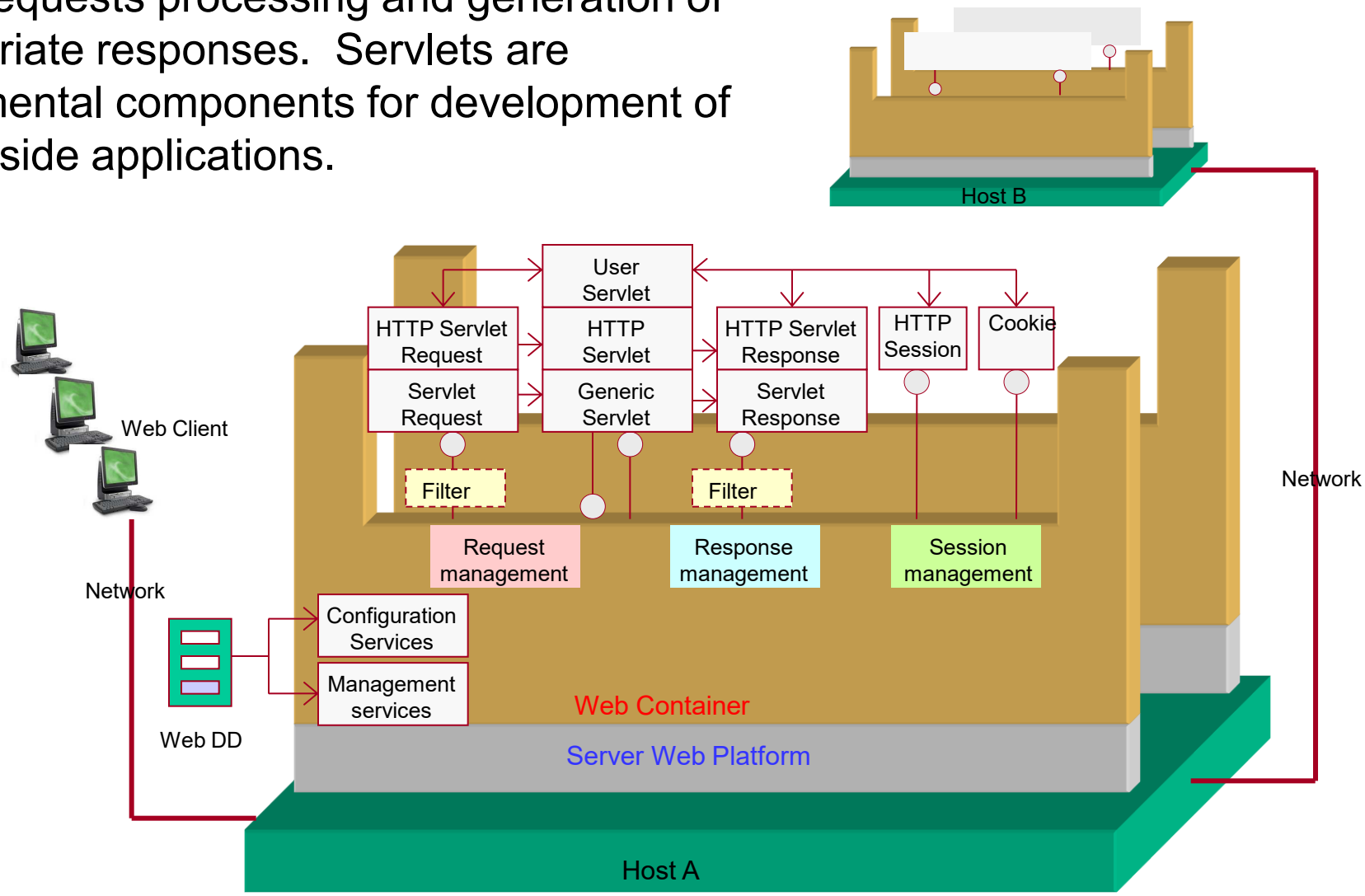


Servlet Programming

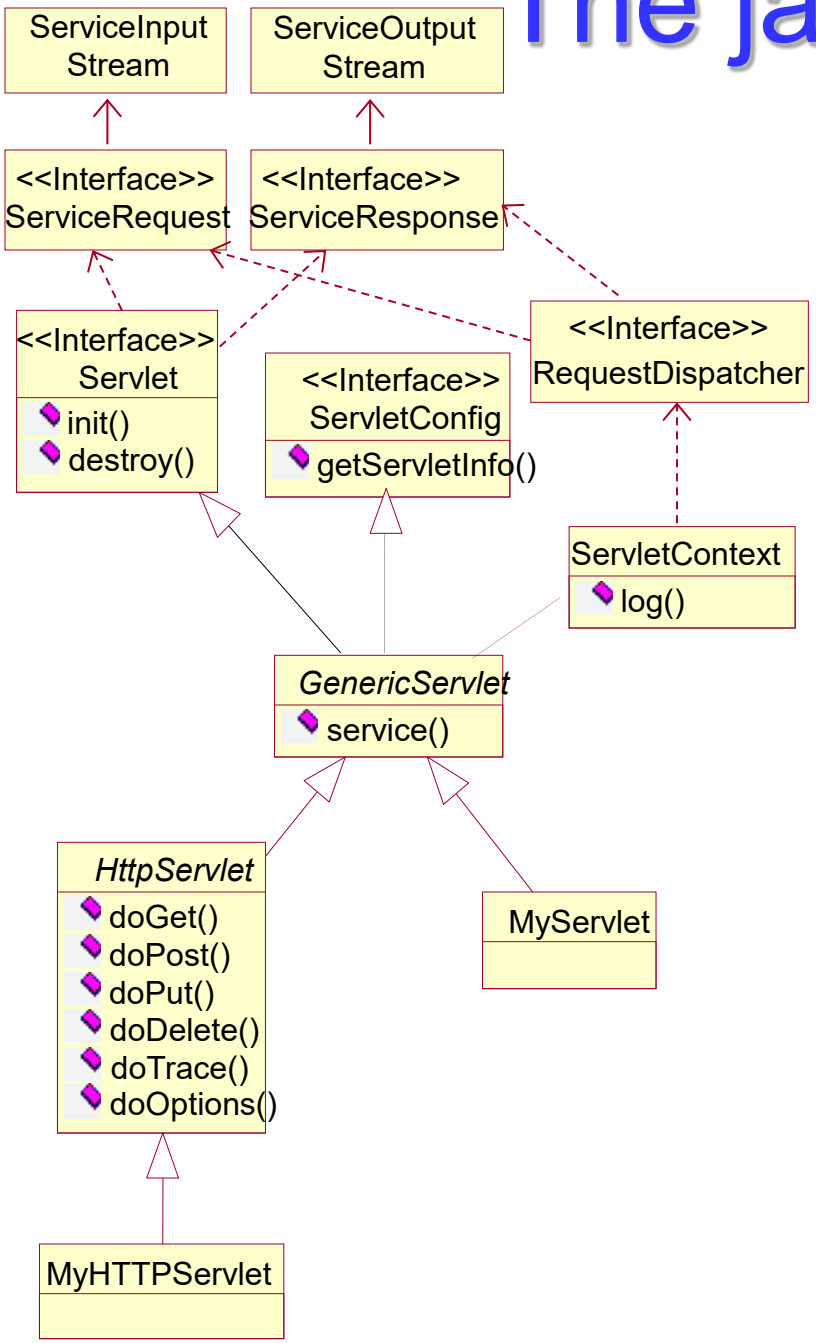


Servlet Architecture

The servlet API provides a standard interface for cleint requests processing and generation of appropriate responses. Servlets are fundamental components for development of server-side applications.



The javax.servlet package

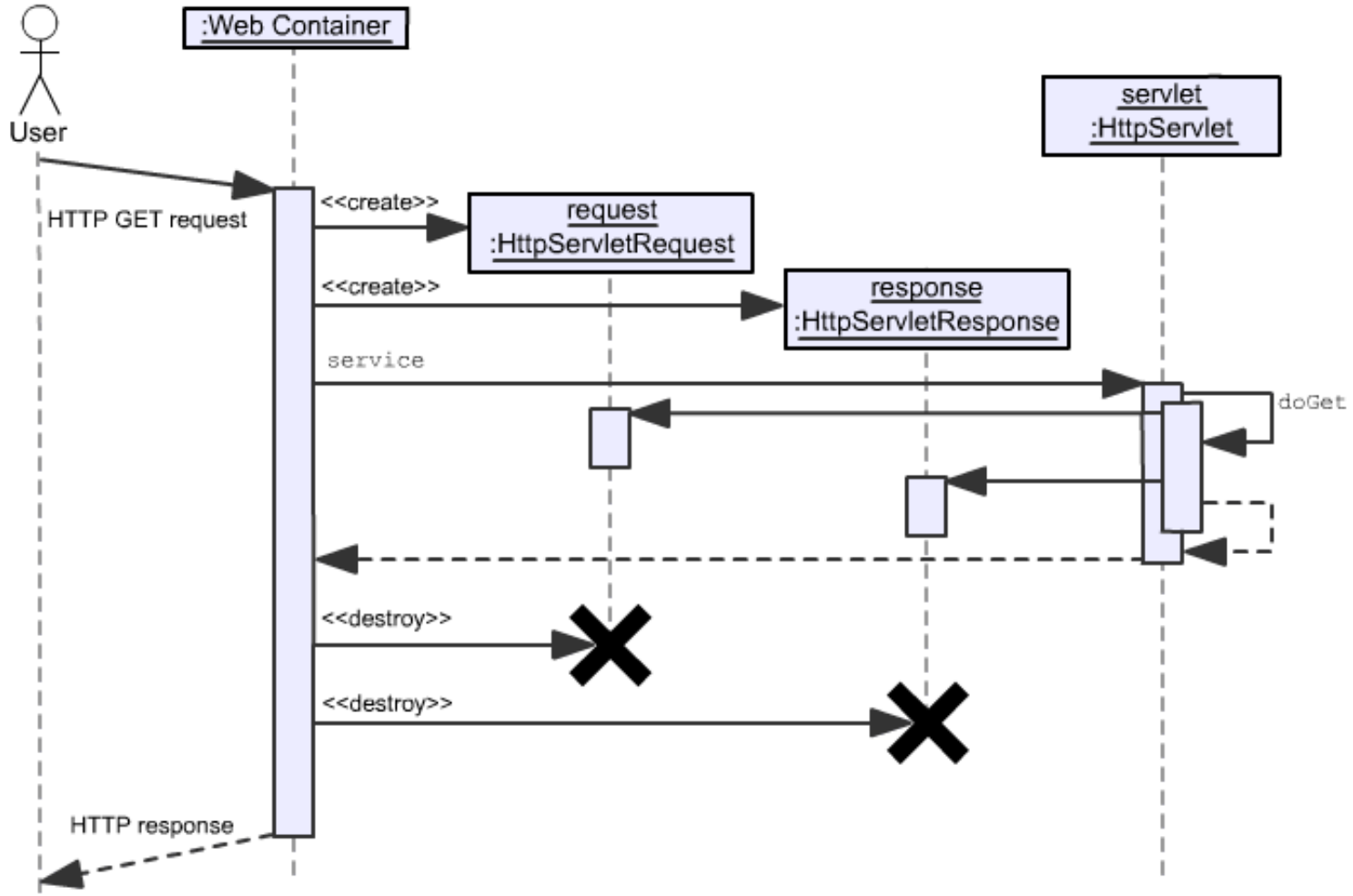


All servlets implement the interface javax.servlet.Servlet. For this they can:

- extend the abstract class *GenericServlet*
- extend the abstract subclass *HttpServlet*
- implement directly the interface *Servlet*.

The interface HttpServlet provides a simple framework for using the HTTP protocol. Its methods are not abstract, have a minimal functionality and usually overriden .
The servlets are executed in multithreaded server. As a consequence they have to written for handling several simultaneous services. The acces to shared resources has to be *synchronized*.

Request and Response Process



“Hello, world!” with `service()`

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.PrintWriter;
import java.io.IOException;
public class SampleServlet extends HttpServlet {
  static int numRequests;
  public void init(ServletConfig config) throws ServletException  {
    super.init(config);
    numRequests = 0;
  }
  public void service(HttpServletRequest request, HttpServletResponse response)
   throws ServletException, IOException  {
    // set the MIME type for the HTTP header
    response.setContentType("text/html");
    // obtain a reference to the output stream
    PrintWriter out = response.getWriter();
    out.println("<HTML>");
    out.println("<HEAD><TITLE>Sample of servlet</TITLE></HEAD>");
    out.println("<BODY>");
    out.println("<H1>Hello, world!</H1>");
    numRequests++; //increment the hit counter
    out.println("<P>This servlet has been requested " + numRequests + " times.");
    out.close(); // close always the output stream!
  }
  public String getServletInfo()   {
    return "Sample Servlet version 1.0";
  }

  public void destroy()  {
    . . . . .
  }
}
```

A Servlet that Generates HTML

Tell the browser that you are sending it a HTML page:

```
response.setContentType("text/html");
```

- Modify the `println` statements to build a legal Web page

  `print` statements should output HTML tags.

- Check your HTML with a formal syntax validator:

  http://validator.w3.org/

  http://www.htmlhelp.com/tools/validator/

```
public class HelloServlet1 extends HttpServlet {
  public void doGet(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String docType = "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 "+
      "Transitional//EN\">\n";
    out.println(docType +
                "<HTML>\n" +
                "<HEAD><TITLE>Hello</TITLE></HEAD>\n"+
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<H1>Hello</H1>\n" +
                "</BODY></HTML>");
  }
}
```

HelloServlet1

Packaging Servlets

1. Insert a package statement in the class file
2. Compile the file `.java`
3. Move the file `.class` to :
   `<SERVER_HOME>/webapps/ROOT/WEB-INF/classes/course`
4. Access it with:
   `http://localhost:8080/servlet/course.HelloServlet2`

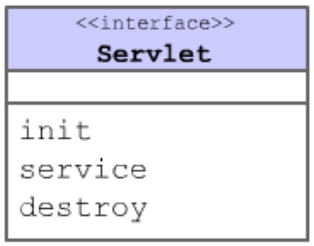
```
package course;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class HelloServlet2 extends HttpServlet {
  public void doGet(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String docType = "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 "+
      "Transitional//EN\">\n";
    out.println(docType +
                "<HTML>\n" +
                "<HEAD><TITLE>Hello (2)</TITLE></HEAD>\n"+
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<H1>Hello (2)</H1>\n" +
                "</BODY></HTML>");
  }
}
```

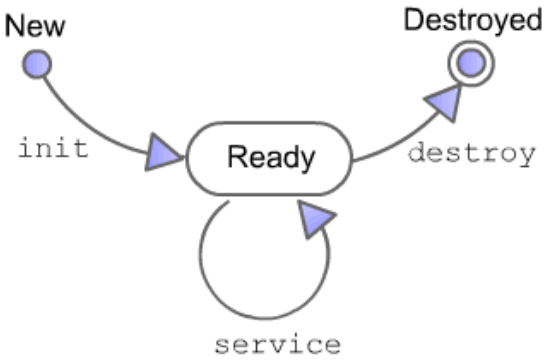
HTML-Building Utilities

```
package course;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class HelloServlet3 extends HttpServlet {
  public void doGet(HttpServletRequest request, HttpServletResponse response)
             throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String title = "Hello (3)";
    out.println(ServletUtilities.headWithTitle(title)+
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<H1>" + title + "</H1>\n" +
                "</BODY></HTML>");
  }
}

public class ServletUtilities {
  public static final String DOCTYPE =
    "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 " +  "Transitional//EN\">";
  public static String headWithTitle(String title) {
    return(DOCTYPE + "\n" +
           "<HTML>\n" +
           "<HEAD><TITLE>" + title +
           "</TITLE></HEAD>\n");
  }
  ...
}
```

The Servlet Life Cycle (I)





init()

Executed once when the servlet is first loaded.

Not called for each request.

service()

Called in a new thread by server for each request. Dispatches to `doGet`, `doPost`, etc.

Do not override this method!

service() gives support for:

- HEAD requests
- OPTIONS requests
- TRACE requests

doGet(), doPost(), doXxx()

Handles GET, POST, etc. requests.

Override these to provide desired behavior.

destroy()

Called when server deletes servlet instance. *Not called after each request.*

Servlet initialization

After a servlet loading and instantiation and before requests processing the web server must initialize the servlet. This initialization can be customized by overriding the init() method of the HttpServlet class, for instance by allowing the servlet to:

- read configuration data
- initialize the resources
- do other activities.

```
public class CatalogServlet extends HttpServlet {
   private BookDB bookDB;
   public void init() throws ServletException {
     bookDB = (BookDB)getServletContext().getAttribute("bookDB");
     if (bookDB == null)
       throw new UnavailableException("Couldn't get database.");
   }
 }
```

The Servlet Life Cycle (II)

The container identifies and loads the servlet class

The container creates one or more instances of the servlet class

init

•The container calls the method init() for each instance with an object that implements ServletConfig.
•During init() we can read persistent data from databases or create connections to databases etc.
•We can access the ServletContext object

A request for the servlet arrives

service

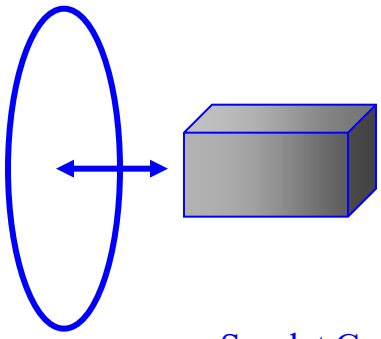
The container selects the servlet instance and calls the method service()

The method service() processes the request and returns the output to the client

The container waits for the next request

destroy

The container destroys the servlet after the call of the method destroy()



Server

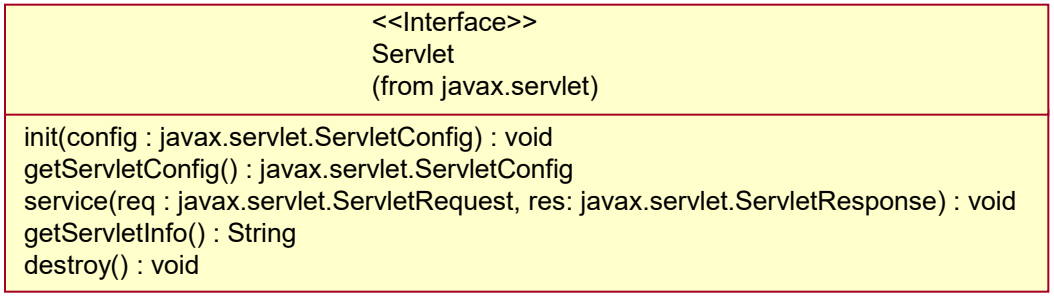
Servlet Container

Servlet Reloading

- Servlets need a specialized class loader. When the web server sends a request to the servlet, it verifies if the `.class` of the servlet was modified. If the file was modified, a new instance is created by the loader and loaded in memory.
- This is why the servlets are not located in the web server classpath, but in a special directory *`server_root`*`/servlets` where the loader search them.
- This mechanism allows us to dynamically substitute a servlet without the server shutting down.
- Warning. The class loader is used for only servlet classes that are not found in the server classpath. As a consequence:

1. The servlet in the server classpath are non reloaded or verified for reloading.
2. The classes supporting the servlet are not reloaded. They are loaded only one time and remain in memory even if the servlet `.class` file changes.

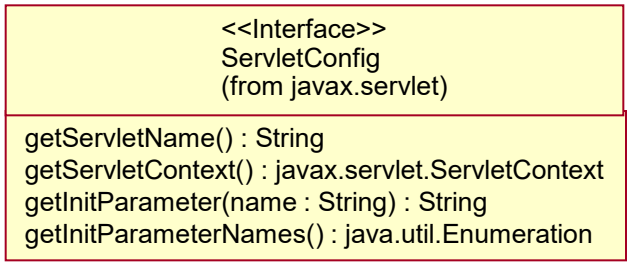
Servlet Interface



- `init()` is called by the container when the servlet is instantiated. The method has a `javax.servlet.ServletConfig` parameter containing initialization information provided by the container environment.
- `destroy()` is called before the servlet is removed in order to free the engaged resources.
- `getServletConfig()` is used to obtain a "handler" of the `ServletConfig` object.
- `getServletInfo()` return information on the servlet author and version.
- `service()` is called by the container to process requests and generate responses during the normal use of the servlet. The method has two parameters: `javax.servlet.ServletRequest` and `javax.servlet.ServletResponse`.

The container calls `service()` for more than one concurrent calls. So, `service()` is implemented to be run on separate threads.

ServletConfig



The Servlet specification provides an interface called `ServletConfig` to access servlet configuration information. This information includes the servlet's name and the initialization parameters configured in the deployment descriptor.

- `getInitParameter()` provides the servlet with access to the initialization parameters for that servlet instance. These parameters are declared in the deployment descriptor. The names and values are arbitrary strings.
- `getInitParameterNames()` returns an `Enumeration` object that can be used to scroll initialization parameters.
- `getServletName()` provides the servlet name.
- `getServletContext()` provides the servlet context.

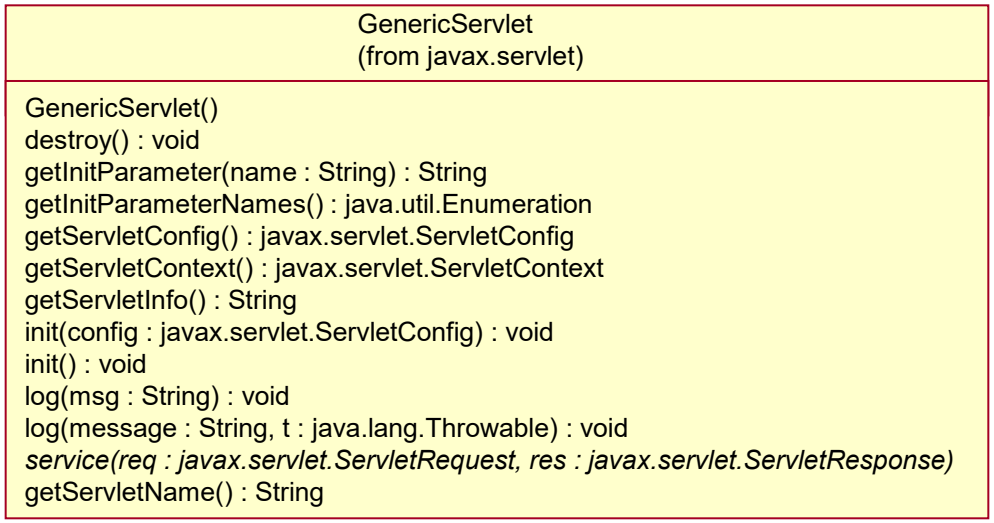
Using Servlet Configuration

```
<servlet>
        <servlet-name>TexasHello</servlet-name>
        <servlet-class>sl314.web.HelloServlet</servlet-class>
       <init-param>
            <param-name>greetingText</param-name>
            <param-value>Howdy</param-value>
        </init-param>
        <init-param>
            <param-name>languageCode</param-name>
            <param-value>en-US</param-value>
        </init-param>
    </servlet>
```




```
public class HelloServlet extends HttpServlet {
  private String greetingText;
  private String languageCode;
  public void init() {
    greetingText = getInitParameter("greetingText");
    languageCode = getInitParameter("languageCode");
   }
  public void doPost(HttpServletRequest request,
        HttpServletResponse response) throws IOException { ..... }
```

GenericServlet Abstract Class



The class provides default implementations for `Servlet` and `ServletConfig` interfaces. Among implementes methods there are `init()` and two `log()` methods.

The `GenericServlet's` concrete classes of (for instance, `HTTPServlet`) define specific mechanisms for requesting and responsing. The programmer has to implement the abstract method `service()`.

<<Interface>>
ServletContext
(from javax.servlet)

getContext(uripath : String) : ServletContext
getMajorVersion() : int
getMinorVersion() : int
getMimeType(file: String) : String
getResourcePaths(path : String) : java.util.Set
getResource(path : String) : java.net.URL
getResourceAsStream(path : String) : java.io.InputStream
getRequestDispatcher(path : String) : javax.servlet.RequestDispatcher
getNamedDispatcher(name : String) : javax.servlet.RequestDispatcher
log(msg : String) : void
log(message : String, throwable : java.lang.Throwable) : void
getRealPath(path : String) : String
getServerInfo() : String
getInitParameter(name : String) : String
getInitParameterNames() : java.util.Enumeration
getAttribute(name : String) : Object
getAttributeNames() : java.util.Enumeration
setAttribute(name : String, object : Object) : void
removeAttribute(name : String) : void
getServletContextName() : String

ServletContext (I)

Every servlet within a given Web application has access to the ServletContext object for that Web application by using the getServletContext method that is inherited from the GenericServlet class.

ServletContext provides a reference to the servlet container context . There is a instance for any servlet instance and JVM instance.
In the HttpServlet case, the servlet context is specified as relative to a root context .
`getContext()` provides a reference to another ServletContext context on the same Web Server if an absolute path begginin with '/' is provided from the root of server documents.
`getRequestDispatcher()` and `getNamedDispatcher()`. ServletContext is used as input point where the web requests can be dispatched to other servlet , in the same servlet context .

Context Initialization Parameters

Just as each servlet definition may have one or more initialization parameters, a Web application may have one or more initialization parameters. These are also configured in the deployment descriptor using the context-param element tag.

```
<web-app>
    <display-name>SL-314 Serlvet Context Example . . .</display-name>
    <description>
        This Web Application demonstrates application-scoped variables
        (in the servlet context) and WebApp lifecycle listeners.
   </description>
    <!-- Initialize Catalog -->
    <context-param>
       <param-name>catalogFileName</param-name>
       <param-value>/WEB-INF/catalog.txt</param-value>
    </context-param>
```

Context initialization parameters are accessed using the `getInitParameter` method on the ServletContext object.

ServletContext (II)

`getRealPath()` provides a full path of a web resources if its relative path in the server context is included as a parameter.
`getResource()` provides a URL of a resource included with the relative path as a parameter.
`getResourceAsStream()` returns as an InputStream object a resource included as a parameter with its relative path in the server context.
`getMimeType()` provides a string that specifies the MIME type/subtype of the information contained in a file included with its name as a parameter.
`getInitParameter()` returns the value of a parameter of the container environment configuration. The parameter name is included as a parameter.
`getInitParameterNames()` returns an Enumeration objects of a list of strings representing the container environment configuration .
Besides configuration parameters, we can associate as attributes to the container environment a collection of `java.lang.Object` objects. These attributes can be shared with other servlets in the same context, as well as declared, found, or removed with `setAttribute()`, `getAttribute()` and `removeAttribute()`.
`getAttributeNames()` returns an Enumeration object containing the name of all attributes of the container context.
`log()` are methods used for writing information on exceptions in a log file.

A Servlet Example

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import com.datamat.personnel.*;
import com.datamat.sql.*;
public class ServletAnagrafe extends HttpServlet {
   public void doPost (HttpServletRequest request,
   ServletException, IOException   {
      Anagrafe persona=null;
      PrintStream out = new PrintStream(response.getOutputStream());
      response.setContentType ("text/html");
      try {
         persona = new Anagrafe(
            request.getParameter("firstname"),
            request.getParameter("lastname"),
            request.getParameter("day"),
            request.getParameter("month"),
            request.getParameter("year"),
            request.getParameter("sex").charAt(0),
            request.getParameter("CNP"),
      } catch (CNPErratoException b) {
        out.println("Codice fiscale errato.");
        b.printStackTrace();
      }
      try {
        Communicator.wri
        out.println("Done. ");
      } catch( Exception x ) {
         x.printStackTrace(out);
      }
      out.close();
      response.getOutputStream().close();
   }
} // class Process
```

Process extends HttpServlet

Overriding of the method *doPost()* of the abstract class HttpServlet. POST is preferred because the operation is a transaction (updating). The method parametersare objects HttpServletRequest and HttpServletResponse.

setContentType() is used to indicate the output type (in this case HTML). Closed with close().

getParameter() is a method of the superclass ServletRequest that returns the values of the request parameters

Debugging code with other available methods

*Communicato*r is imported from "com.datamat.personnel"

```
// debugging code
out.println("<BR>URI: " +
request.getRequestURI().toString());
out.println("<BR>Query String: " + request.getQueryString());
out.println("<BR>Method: " + request.getMethod());
out.println("<BR>Header Names: " );
String name, value;
for (Enumeration e = request.getHeaderNames();
                                    e.hasMoreElements();)
  {
    name = (String) e.nextElements();
    value = (String) request.getHeader(name);
    out.println("<BR> " + name + "=" + value);
  }
```

Handling the Client Request: FORM Data

- The role of form data
- Creating and submitting HTML forms
- Reading individual request parameters
- Reading the entire set of request parameters
- Handling missing and malformed data
- Dealing with incomplete form submissions
- Filtering special characters out of the request parameters

The Role of FORM Data

- Example URL at online travel agent:

```
http://host/path?user=John+Smith&origin=ppp&dest=ttt
                QUERY_STRING
```

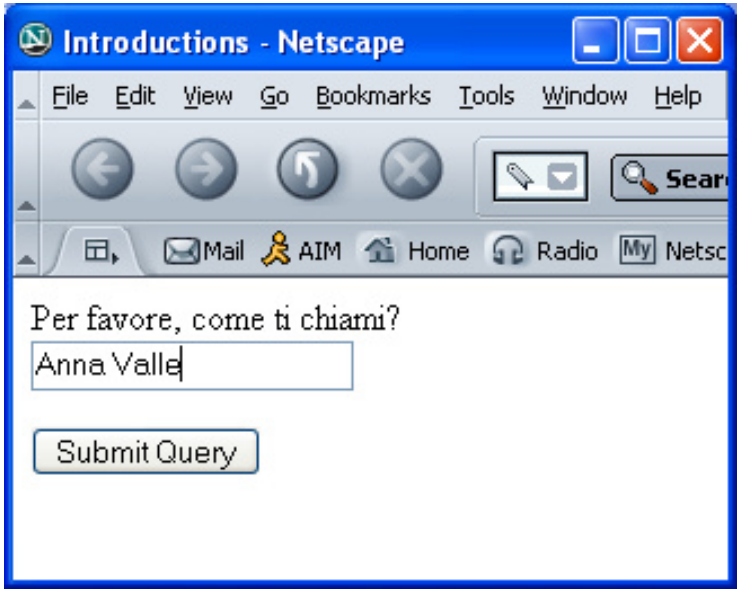
  In `query_string` the names come from HTML and the values from the user.
- Parsing form (query) data in traditional CGI
  - Read the data one way (QUERY_STRING) for GET requests, another way (standard input) for POST requests
  - Chop pairs at ampersands, then separate parameter names (left of the =) from parameter values (right of the =)
  - URL decode values (e.g., "%7E" becomes "~")
- Greatly simplified in servlets
  - Use `request.getParameter()` in all cases.
  - Gives URL-decoded result .

Creating Form Data (I)

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
  <HEAD>
    <TITLE>Introductions</TITLE>
  </HEAD>
  <BODY>
    <FORM METHOD="GET" ACTION="http://localhost:8080/servlet/Ciao">
       Per favore, come ti chiami?
      <INPUT TYPE=TEXT NAME="name"><P>
      <INPUT TYPE=SUBMIT NAME="Submit Query">
    </FORM>
  </BODY>
</HTML>
```

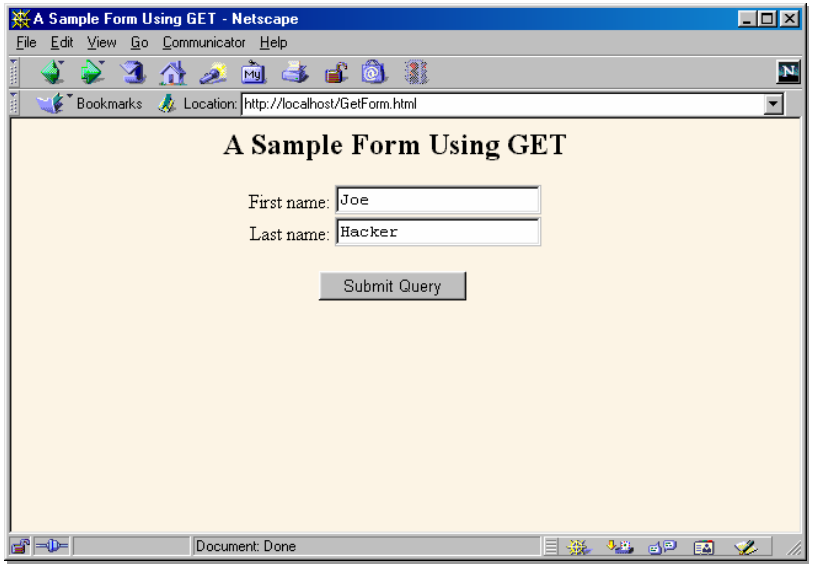
The request to be sent to web server is:

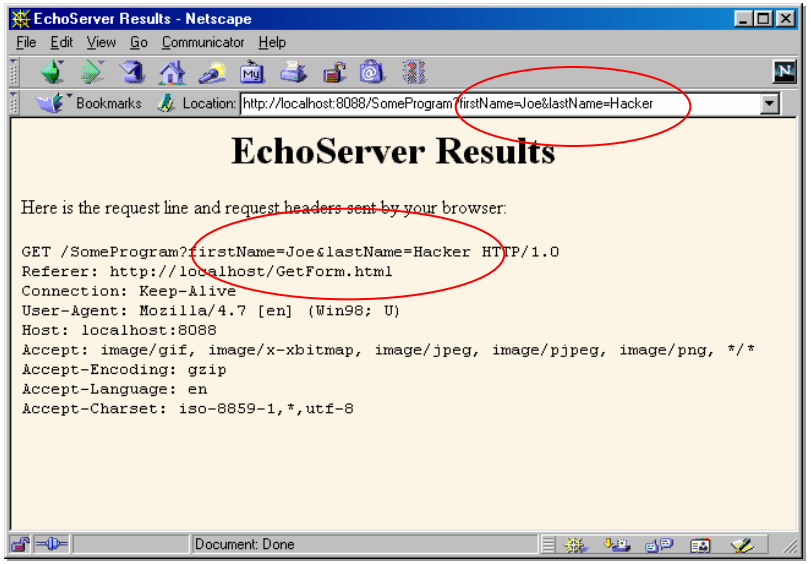
http://localhost:8080/servlet/Ciao?name=Anna+Valle



Creating Form Data

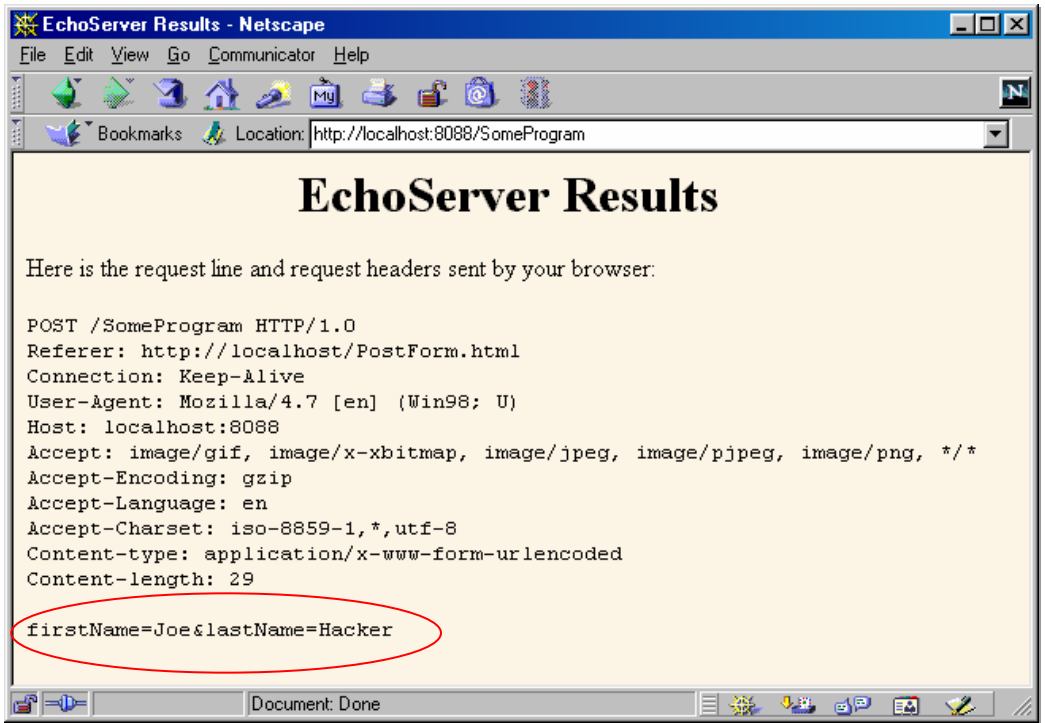
```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD><TITLE>A Sample Form Using GET</TITLE></HEAD>
<BODY BGCOLOR="#FDF5E6">
<H2 ALIGN="CENTER">A Sample Form Using GET</H2>
<FORM ACTION="http://localhost:8080/SomeProgram">
  <CENTER>
  First name:
  <INPUT TYPE="TEXT" NAME="firstName" VALUE="Joe"><BR>
  Last name:
  <INPUT TYPE="TEXT" NAME="lastName" VALUE="Hacker"><P>
  <INPUT TYPE="SUBMIT"> <!- Press this to submit form -->
  </CENTER>
</FORM>
</BODY></HTML>
```



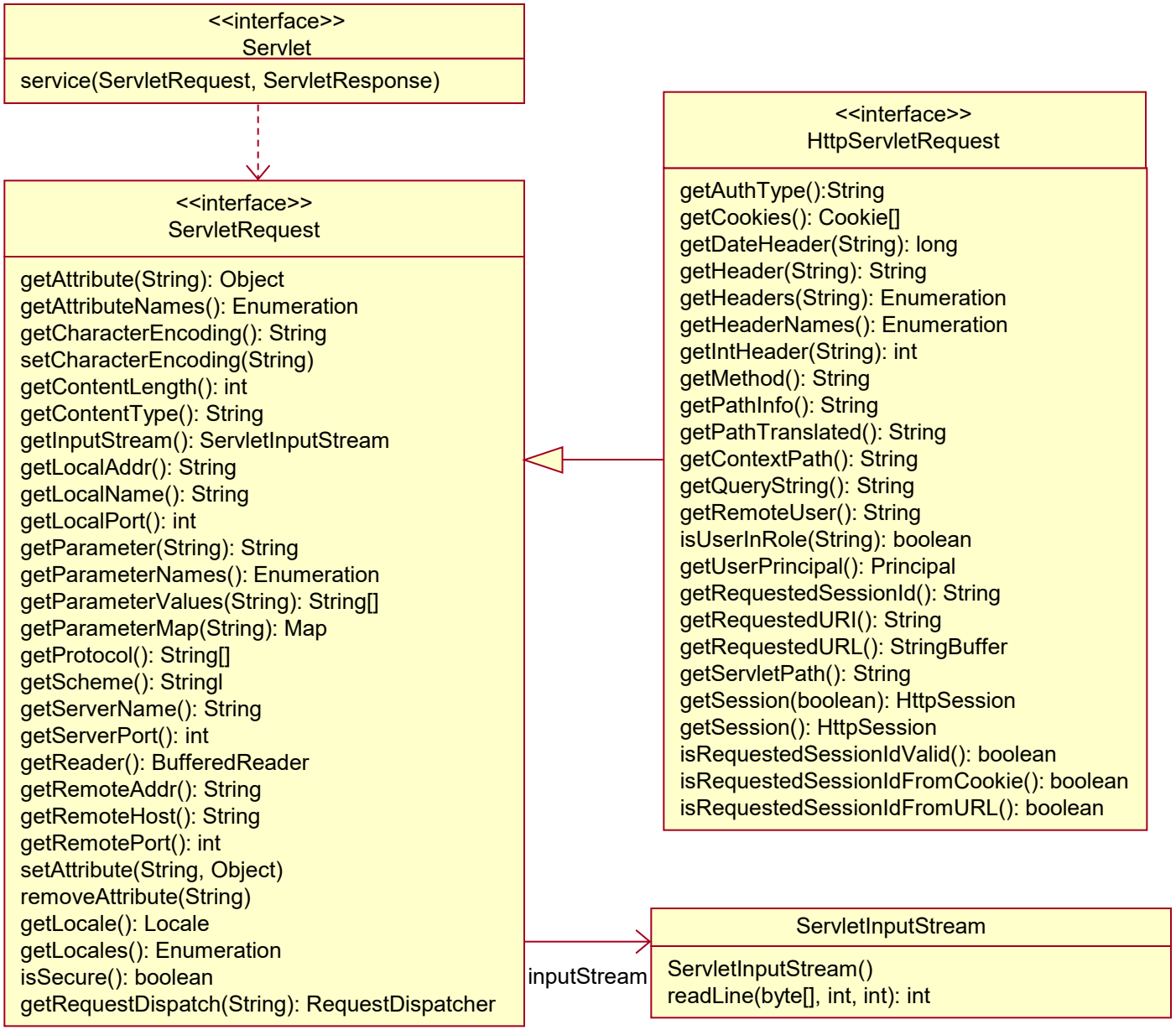


Sending POST Data

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD><TITLE>Form che utilizza il metodo POST </TITLE></HEAD>
<BODY BGCOLOR="#FDF5E6">
<H2 ALIGN="CENTER">Form che utilizza il metodo POST</H2>
<FORM ACTION="http://localhost:8080/anagrafe" METHOD="POST">
  <CENTER>
  Nome:
  <INPUT TYPE="TEXT" NAME="firstName" VALUE="John"><BR>
  Cognome:
  <INPUT TYPE="TEXT" NAME="lastName" VALUE="Smith"><P>
  <INPUT TYPE="SUBMIT">
  </CENTER>
</FORM>
</BODY>
</HTML>
```



Request processing



HttpServletRequest Interface

- HttpServletRequest extends ServletRequest to incapsulate specific aspects of the HTTP requests.
- When the container receives a HTTP request a HttpServletRequest object is created and passed as an argument to the `service()` method call. Then the HttpServlet object calls the appropriate `doXXX()` method, as it was identified by the `getMethod()` method.
- Information on HTTP requests can be obtained by calling the HttpServletRequest object:
  - Names, values and dates in the header with the following methods: `getHeaderNames()`, `getHeader()` and `getIntHeader()` and, respectively, `getDateHeader()` ;
  - Parts of the URI path of the requested resource with the following methods : `getRequestURI()`, `getContextPath()`, `getServletPath()`, `getQueryString()` and `getPathTranslated()`;
- Example. Let consider the following request:

`http://www.campers.com/applicMia/CamperServlet/accessori/GPS?tipo=LX`

`getRequestURI()` returns `applicMia/CamperServlet/accessori/GPS?tipo=LX`

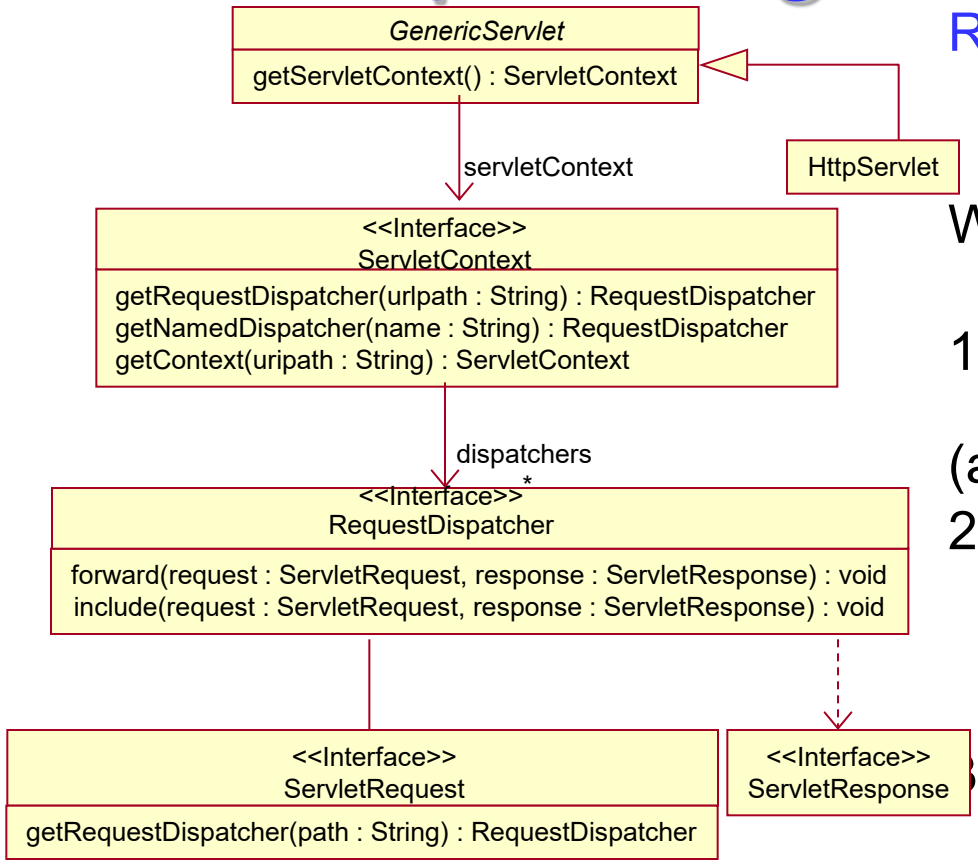
`getContextPath()` returns `applicMia`

`getServletPath()` returns `CamperServlet`

`getPathTranslated()` returns `accessori/GPS`

`getQueryString()` returns `tipo=LX`

Dispatching HTTP Requests



A RequestDispatcher can dispatch a request to an other component with forward() only when there is not generated output data. Otherwise with include() the content of a resource can be included in the current servlet output.

RequestDispatcher = an object used to dispatch requests arrived from other components.

We get a RequestDispatcher in three ways:

1. Call `ServletContext` with `getNamedDispatcher()`
(all web components are named in DD)
2. Call `ServletContext` with: `getRequestDispatcher()` with the resource URL relative to the servlet context (/…)
3. Call the ServletContext object of an other environment with `getRequestDispatcher()`. In order to obtain the ServletContext object of an other environment you have to call the current ServletContext object with `getContext()` and provide it with the resource URL relative to the current context

Reading Form Data In Servlets

`request.getParameter("name")`

- Returns URL-decoded value of first occurrence of name in query_string
- Works identically for GET and POST requests
- Returns null if no such parameter is in query data.

`request.getParameterValues("name")`

- Returns an array of the URL-decoded values of *all* occurrences of name in query_string
- Returns a one-element array if param not repeated
- Returns null if no such parameter is in query

`request.getParameterNames()` or `request.getParameterMap()`

- Returns Enumeration or Map of request params
- Usually reserved for debugging

Handling input in multiple languages

- Use server's default character set:

```
String nome = request.getParameter("firstName");
```

- It is possible to set the character set. Ex.: Convert from English (Latin-1) to Japanese:

```
String nomeWrongEncoding = request.getParameter("firstName");
String nome = new String(nomeWrongEncoding.getBytes(), "Shift_JIS");
```

- Accept either English or Japanese:

```
request.setCharacterEncoding("JISAutoDetect");
String nome = request.getParameter("firstName");
```

Reading Raw Form Data and Parsing Uploaded Files

Raw Data

- `request.getReader()`
- `request.getInputStream()`

Data no longer available via `getParameter()` after this

Parsing uploaded files

- HTML has a way of submitting entire files;

```
<INPUT TYPE="FILE". . .>
```

- Servlet/JSP APIs have no builtin way to parse files e;

- Popular third-party library available from the Apache/Jakarta "Commons" library di `Apache/Jakarta`, visitare:

```
http://jakarta.apache.org/commons/fileupload/
```

Reading the Three Parameters

Un form con tre parametri

```
<FORM ACTION="/servlet/corsoservlets.TreParam">
  First Parameter:  <INPUT TYPE="TEXT" NAME="param1"><BR>
  Second Parameter: <INPUT TYPE="TEXT" NAME="param2"><BR>
  Third Parameter:  <INPUT TYPE="TEXT" NAME="param3"><BR>
  <CENTER><INPUT TYPE="SUBMIT"></CENTER>
</FORM>
```

TreParam

Lettura e analisi del form

```
public class ThreeParams extends HttpServlet {
  public void doGet(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String title = "Tre parametri";
    out.println(ServletUtilities.headWithTitle(title) +
          "<BODY BGCOLOR=\"#FDF5E6\">\n" +
          "<H1 ALIGN=\"CENTER\">" + title + "</H1>\n" +
          "<UL>\n" +
          "  <LI><B>param1</B>: "
          + request.getParameter("param1")+"\n"
          "  <LI><B>param2</B>: "
          + request.getParameter("param2")+"\n"
          "  <LI><B>param3</B>: "
          + request.getParameter("param3")+"\n"
          "</UL>\n" +
          "</BODY></HTML>");
  }
}
```



Reading All Parameters

```
public class ShowParameters extends HttpServlet {
  public void doGet(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String docType = "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 " + "Transitional//EN\">\n";
    String title = " Reading All Request Parameters ";
    out.println(docType +
                "<HTML>\n" +
                "<HEAD><TITLE>"+title + "</TITLE></HEAD>\n"+
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<H1 ALIGN=CENTER>" + title + "</H1>\n" +
                "<TABLE BORDER=1 ALIGN=CENTER>\n" +
                "<TR BGCOLOR=\"#FFAD00\">\n" +
                "<TH>Parameter Name<TH>Parameter Value(s)");
    Enumeration paramNames = request.getParameterNames();
    while(paramNames.hasMoreElements()) {
      String paramName = (String)paramNames.nextElement();
      out.print("<TR><TD>" + paramName + "\n<TD>");
      String[] paramValues = request.getParameterValues(paramName);
      if (paramValues.length == 1) {
        String paramValue = paramValues[0];
        if (paramValue.length() == 0)
           out.println("<I>No Value</I>");
        else
           out.println(paramValue);
      } else {
         out.println("<UL>");
         for(int i=0; i<paramValues.length; i++) {
            out.println("<LI>" + paramValues[i]);
        }
        out.println("</UL>");
      }
    }
    out.println("</TABLE>\n</BODY></HTML>");
  }
  public void doPost(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
    doGet(request, response);
  }
}
```

Checking for Missing and Malformed Data

Missing

- Field missing in form :
  - `getParameter()` returns null
- Field blank when form submitted
  - `getParameter()` returns an empty string (or possibly a string with whitespace in it)
- Must check for null before checking for empty string :

```
String param = request.getParameter("someName");
if ((param == null) || (param.trim().equals("")))
  doSomethingForMissingValues(...);
else
  doSomethingWithParameter(param);
```

Malformed

Value is a nonempty string in the wrong format.

Handling Missing and Malformed Data

- Use default values
  - Replace missing values with application-specific standard values
- **Redisplay the form**
  - Show the form again, with missing values flagged
  - Previously-entered values should be preserved
  - Four options to implement this
    - Have the same servlet present the form, process the data, and present the results.
    - Have one servlet present the form; have a second servlet process the data and present the results.
    - Have a JSP page "manually" present the form; have a servlet or JSP page process the data and present the results.
    - Have a JSP page present the form, automatically filling in the fields with values obtained from a data object. Have a servlet or JSP page process the data and present the results.

Data correction

```
headingFont = replaceIfMissingOrDefault(headingFont, "");
int headingSize = getSize(request.getParameter("headingSize"), 32);
String bodyFont = request.getParameter("bodyFont");
bodyFont = replaceIfMissingOrDefault(bodyFont, "");
int bodySize = getSize(request.getParameter("bodySize"), 18);
String fgColor = request.getParameter("fgColor");
fgColor = replaceIfMissing(fgColor, "BLACK");
String bgColor = request.getParameter("bgColor");

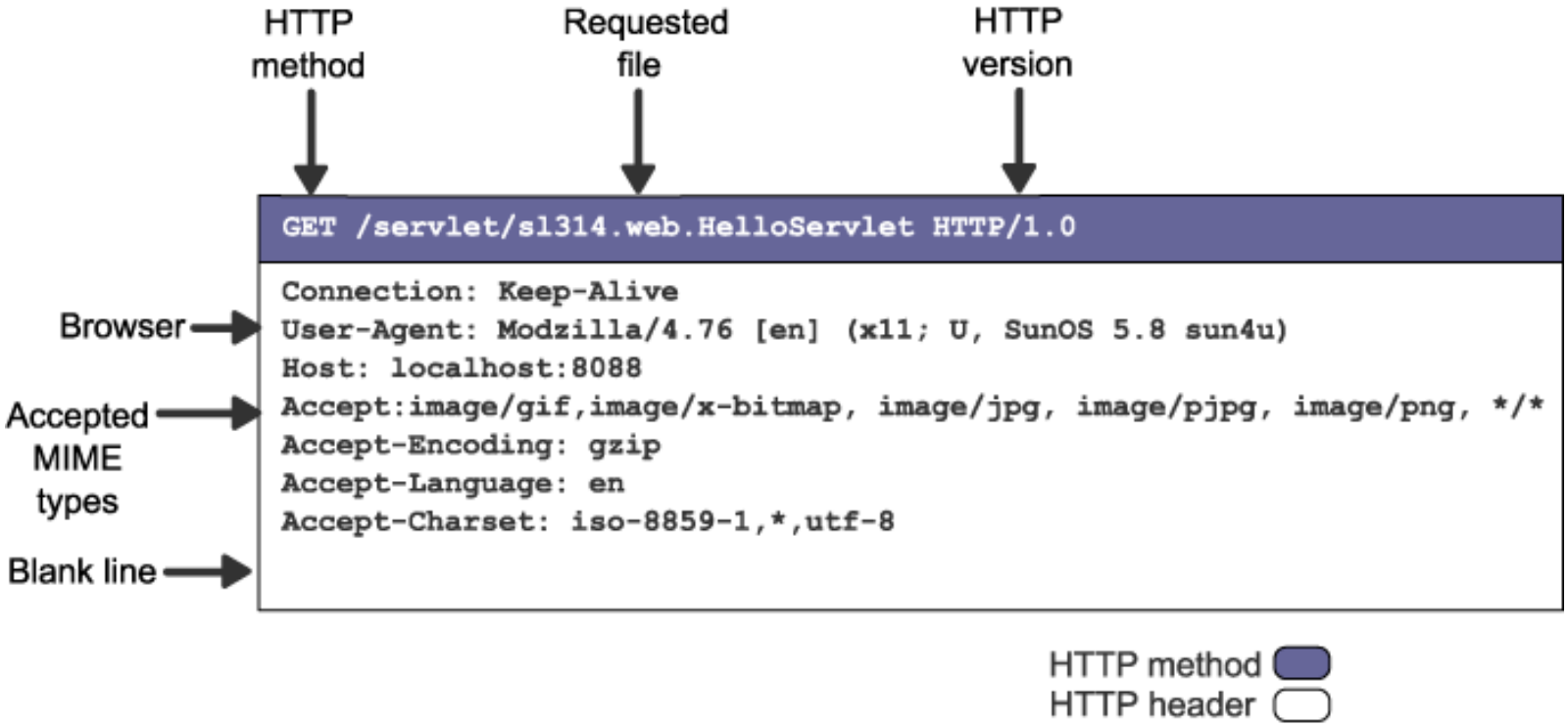
private String replaceIfMissingOrDefault(String orig, String replacement) {
    if ((orig == null) || (orig.trim().equals(""))) {
      return(replacement);
    } else {
      return(orig);
    }
  }
```

Handling the Client Request: HTTP Request Headers

- Reading HTTP request headers
- Building a table of all the request headers
- Understanding the various request headers
- Reducing download times by compressing pages
- Differentiating among types of browsers

The `HttpServletRequest` API

A Typical HTTP Request



Methods in HttpServletRequest

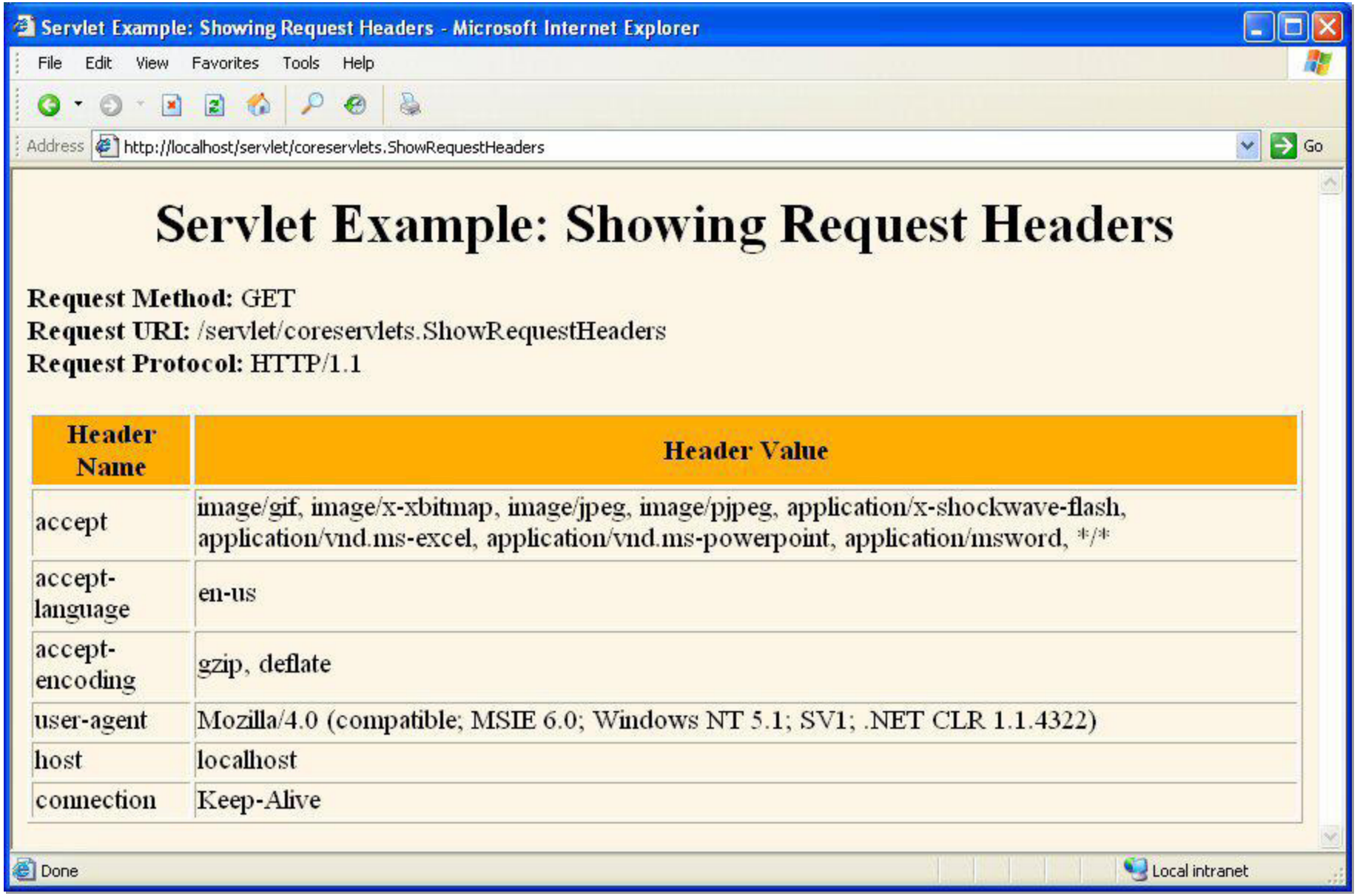
- General
  `getHeader` (header name is not case sensitive!)
  `getHeaders getHeaderNames`
- Specialized
  `getCookies getAuthType getRemoteUser getContentLength getContentType`
  `getDateHeader getIntHeader`
- Related Info
  `getMethod, getRequestURI , getQueryString, getProtocol`

Making a Table of All Request Headers

```
package course;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;
public class ShowRequestHeaders extends HttpServlet {
  public void doGet(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String docType = "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 " + "Transitional//EN\">\n";
    String title = "Mostra intestazioni della richiesta";
    out.println
      (docType +
       "<HTML>\n" +
       "<HEAD><TITLE>"+title+"</TITLE></HEAD>\n"+
       "<BODY BGCOLOR=\"#FDF5E6\">\n" +
       "<H1 ALIGN=\"CENTER\">" + title + "</H1>\n" +
       "<B>Request method: </B>" +
       request.getMethod() + "<BR>\n" +
       "<B>Request URI: </B>" +
       request.getRequestURI() + "<BR>\n" +
       "<B>Request protocol: </B>" +
       request.getProtocol()+ "<BR><BR>\n" +"<TABLE BORDER=1 ALIGN=\"CENTER\">\n" +
                "<TR BGCOLOR=\"#FFAD00\">\n" +
                "<TH>Header Name<TH>Header Value");
    Enumeration headerNames = request.getHeaderNames();
    while(headerNames.hasMoreElements()) {
      String headerName = (String)headerNames.nextElement();
      out.println("<TR><TD>" + headerName);
      out.println("    <TD>"+request.getHeader(headerName));
    }
    out.println("</TABLE>\n</BODY></HTML>");
  }
  // Since this servlet is for debugging, have it handle GET and POST identically.
  public void doPost(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
    doGet(request, response);
  }
}
```

MostraIntestazioniRichiesta

Making a Table of All Request Headers



Common HTTP 1.1 Request Headers (I)

- Accept
  - Indicates MIME types browser can handle
  - Can send different content to different clients. For example, PNG files have good compression characteristics but are not widely supported in browsers. A servlet could check to see if PNG is supported, sending
    `<IMG SRC="picture.png" ...>` if it is supported, and
    `<IMG SRC="picture.gif" ...>` if not.
  - Warning: IE incorrectly sets this header when you hit the Refresh button. It sets it correctly on original request.

Accept-Encoding

  - Indicates encodings (e.g., gzip or compress) browser can handle.

- Authorization
  - User identification for password-protected pages.
  - Instead of HTTP authorization, use HTML forms to send username/password and store info in session object. This approach is usually preferable because standard HTTP authorization results in a small, terse dialog box that is unfamiliar to many users.
  - Servers have high-level way to set up password-protected pages without explicit programming in the servlets.

Common HTTP 1.1 Request Headers (II)

- Connection
  - In HTTP 1.0, keep-alive means browser can handle persistent connection. In HTTP 1.1, persistent connection is default. Persistent connections mean that the server can reuse the same socket over again for requests very close together from the same client (e.g., the images associated with a page, or cells within a framed page).
  - Servlets can't do this unilaterally; the best they can do is to give the server enough info to permit persistent connections. So, they should set `Content-Length` with `setContentLength` (using `ByteArrayOutputStream` to determine length of output).
- Cookie
  - Gives cookies previously sent to client. Use `getCookies`, not `getHeader`.
- Host
  - Indicates host given in original URL. This is a required header in HTTP 1.1.
- If-Modified-Since
  - Indicates client wants page only if it has been changed after specified date
  - Don't handle this situation directly; implement `getLastModified()` instead.

Common HTTP 1.1 Request Headers (III)

Referer

- URL of referring Web page
- Useful for tracking traffic; logged by many servers
- Can also be used to let users set preferences and then return to the page they came from
- Can be easily spoofed; don't let this header be sole means of deciding how much to pay sites that show your banner ads.
- Some browsers (Opera), ad filters (Web Washer), and personal firewalls (Norton) screen out this header

User-Agent

  - Best used for identifying *category of client*
    - Web browser vs. I-mode cell phone, etc.
  - For Web applications, use other headers if possible
  - Again, can be easily spoofed

Access to the server information

```
public String ServletRequest.getServerName() // the server name
public int ServletRequest.getServerPort()    // the server port
public String ServletContext.getServerInfo() // info on the software server
public Object ServletContext.getAttribute(String name)  // an attribute of the server environment
```

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
public class InformazioniServer extends GenericServlet {
   public void service(ServletRequest req, ServletResponse res)
     throws ServletException, IOException {
     res.setContentType("text/plain");
     PrintWriter out = res.getWriter();
     out.println("req.getServerName(): " + req.getServerName());
     out.println("req.getServerPort(): " + req.getServerPort());
     out.println("getServletContext().getServerInfo(): " +
         getServletContext().getServerInfo());
     out.println("getServerInfo() nome: " +
         getServerInfoName(getServletContext().getServerInfo()));
     out.println("getServerInfo() version: " +
         getServerInfoVersion(getServletContext().getServerInfo()));
     out.println("getServletContext().getAttribute(\"attribute\"): " +
         getServletContext().getAttribute("attribute"));
   }
    private String getServerInfoName(String serverInfo) {
      int slash = serverInfo.indexOf('/');
      if (slash == -1) return serverInfo;
      else return serverInfo.substring(0, slash);
    }
    private String getServerInfoVersion(String serverInfo) {
     int slash = serverInfo.indexOf('/');
     if (slash == -1) return null;
     else return serverInfo.substring(slash + 1);
   }
}
```

Access to the client information

```
public String ServletRequest.getRemoteAddr()
public String ServletRequest.getRemoteHost()
public String HTTPServletRequest.getRemoteUser()
public String HTTPServletRequest.getAuthType()
```

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class PersonalizedWelcome extends HTTPServlet {
   Hashtable accesses = new Hashtable();
   public void doGet(HTTPServletRequest req, HTTPServletResponse res)
         throws ServletException, IOException {
      res.setContentType("text/html");
      PrintWriter out = res.getWriter();  // ... HTML ...
      String remoteUser = req.getRemoteUser();
      if (remoteUser == null) {
        out.println("Welcome!");
      }
      else {
        out.println("Welcome, " + remoteUser + "!");
        Date lastAccess = (Date) accesses.get(remoteUser);
        if (lastAccess == null) {
          out.println("This is the first visit!");
        }
        else {
          out.println("Your last visit was " + accesses.get(remoteUser));
        }
        if (remoteUser.equals("PROFESSOR FALKEN")) {
          out.println("Would you play with me?");
        }
        accesses.put(remoteUser, new Date());
      }          // ...etc. etc....
   }
}
```

How differentiate the browser types

- To differentiate among Firefox, Netscape, and Internet Explorer, check for "MSIE," not "Mozilla."

```
public class BrowserInsult extends HttpServlet {
  public void doGet(HttpServletRequest request, HttpServletResponse response)
                throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String title, message;
    // Assume for simplicity that Firefox and IE are the only two browsers.
    String userAgent = request.getHeader("User-Agent");
    if ((userAgent != null) && (userAgent.indexOf("MSIE") != -1)) {
      title = "Microsoft Minion";
       message = "Welcome, to Microsof Empire.";
    } else {
      title = "Hopeless Firefox Rebel ";
      message = "Enjoy it while you can . " + "Sarai <I>assimilato</I> !";
    }
    String docType = "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 " + "Transitional//EN\">\n";
    out.println (docType +
       "<HTML>\n" +
       "<HEAD><TITLE>"+title+"</TITLE></HEAD>\n"+
       "<BODY BGCOLOR=\"#FDF5E6\">\n" +
       "<H1 ALIGN=\"CENTER\">" + title + "</H1>\n" +
       "<BR>" + message + "<BR>\n" +
       "</BODY></HTML>");

  }
}
```

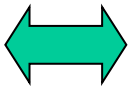
Generating the Server Response: HTTP Status Codes

- Format of the HTTP response
- How to set status codes
- What the status codes are good for
- Shortcut methods for redirection and error pages
- A servlet that redirects users to browser-specific pages
- A front end to various search engines

HTTP Request/Response

Request

```
GET /servlet/Nome HTTP/1.1
Host: ...
Header2: ...
...
HeaderN:
   (Blank Line)
```



Response

```
HTTP/1.1 200 OK
Content-Type: text/html
Header2: ...
...
HeaderN: ...
    (Blank Line)
<!DOCTYPE ...>
<HTML>
<HEAD>...</HEAD>
<BODY>
...
</BODY>
</HTML>
```

Common HTTP 1.1 Status Codes

`response.setStatus(int statusCode)`

- Use a constant for the code, not an explicit int. Constants are in `HttpServletResponse`
- Names derived from standard message. E.g., `SC_OK`, `SC_NOT_FOUND`, etc.

`response.sendError(int code, String message)`

- Wraps message inside small HTML document

`response.sendRedirect(String url)`

- Sets status code to 302
- Sets Location response header also.

200 (OK)

- Everything is fine; document follows.
- Default for servlets.

204 (No Content)

- Browser should keep displaying previous document.

301 (Moved Permanently)

- Requested document permanently moved elsewhere (indicated in Location header).
- Browsers go to new location automatically.

302 (Found)

401 (Unauthorized) etc.

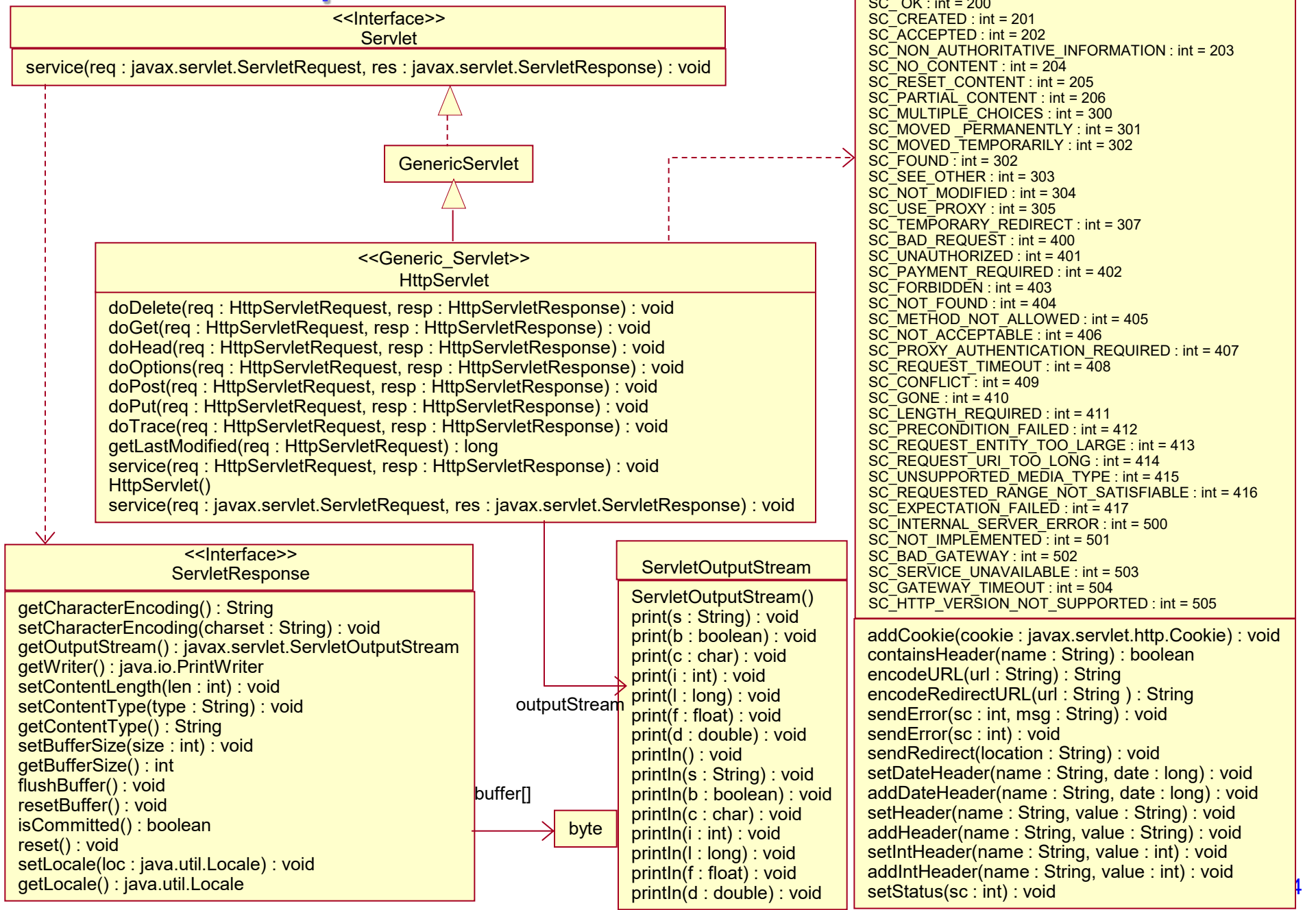
A Servlet That Redirects Users to Browser-Specific Pages

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class DestinazioneSbagliata extends HttpServlet {
public void doGet(HttpServletRequest request,
      HttpServletResponse response)throws ServletException, IOException {
    String userAgent = request.getHeader("User-Agent");
    if ((userAgent != null) && (userAgent.indexOf("MSIE") != -1)) {
       response.sendRedirect("http://home.netscape.com");
    } else {
       response.sendRedirect("http://www.microsoft.com");
    }
  }
}
```

Generating the Server Response

HTTP Response Generation



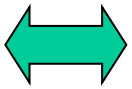
Generating the Server Response: HTTP Response Headers

- Format of the HTTP response
- Setting response headers
- Understanding what response headers are good for

HTTP Request/Response

Request

```
GET /servlet/Nome HTTP/1.1
Host: ...
Header2: ...
...
HeaderN:
   (Blank line)
```



Response

```
HTTP/1.1 200 OK
Content-Type: text/html
Header2: ...
...
HeaderN: ...
   (Blank line)
<!DOCTYPE ...>
<HTML>
<HEAD>...</HEAD>
<BODY>
...
</BODY>
</HTML>
```

Setting Common Response Headers

`public void setHeader(String headerName, String headerValue)`

Sets an arbitrary header `headerName`

`public void setDateHeader(String name, long millisecs)`

Converts milliseconds since 1970 to a date string in GMT format

`public void setIntHeader(String name, int headerValue)`

Prevents need to convert int to String before calling setHeader

- **`addHeader, addDateHeader, addIntHeader`**

Adds new occurrence of header instead of replacing

`setContentType`

Sets the Content-Type header. Servlets almost always use this. See table of common MIME types.

`setContentLength`

Sets the Content-Length header. Used for persistent HTTP connections. See Connection request header.

`addCookie`

Adds a value to the Set-Cookie header. See separate section on cookies.

`sendRedirect`

Sets the Location header (plus changes status code).

MIME Types

Type	Meaning
application/msword	Microsoft Word document
application/octet-stream	Unrecognized or binary data
application/pdf	Acrobat (.pdf) file
application/postscript	PostScript file
application/vnd.ms-excel	Excel spreadsheet
application/vnd.ms-powerpoint	Powerpoint presentation
application/x-gzip	Gzip archive
application/x-java-archive	JAR file
application/x-java-vm	Java bytecode (.class) file
application/zip	Zip archive
audio/basic	Sound file in .au or .snd format
audio/x-aiff	AIFF sound file
audio/x-wav	Microsoft Windows sound file
audio/midi	MIDI sound file
text/css	HTML cascading style sheet
text/html	HTML document
text/plain	Plain text
text/xml	XML document
image/gif	GIF image
image/jpeg	JPEG image
image/png	PNG image
image/tiff	TIFF image
video/mpeg	MPEG video clip
video/quicktime	QuickTime video clip

Common HTTP 1.1 Response Headers (I)

Cache-Control (1.1) and Pragma (1.0)

A no-cache value prevents browsers from caching page .

Content-Disposition

Lets you request that the browser ask the user to save the response to disk in a file of the given name.

```
Content-Disposition: attachment; filename=file-name
```

Content-Encoding

The way document is encoded.

Content-Length

The number of bytes in the response. Use `ByteArrayOutputStream` to buffer document before sending it, so that you can determine size.

Content-Type

The MIME type of the document being returned. Use `setContentType()` to set this header.

Expires

The time at which document should be considered out-ofdate and thus should no longer be cached. Use `setDateHeader()` to set this header.

Common HTTP 1.1 Response Headers (II)

Last-Modified

The time document was last changed. Don't set this header explicitly; provide a `getLastModified()` method instead.

Location

The URL to which browser should reconnect. Use `sendRedirect()` instead of setting this directly.

Refresh

The number of seconds until browser should reload page. Can also include URL to connect to.

Set-Cookie

The cookies that browser should remember. Don't set this header directly; use `addCookie()` instead.

WWW-Authenticate

The authorization type and realm needed in Authorization header.

Requirements for Handling Long-Running Servlets

A way to store data between requests.

- For data that is not specific to any one client, store it in a field (instance variable) of the servlet.
- For data that is specific to a user, store it in the HttpSession object
- For data that needs to be available to other servlets or JSP pages (regardless of user), store it in the ServletContext

A way to keep computations running after the response is sent to the user

- This task is simple: start a Thread. The only subtlety: set the thread priority to a low value so that you do not slow down the server.

A way to get the updated results to the browser when they are ready

- Use Refresh header to tell browser to ask for updates.

Handling Cookies

- Understanding the benefits and drawbacks of cookies
- Sending outgoing cookies
- Receiving incoming cookies
- Tracking repeat visitors
- Specifying cookie attributes
- Differentiating between session cookies and persistent cookies
- Simplifying cookie usage with utility classes
- Modifying cookie values
- Remembering user preferences

The Potential of Cookies

- A cookie is a message sent to a Web browser by a Web server. The cookie is stored in a text file on the local disk of the client-computer. The message can be sent to the Web server each time the client requests a server page.
- The cookies are used to provide information for the web server on the application status, allowing for instance the server to know who is the user.

Cookie and servlet

- Servlet sends a simple name and value to client.
- Client returns same name and value when it connects to same site (or same domain, depending on cookie settings).

Typical Uses of Cookies

- Identifying a user during an e-commerce session. Servlets have a higher-level API for this task
- Avoiding username and password
- Customizing a site
- Focusing advertising.

Some problems with cookies

- Servers can remember your previous actions
- If you give out personal information, servers can link that information to your previous actions
- Servers can share cookie information through use of a cooperating third party like doubleclick.net
- Poorly designed sites store sensitive information like credit card numbers directly in cookie

Handling Cookies

Sending cookies to the client

1. Create a Cookie object. Call the Cookie constructor with a cookie name and a cookie value, both of which are strings :
   `Cookie c = new Cookie("userID", "a1234");`
2. Set the maximum age for the cookie
   `c.setMaxAge(60*60*24*7);  // a week`
3. Place the Cookie into the HTTP response:
   Use `response.addCookie(c)`.

Reading cookies from the client

1. Call `request.getCookies()`. This yields an array of Cookie objects.
2. Loop down the array calling `getName()` on each entry until you find the cookie of interest. Use the value `getValue()` in application-specific way:

```
String cookieName = "userID";
Cookie[] cookies = request.getCookies();
if (cookies != null) {
  for(int i=0; i<cookies.length; i++) {
    Cookie cookie = cookies[i];
    if (cookieName.equals(cookie.getName())) {
       executeSomething(cookie.getValue());
    }
  }
}
```

Using Cookies to Detect First-Time Visitors

```
public class RepeatVisitor extends HttpServlet {
  public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    boolean nuovo = true;
    Cookie[] cookies = request.getCookies();
    if (cookies != null) {
      for(int i=0; i<cookies.length; i++) {
        Cookie c = cookies[i];
        if ((c.getName().equals("repeatVisitor")) && (c.getValue().equals("yes"))) {
           nuovo = false;
           break;
        }
      }
    }
    String title;
    if (nuovo) {
      Cookie cookieVisitatore = new Cookie("repeatVisitor", "yes");
      cookieVisitatore.setMaxAge(60*60*24*365);
      response.addCookie(returnVisitorCookie);
      titolo = "Welcome Aboard";
    } else {
      titolo = "Welcome Back";
    }
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    . . .  // (Output page with above title)
```

Using Cookie Attributes

`String getDomain()/setDomain(String)`

Lets you specify domain to which cookie applies. Current host must be part of domain specified.

`int getMaxAge()/setMaxAge(int)`

Gets/sets the cookie expiration time (in seconds). If you fail to set this, cookie applies to current browsing session only.

`String getName()`

Gets the cookie name. There is no setName method; you supply name to constructor. For incoming cookie array, you use getName to find the cookie of interest.

`String getPath()/setPath(String)`

Gets/sets the path to which cookie applies. If unspecified, cookie applies to URLs that are within or below directory containing current page.

`boolean getSecure/setSecure(boolean)`

Gets/sets flag indicating whether cookie should apply only to SSL connections or to all connections.

`String getValue/setValue(String)`

Gets/sets value associated with cookie. For new cookies, you supply value to constructor, not to setValue. For incoming cookie array, you use `getName()` to find the cookie of interest, then call `getValue()` on the result. If you set the value of an incoming cookie, you still have to send it back out with `response.addCookie`.

Setting Cookies

```
package course;
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
public class CookieSetting extends HttpServlet {
  public void doGet(HttpServletRequest request,HttpServletResponse response)
    throws ServletException, IOException {
    for(int i=0; i<3; i++) {
      Cookie cookie = new Cookie("Session-Cookie-" + i, "Cookie-Value-S" + i);
      // No maxAge (ie maxAge = -1)
      response.addCookie(cookie);
      cookie = new Cookie("Persistent-Cookie-" + i, "Cookie-Value-P" + i);
      // set maxAge
      cookie.setMaxAge(3600);
      response.addCookie(cookie);
    }

     … // Start an HTML table

    Cookie[] cookies = request.getCookies();
    if (cookies == null) {
         out.println("<TR><TH COLSPAN=2>No cookies");
    } else {
        for(Cookie cookie: cookies) {
           out.println
              ("<TR>\n" +
               " <TD>" + cookie.getName() + "\n" +
               " <TD>" + cookie.getValue());
         }
    }
    out.println("</TABLE></BODY></HTML>");
  }
}
```

Reading Cookies

```
package course;
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
public class CookieReading extends HttpServlet {
  public void doGet(HttpServletRequest request,HttpServletResponse response)
    throws ServletException, IOException {
    // Inizia una tabella HTML
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String titolo = "Cookie attivi";
    out.println
      (ServletUtilities.headWithTitle(titolo) +
       "<BODY BGCOLOR=\"#FDF5E6\">\n" +
       "<H1 ALIGN=\"CENTER\">" + titolo + "</H1>\n" +
       "<TABLE BORDER=1 ALIGN=\"CENTER\">\n" +
       "<TR BGCOLOR=\"#FFAD00\">\n" +
       "  <TH>Nome cookie\n" +
       " <TH>Valore cookie");
    Cookie[] cookies = request.getCookies();
    if (cookies == null) {
      out.println("<TR><TH COLSPAN=2>Non ci sono cookies");
    } else {
        Cookie cookie;
        for(int i=0; i<cookies.length; i++) {
          cookie = cookies[i];
          out.println
              ("<TR>\n" +
               " <TD>" + cookie.getName() + "\n" +
               " <TD>" + cookie.getValue());
     }
     out.println("</TABLE></BODY></HTML>");
  }
}
```

Cookie Utilities

```
public class LongLivedCookie extends Cookie {
  public static final int SECONDS_PER_YEAR = 60*60*24*365;
  public LongLivedCookie(String name, String value) {
    super(name, value);
    setMaxAge(SECONDS_PER_YEAR);
  }
}
```

```
package course;
public class CookieUtilities {
  public static String getCookieValue(HttpServletRequest request,
                                       String cookieName,
                                       String defaultValue) {
    Cookie[] cookies = request.getCookies();
    if (cookies != null) {
       for(int i=0; i<cookies.length; i++) {
          Cookie cookie = cookies[i];
          if (cookieName.equals(cookie.getName())) {
             return(cookie.getValue());
          }
       }
    }
    return(defaultValue);
  }
. . .
}
```

Using Cookies to Remember User Preferences

RegistrationForm servlet

- Uses cookie values to prepopulate form field values
- Uses default values if no cookies are found

Registration servlet

- Creates cookies based on request parameters received
- Displays values if all parameters are present
- Redirects to form if any parameter is missing

RegistrationForm Servlet

```
package course;
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
public class RegistrationForm extends HttpServlet {
  public void doGet(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String actionURL = "/servlet/course.ServletRegistrazione";
    String firstName = CookieUtilities.getCookieValue(request, "firstName", "");
    String secondName = CookieUtilities.getCookieValue(request, "lastName", "");
    String emailAddress = CookieUtilities.getCookieValue(request,"emailAddress","");
    String title = "Form di registrazione";
    out.println
       (docType +
        "<HTML>\n" +
        "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n" +
        "<BODY BGCOLOR=\"#FDF5E6\">\n" +
        "<CENTER>\n" +
        "<H1>" + title + "</H1>\n" +
        "<FORM ACTION=\"" + actionURL + "\">\n" +
        "First Name:\n" +
        " <INPUT TYPE=\"TEXT\" NAME=\"firstName\" " +
        "VALUE=\"" + firstName + "\"><BR>\n" +
        "Last Name:\n" +
        " <INPUT TYPE=\"TEXT\" NAME=\"lastName\" " +
        "VALUE=\"" + lastName + "\"><BR>\n"+
        "Email Address: \n" +
        " <INPUT TYPE=\"TEXT\" NAME=\"emailAddress\" " +
        "VALUE=\"" + emailAddress + "\"><P>\n" +
        "<INPUT TYPE=\"SUBMIT\" VALUE=\"Register\">\n" +
        "</FORM></CENTER></BODY></HTML>");
  }
}
```

Registration Servlet

```
package course;
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class RegistrationServlet extends HttpServlet {
public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
  response.setContentType("text/html");
  boolean isMissingValue = false;
  String firstName = request.getParameter("firstName");
  if (isMissing(firstName)) {
    firstname = "Missing first name";
    isMissingValue = true;
  }
  String lastName = request.getParameter("lastName");
  if (isMissing(lastName)) {
    lastName = "Missing last name";
    isMissingValue = true;
  }
   . . .
  Cookie c1 = new LongLivedCookie("firstName", firstName);
  response.addCookie(c1);
  Cookie c2 = new LongLivedCookie("lastName", lastName);
  response.addCookie(c2);
  Cookie c3 = new LongLivedCookie("emailAddress", emailAddress);
  response.addCookie(c3);
  String formAddress = "/servlet/course.RegistrationServlet ";
  if (isMissingValue) {
    response.sendRedirect(formAddress);
  } else {
   . . .
  }
```

Information Sharing

Information Sharing

Web components must collaborate with each other. For this they can use the following possibilities:

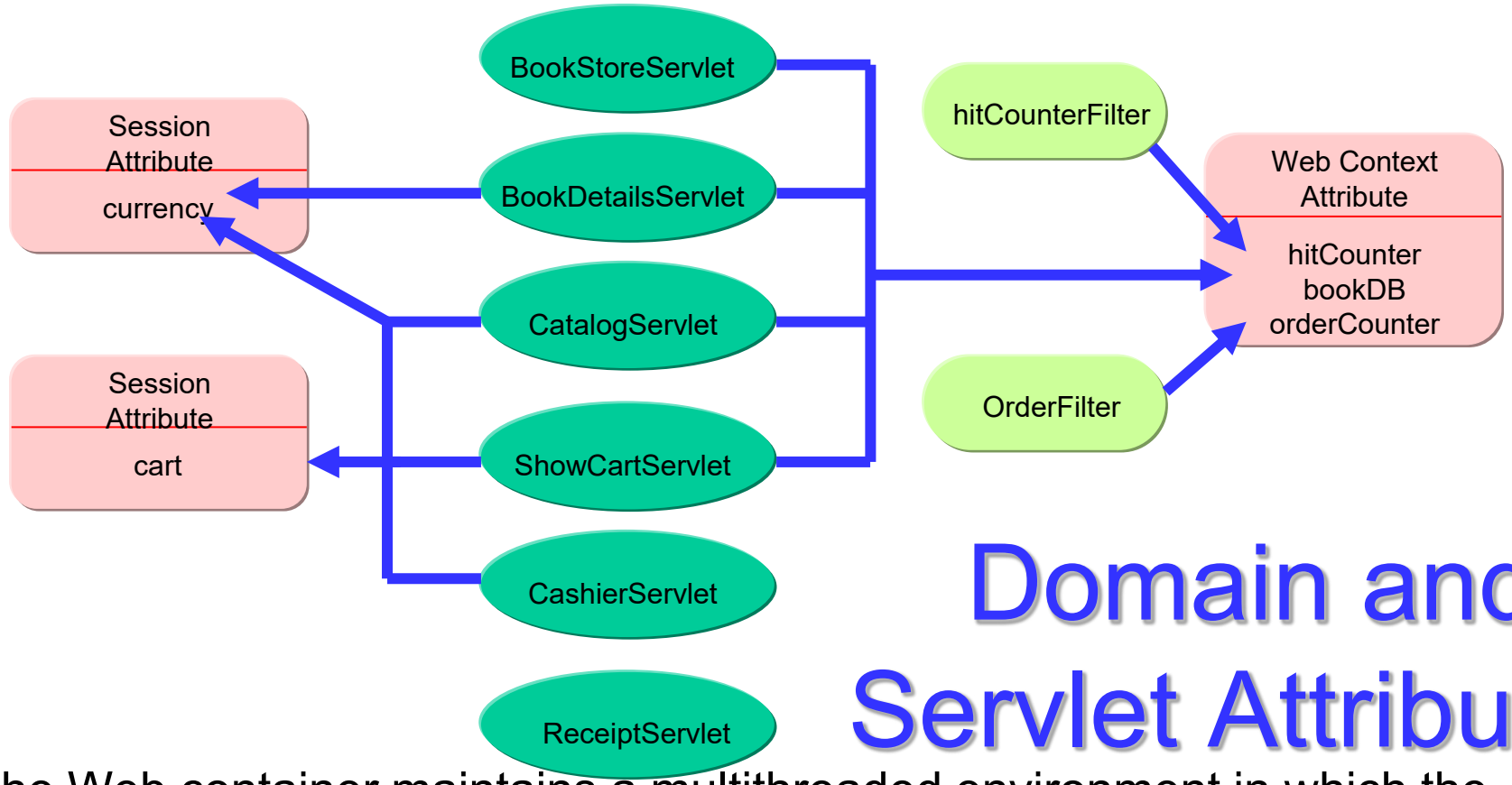
- Inter-component calls;
- Private helper objects as JavaBeans components;
- Public scope attributes which are objects stored as attributes in the various domains;
- The same DB tables;
- Direct or indirect calls to other web resources:
  - Direct call: during the servlet execution other resource can be dinamically included and a call to it can be issued;
  - Indirect call: in the servlet response a URL wich refers an other component is included.

Public Scopes

Public Scope (Domains) are objects where we can store as attributes other objects that become shared with all which access the domain.

This attributes are accessed with [get/set]Attribute methods of the class that represents the domain.

Scope	Class	Access from
Web context	javax.servlet.ServletContext	Web components in a web context.
Session	javax.servlet.HttpSession	Web components which process a request belonging to a session.
Request	sub-class of javax.servlet.ServletRequest	Web components which process a request
Page	javax.servlet.jsp.JspContext	The JSP page where the object is created



Domain and Servlet Attributes

The Web container maintains a multithreaded environment in which the resources shared can be accessed at the same time. Beyond that attribute-objects of the domains, these resources can be represented as instance- or class-variables, external objects like files, connections to the database or the net. The situations can be the followings:

- Web components which access to objects stored in the Web context;
- Web components which access objects stored in the Session object;
- Web components which access instance-variables of the servlet.

Note. A Web container creates a thread for each request.

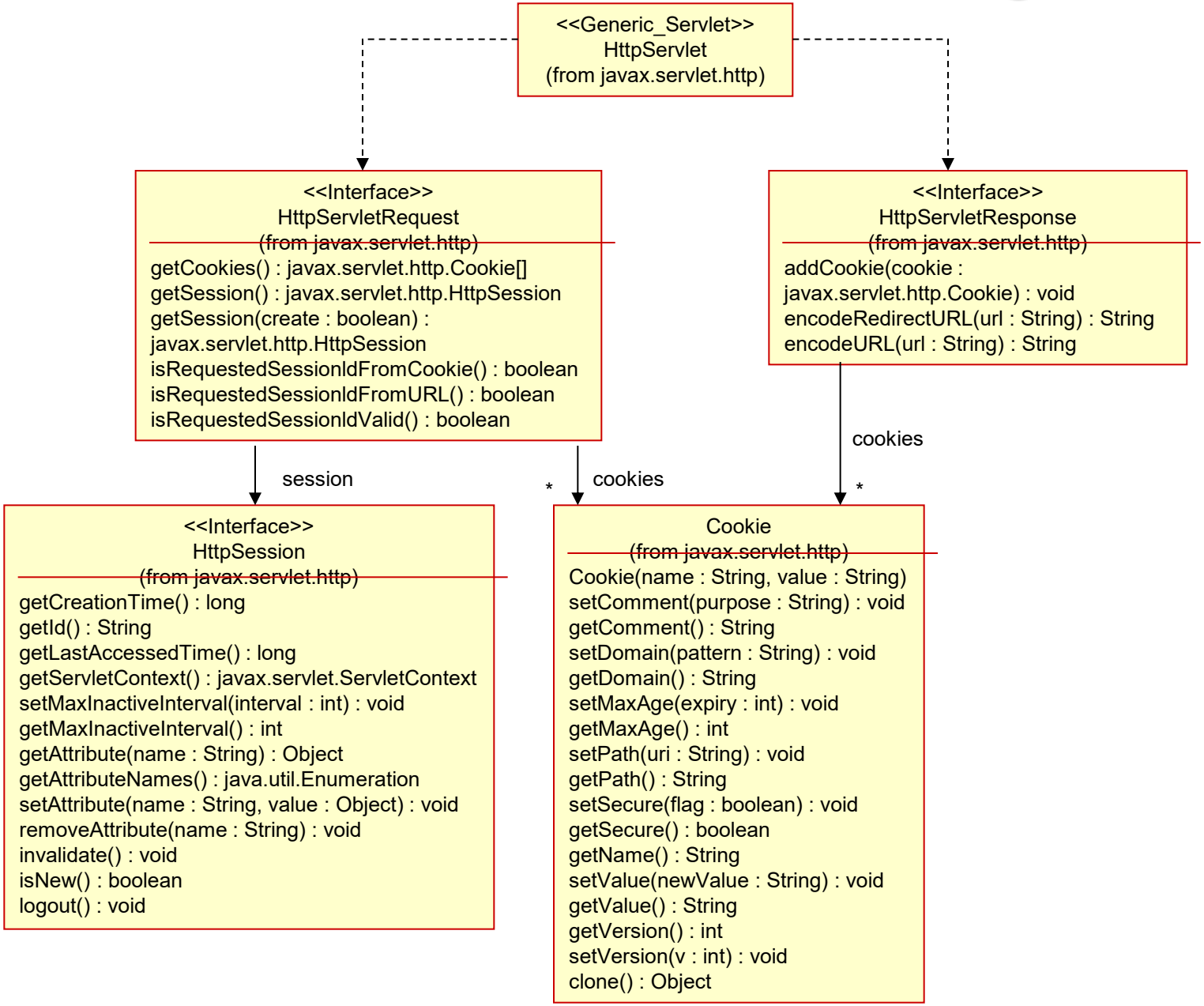
Shared Access to Database

```
public void buyBooks(ShoppingCart cart) throws OrderException {
   Collection items = cart.getItems();
   Iterator i = items.iterator();
   try {
     getConnection();
     con.setAutoCommit(false);
     while (i.hasNext()) {
       ShoppingCartItem sci = (ShoppingCartItem)i.next();
       BookDetails bd = (BookDetails)sci.getItem();
       String id = bd.getBookId();
       int quantity = sci.getQuantity();
       buyBook(id, quantity);
      }
     con.commit();
     con.setAutoCommit(true);
     releaseConnection();
   } catch (Exception ex) {
     try {
      con.rollback();
      releaseConnection();
      throw new OrderException("Transaction failed: "+ ex.getMessage());
     } catch (SQLException sqx) {
       releaseConnection();
       throw new OrderException("Rollback failed: "+ sqx.getMessage());
     }
```

Session Tracking

- Implementing session tracking from scratch
- Using basic session tracking
- Tracking user access counts
- Accumulating user purchases
- Implementing a shopping cart

HTTP Session Management



Needs of client tracking

HTTP is a “stateless” protocol. At the end of a request-response cycle the server doesn’t keep contextual information of the client.

For e-commerce applications it is important for the server to know its clients and follow up them until the end of the transaction → client tracking.

There are three solutions for vlient tracking:

1. Cookie
2. URL modification
3. Hidden fields in the form.

Using cookies for client tracking

Associate cookie with data on server.

- In a online shopping session each HTTP connection can identify in the current session and then extract information on the session :

```
String sessionID = makeUniqueString();
HashMap sessionInfo = new HashMap();
HashMap globalTable = findTableStoringSessions();
globalTable.put(sessionID, sessionInfo);
Cookie sessionCookie = new Cookie("JSESSIONID", sessionID);
sessionCookie.setPath("/");
response.addCookie(sessionCookie);
```

- In the successive requests the server can use a `globalTable` to extract the hashtable `sessionInfo` with the ID found in the cookie `JSESSIONID`.
- To work with cookies there are other aspects to consider:
  - Extracting cookie that stores session identifier
  - Setting appropriate expiration time for cookie
  - Associating the hash tables with each request
  - Generating the unique session identifiers.

URL-Rewriting

- Client appends some extra data on the end of each URL that identifies the session
- Server associates that identifier with data it has stored about that session
- Exemple: `http://host/path/file.html;jsessionid=1234` the string `jsessionid=1234` was associated to the server.

Advantage

Works even if cookies are disabled or unsupported.

Disadvantages

- Must encode all URLs that refer to your own site
- All pages must be dynamically generated
- Fails for bookmarks and links from other sites

Hidden Form Fields

Include in the form such an element:

```
<INPUT TYPE="HIDDEN" NAME="session" VALUE="...">
```

When the form is sent to the server the hidden element name and value can be extracted to access the information on the session.

Advantage

Works even if cookies are disabled or unsupported

Disadvantages

- Lots of tedious processing
- All pages must be the result of form submissions

Session Tracking Steps

1. Obtain the Session object associated to the user.

- Session objects live on the server and can be accessed in the servlet with `request.getSession(true)`. They are hashtables. Behind the scenes, the system looks at cookie or URL extra info and sees if it matches the key to some previously stored session object. If so, it returns that object. If not, it creates a new one, assigns a cookie or URL info as its key, and returns that new session object.
  With `isNew()` we can interrogate if the object is new or old.

2. Access information in the Session object with `setAttribute()` , `getAttribute()`

```
HttpSession session = request.getSession();
SomeClass val = (SomeClass)session.getAttribute("someID");
if (val == null) {
  val  = new SomeClass(...);
  session.setAttribute("someID", val );
}
doSomethingWith(val );
```

3. Discard session data
   - Call `removeAttribute(String)` to discard a specific value.
   - Call `invalidate()` to discard an entire session
   - Call `logout()` to close all sessions associated to this client.

Show Access Counts

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.net.*;
imoprt java.util.*;
public class ShowSessione extends HttpServlet {
  public void doGet(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");
    HttpSession session = request.getSession();
    String heading;
    Integer accessCount = (Integer)session.getAttribute("accessCount");
    if (accessCount == null) {
       accessCount = new Integer(0);
       heading = "Welcome, Newcomer!";
    } else {
        heading = "Welcome Back!";
       accesCount = new Integer(accessCount.intValue() + 1);
    }
    session.setAttribute("accessCount", accessCount);
    PrintWriter out = response.getWriter(); . . . . .
    out.println
        (ServletUtilities.headWithTitle(title) +
         "<BODY BGCOLOR=\"#FDF5E6\">\n" +
         "<CENTER>\n" +
         "<H1>" + heading + "</H1>\n" +
         "<H2>Information on Your Session:</H2>\n" +
         "<TABLE BORDER=1 ALIGN=\"CENTER\">\n" +
         "<TR BGCOLOR=\"#FFAD00\">\n" +
         " <TH> Info Type<TH>Value\n" +
         " <TR>\n " +
         " <TD>ID\n " +
         " <TD> " + session.getId() + "\n" +
         " <TR>\n " +
         " <TD>Creation data\n " +
         " <TD> " + new Date(session.getCreationTime()) + "\n" +
         " <TR>\n " +
         " <TD>Last access\n " +
         " <TD> " + new Date(session.getLastAccessedTime()) + "\n" +
         " <TR>\n " +
         " <TD>Number of previous accesses\n" +
         " <TD>" + accessCount + "\n" +
         "</TABLE>\n" +
         "</CENTER></BODY></HTML>");
```

Accumulating a List of User Data

```
public class ShowItems extends HttpServlet {
  public void doGet(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
    HttpSession session = request.getSession();
    ArrayList<String> previousItems =
        (ArrayList<String>)session.getAttribute("previousItems");
    if (previousItems == null) {
       previousItems = new ArrayList();
       session.setAttribute("previousItems", previousItems);
    }
    String newItem = request.getParameter("newItem");
    if ((newItem != null) && (!newItem.trim().equals(""))) {
      previousItems.add(newItem);
    }
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String title = "Items Purchased";
    String docType =
        "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 "+"Transitional//EN\">\n";
        out.println(docType+"<HTML>\n"+"<HEAD><TITLE>"+title +"</TITLE></HEAD>\n"+
                    "<BODY BGCOLOR=\"#FDF5E6\">\n" + "<H1>" + title + "</H1>");
    if (previousItems.size() == 0) {
       out.println("<I>No items</I>");
    } else {
       out.println("<UL>");
       for(String item: previousItems) {
         out.println(" <LI>" + item);
       }
       out.println("</UL>");
    }
    out.println("</BODY></HTML>");
  }
}
```

Session Management. Customer Registration

```
private String registerNewCustomer(HttpServletRequest servletRequest) throws ServletException {
  // Code for information extraction from servletRequest :
  // firstName, lastName, middleName, password, phone, eMail,
  // address_1, address_2, city, state, zip_code
  ...
  // Create an Address object from POST data
  Address address = new Address(address_1, address_2, city, state,zip_code);
  // Create a Customer object from POST data and the Address object
  Customer customer = new Customer(firstName,lastName,middleName,address,phone,eMail,password);
  // Ask for the session object and add the Customer object to the session
  HttpSession session = servletRequest.getSession();
  session.setAttribute(Customer.CUSTOMER_OBJECT, customer);
  // Returns the user name
  return firstName;
}
public void doPost(HttpServletRequest servletRequest, HttpServletResponse servletResponse)
   throws ServletException, IOException {
  ...
  // Create a Customer object from POST data
  String firstName = registerNewCustomer(servletRequest);
  // Obtain the session object e il value of the cookie NAME
  HttpSession session = servletRequest.getSession(false);
  String cookieValue = (String)session.getAttribute(ServletsHelper.COOKIE_NAME);
  // If the cookie is missing, add the cookie name
  if(cookieValue == null){
    // create a Cookie for this customer
    Cookie cookie = new Cookie(ServletsHelper.COOKIE_NAME, firstName);
    cookie.setMaxAge(SIX_MONTHS_IN_SECONDS);
    servletResponse.addCookie(cookie);
  }
```

Session Management. Customer Retrieval

Another servlet needs to know the component it communicates with and call `manageSession()`

```
public void service (HttpServletRequest servletRequest, HttpServletResponse servletResponse)
throws ServletException, IOException {
  ...
  // Extract info on session
  String sessionCookie = manageSession(servletRequest);
  ...
}
private String manageSession(HttpServletRequest servletRequest) {
  // Create a session any time the user calls this site.
  HttpSession session = servletRequest.getSession(true);
  // Obtain the servlet request cookies
  Cookie[] cookies = servletRequest.getCookies();
  // Obtain the value if the cookie BEESHIRTS exists, otherwise return null
  String cookieValue = ServletsHelper.getOurCookieValue(cookies);
  System.out.println("Cookie Value :" +cookieValue);
  // If the cookie value exists, add it to the HttpSession object
  if(cookieValue != null){
    session.setAttribute(ServletsHelper.COOKIE_NAME, cookieValue);
  }
  // Return the cookie value
  return cookieValue;
}
// the method uses the cookie value for customized message visualization
private void writePageCenterArea(PrintWriter printWriter, String cookieValue) {
  ...
  if(cookieValue != null){
    printWriter.println(" <STRONG> WELCOME BACK : " + cookieValue +"</STRONG> <BR>");
  }
  ...
}
```

Servlet Event Handling



Servlet Events

- The management of the events during the life of a servlet allows the container to notify to the application the occurrence of such events. The management is obtained by defining the objects listeners whose methods are invoked when events occur. The application can receive notification on the initialization and the destruction of the servlet context or if an attribute of the ServletContext or the ServletRequest has been added, removed or modified. In order to receive events:
  - A class is defined that implements the interface of the event and
  - The event listener class must be recorded in the DD .

Object	Event	Interface Listener	Event Class
Web Context	servlet initialization servlet destruction	ServletContextListener	ServletContextEvent
Web Context	Add, remove, attribute substitution	ServletContexAttributeListener	ServletContextAttributeEvent
Session	Creation invalidation activation passivation timeout	HttpSessionListener HttpSessionActivationListener	HttpSessionEvent
Session	Add, remove, attribute substitution	ServletSessionAttributeListener	HttpSessionBindingEvent
Request	A request to a servlet was initiated	HttpRequestListener	HttpRequestEvent
Request	Add, remove, attribute substitution	ServletRequestAttributeListener	HttpRequestAttributeEvent

Context Listener

```
public interface ServletContextListener extends java.util.EventListener{
   /* Notifies that the initialization of the web application is to be initiated.
   The ServletContextListeners are notified before the filters and servlets are
   initialized. */
   void contextInitialized (ServletContextEvent sce);
   /* Notifies that the servlet context is to be destroyed. All the servlets and
   filters were eliminated. */
   void contextDestroyed (ServletContextEvent sce);
}

public interface ServletContextAttributeListener extends java.util.EventListener {
   /* Notifies that a new attribute was added to the servlet context. It is called
   after the attribute was added. */
   void attributeAdded(ServletContextAttributeEvent scab);
   /* Notifies that an existent attribute was removed  from the servlet context. It
   is called after removing. */
   void attributeRemoved(ServletContextAttributeEvent scab);
   /* Notifies that an attribute of the servlet context was substitued. It is called
   after the substitution. */
   void attributeReplaced(ServletContextAttributeEvent scab);
}
```

Session Listeners

```
public interface HttpSessionListener extends java.util.EventListener {
   /* Notifies that a session was created. */
   void sessionCreated(HttpSessionEvent se) ;
   /* Notifies that a session is to be invalidated. */
   void sessionDestroyed(HttpSessionEvent se);
}

public interface HttpSessionAttributeListener extends java.util.EventListener{
   /* Notifies that a new attribute was added to the session. It is called after
   the attribute was added. */
   void attributeAdded (HttpSessionBindingEvent  se);
   /* Notifies that an existing attribute  was removed from the session. It is
   called after removing. */
   void attributeRemoved(HttpSessionBindingEvent se);
   /* Notifies that a session attribute was substituted. It is called after the
   substitution. */
   void attributeReplaced(HttpSessionBindingEvent se);
}
```

Request Listeners

```
  /* A ServletRequestListener can be implemented when a deployer is interested to
  be notified on the requests in arrival or in output from the domain of the web
  component. A request is in arrival in the domain when it is to enter in the first
  servlet or filter in the application web and to output from the domain when it is
  to exit from the last servlet or filter in the chain of filters. */
public interface ServletRequestListener {
  /* Notifies that the request is to enter in the domain of the web application. */
  void requestInitialized(ServletRequestEvent rre) ;
  /* Notifies that the request is to exit from the domain of the web application.*/
  void requestDestroyed(ServletRequestEvent rre);
}

  /* A ServletRequestAttributeListener can be implemented when the deployer is
  interested to be notified from the modifications of the attributes of the
  requests. The notifications will be generated while the request is in the domain
  of the application web in which the listener it is registered. */
public interface ServletRequestAttributeListener {
  /* Notifies that a new attribute was added to the servlet request. It is called
  after the attribute was added. */
  void attributeAdded(ServletRequestAttributeEvent  srae);
  /* Notifies that an existing attribute was removed from the servlet request. It
  is called after removing. */
  void attributeRemoved(ServletRequestAttributeEvent se);
  /* Notifies that an attribute was substitued in the servlet request. It is called
  from the substitution. */
  void attributeReplaced(ServletRequestAttributeEvent se);
}
```

How a context listener is defined?

The program creates and eliminate the object that accesses the database and the application counters when the context is created and respectively eliminated.

```
package listeners;
 import database.BookDB;
 import javax.servlet.*;
 import util.Counter;
 public final class ContextListener implements ServletContextListener {
   private ServletContext context = null;
   public void contextInitialized(ServletContextEvent event) {
     context = event.getServletContext();
     try {
       BookDB bookDB = new BookDB();
       context.setAttribute("bookDB", bookDB);
     } catch (Exception ex) {
       System.out.println("Couldn't create bookstore database bean: " +
          ex.getMessage());
     }
     Counter counter = new Counter();
     context.setAttribute("hitCounter", counter);
     counter = new Counter();
     context.setAttribute("orderCounter", counter);
   }
   public void contextDestroyed(ServletContextEvent event) {
     context = event.getServletContext();
     BookDB bookDB = (BookDB)context.getAttribute("bookDB");
     if (bookDB != null) bookDB.remove();
     context.removeAttribute("bookDB");
     context.removeAttribute("hitCounter");
     context.removeAttribute("orderCounter");
   }
 }
```

How a ContextListener is Declared?

The context listeners must be configured in the deployment descriptor for the Web container to realize that these listeners exist. This is done using the `listener` element tag.

```
<web-app>
    <display-name>SL-314 Serlvet Context Example . . .</display-name>
    <description>
        This Web Application demonstrates application-scoped variables
        (in the servlet context) and WebApp lifecycle listeners.
   </description>
    <!-- Initialize Catalog -->
    <context-param>
       <param-name>catalogFileName</param-name>
       <param-value>/WEB-INF/catalog.txt</param-value>
    </context-param>
    <listener>
       <listener-class>sl314.InitializeList</listener-class>
     </listener>
    <!-- END: Initialize Catalog -->
```

The listener elements must come after any context-param elements and before any servlet definitions. You can declare any number of listeners. The order in the deployment descriptor determines the order in which the listeners are invoked by the Web container; at shutdown, that order is reversed. During Web application startup, all listeners will be invoked and completed before any servlet requests are processed.

Servlet initialization

After a servlet loading and instantiation and before requests processing the web server must initialize the servlet. This initialization can be customized by overriding the init() method of the HttpServlet class, for instance by allowing the servlet to:

- read configuration data
- initialize the resources
- do other activities.

```
public class CatalogServlet extends HttpServlet {
   private BookDB bookDB;
   public void init() throws ServletException {
     bookDB = (BookDB)getServletContext().getAttribute("bookDB");
     if (bookDB == null)
       throw new UnavailableException("Couldn't get database.");
   }
 }
```

A persistent HitCounter (I)

How to track the number of times when a page was accessed ?
The program saves a counter in a file before the servlet destruction and restores it when the file is reloaded .

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class InitDestroyCounter extends HttpServlet {
  int count;
  public void init() throws ServletException {
    // Tentativo di caricamento del contatore dal file
    FileReader fileReader = null;
    BufferedReader bufferedReader = null;
    try {
      fileReader = new FileReader("propfile");
      bufferedReader = new BufferedReader(fileReader);
      String initial = bufferedReader.readLine();
      count = Integer.parseInt(initial);
      return;
    } catch (FileNotFoundException ignored) { }  // lo stato non e' salvato
    catch (IOException ignored) { }              // un problema durante lettura
    catch (NumberFormatException ignored) { }  // stato inquinato
    finally {                                  // Chiudere il file!
      try {
        if (bufferedReader != null) {
          bufferedReader.close();
        }
      } catch (IOException ignored) { }
    }    // Informazioni non ripristinabili, prova con un parametro iniziale
      String initial = getInitParameter("initial");
      try {
           count = Integer.parseInt(initial);
           return;
      }  catch (NumberFormatException ignored) { }  // valore null o non-intera
      // Default "0"
      count = 0;
    }
```

A persistent HitCounter (II)

```
public void doGet(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {
  res.setContentType("text/plain");
  PrintWriter out = res.getWriter();
  count++;
  out.println("Dal inizio, questo servlet e' stato acceduto "+count+" volte.");
}

public void destroy() {
  super.destroy();  // opzionale
  saveState();
}

public void saveState() {
  // Tenta di salvare il contatore
  FileWriter fileWriter = null;
  PrintWriter printWriter = null;
  try {
     fileWriter = new FileWriter("propfile");
     printWriter = new PrintWriter(fileWriter);
     printWriter.println(count);
     return;
  } catch (IOException e) {  // un problema durante la scrittura
  // Log l'eccezione.
  } finally {                // Chiudere il file
     if (printWriter != null) {
       printWriter.close();
     }
  }
 }
```

JSP Programming

Introduction to JSP

- Understanding the need for JSP
- Evaluating the benefits of JSP
- Comparing JSP to other technologies
- Avoiding JSP misconceptions
- Understanding the JSP lifecycle

Why JSP?

With servlets, it is easy to:

- Read form data
- Read HTTP request headers
- Set HTTP status codes and response headers
- Use cookies and session tracking
- Share data among servlets
- Remember data between requests
- Get fun, high-paying jobs

- But it is difficult:
  - Use those println statements to generate HTML
  - Maintain that HTML
- JSPs do nothing more than servlets but with them is easier:
  - Write, read and maintain HTML pages
- JSP programmers still need to know servlet programming

JSP makes it possible to:

- Use standard tools such as Macromedia DreamWeaver, Adobe GoLive.
- Have different members of your team do the HTML layout than do the Java programming

- JSP encourages programmers to separate the (Java) code that creates the content from the (HTML) code that presents it

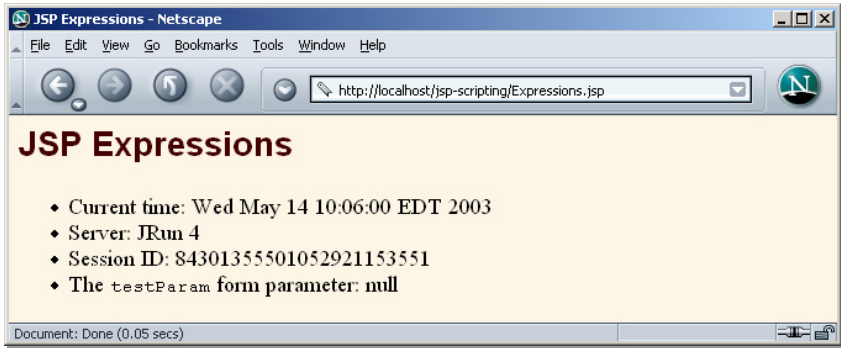
Executing JSPs

- A JSP ia a HTML page written with both HTML code and Java code. The Java code is included between `<%` and `%>` and is called *scriptlet*.
- The file extension is `.jsp` and asks for special processing in the web server environment.

1. Let consider the following code in a file

`<TOMCAT_HOME>/webapps/ROOT/jsp-scripting/Expressions.jsp`

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>JSP Expressions</TITLE>
<META NAME="keywords" CONTENT="JSP,expressions,JavaServer Pages">
<META NAME="description" CONTENT="A quick example of JSP expressions.">
<LINK REL=STYLESHEET HREF="JSP-Styles.css" TYPE="text/css">
</HEAD>
<BODY>
<H2>JSP Expressions</H2>
<UL>
<LI>Current time: <%= new java.util.Date() %>
<LI>Server: <%= application.getServerInfo() %>
<LI>Session ID: <%= session.getId() %>
<LI>The <CODE>testParam</CODE> form parameter:
<%= request.getParameter("testParam") %>
</UL>
</BODY></HTML>
```



2. Call your browser with the following URL resource:

`http://localhost:8080/jsp-scripting/Expressions.jsp`

Write a string in the HTML page

With `<%@page import=...%>` we can import packages ;
With the `out` object we represent the output stream that is the generated HTML page. This way we can write directly in the HTML page.

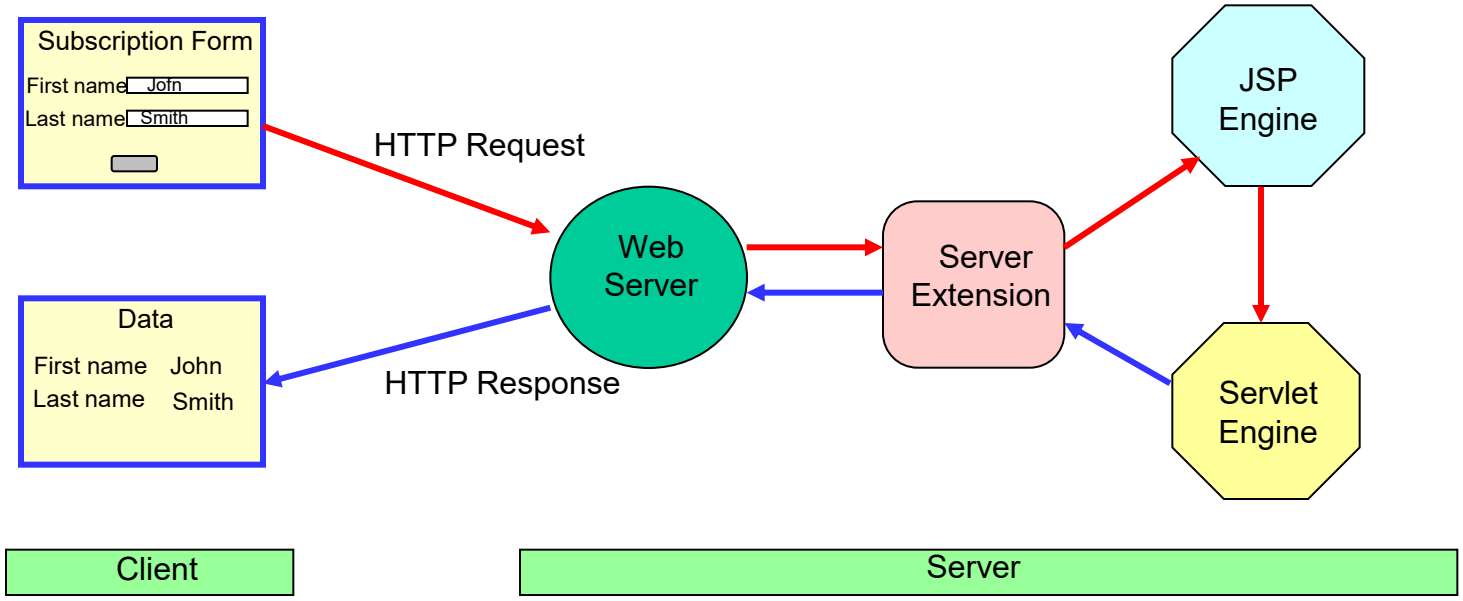
```
<%@page import="java.util.*" %>
<HTML>
<HEAD>
<TITLE>JSP Exemple</TITLE>
</HEAD>
<BODY BGCOLOR="#ffeecc">
<CENTER><H2>Date/Hour</H2>
<%  Date now = new Date();
  out.println("Current date: " + now);
%>
</CENTER>
</BODY>
</HTML>
```

`<%= . . . %>` inserts a string in the output:
Alternative code:

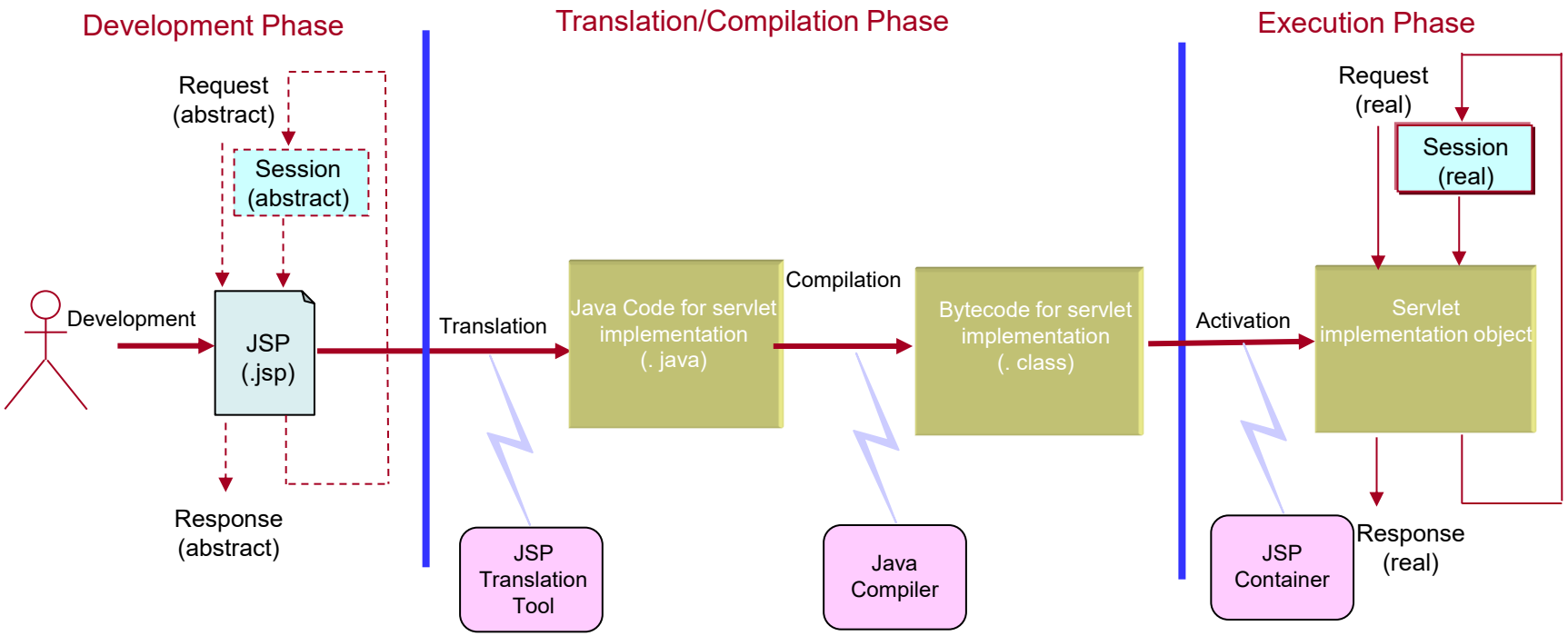
```
<%@page import="java.util.*" %>
<HTML>
<HEAD>
<TITLE>JSP Exemple </TITLE>
</HEAD>
<BODY BGCOLOR="ffeecc">
<CENTER><H2>Date/Hour</H2>
Current date: <%= new Date() %>
</CENTER>
</BODY>
</HTML>
```

What happens at page execution time?

1. When a JSP page is called for the first time, the JSP engine translates it in a servlet class definition .
2. The .java file is compiled in a .class file that is loaded in the web container JVM by a class loader.
3. The container instantiates the class and executes the servlet object initialization by calling `jspInit()`
4. The container calls `_jspService()` in order to respond to requests .
5. The successive calls use the instantiated servlet.
6. For removing the servlet instance the container calls `jspdestroy()`.



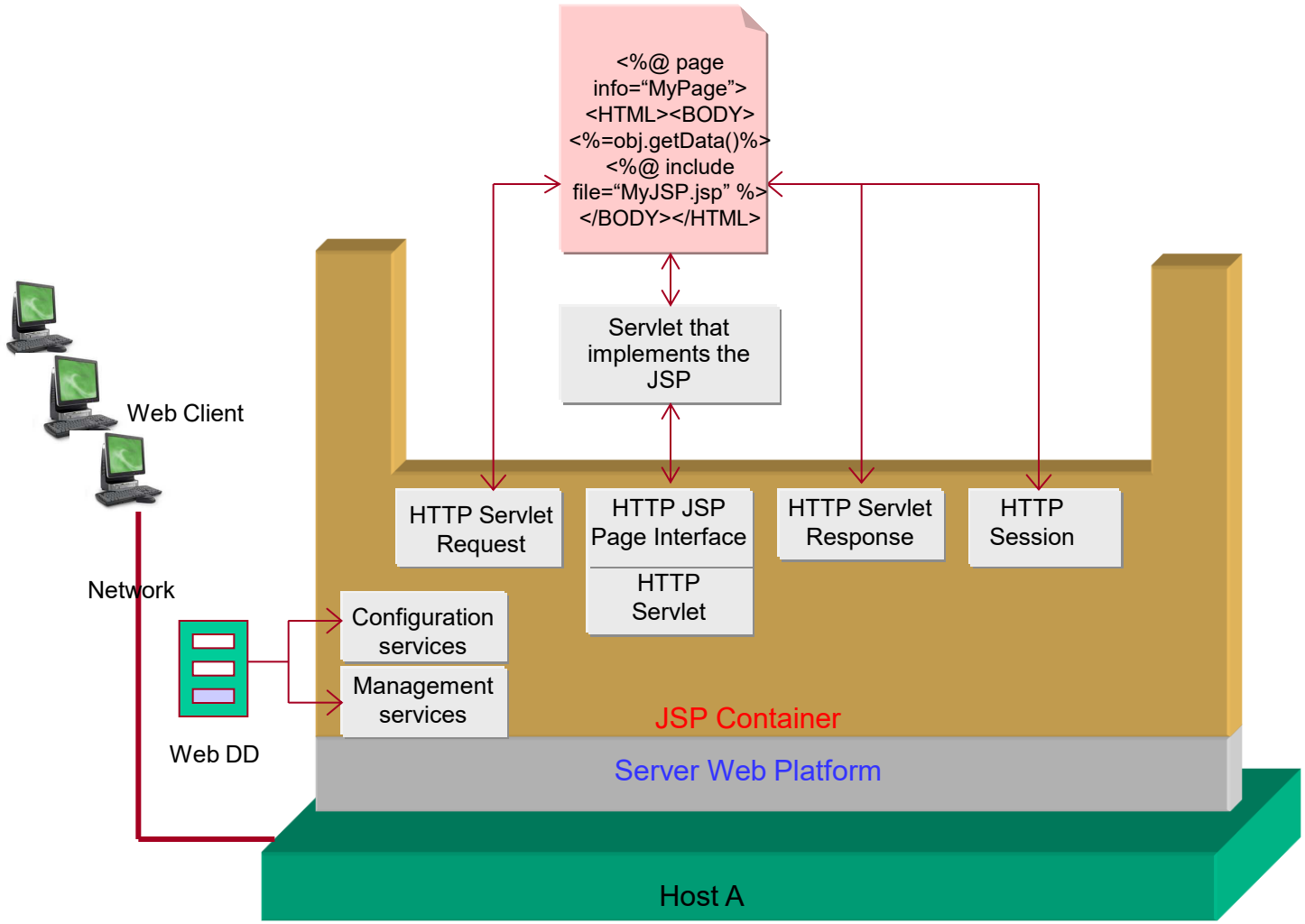
Phases in JSP processing



The JSP Lifecycle

	Page first written	Request #1	Request #2	Server restarted	Request #3	Request #4	Page modified	Request #5	Request #6
JSP is translated in a servlet		Yes	No		No	No		Yes	No
Servlet compiled		Yes	No		No	No		Yes	No
Servlet instantiated and loaded into memory		Yes	No		Yes	No		Yes	No
init() or equivalent called		Yes	No		Yes	No		Yes	No
doGet() or equivalent called		Yes	Yes		Yes	Yes		Yes	Yes

JSP Architecture



Using JSPs

Simple Applications



Complex Applications

- Scripting elements calling servlet code directly
- Scripting elements calling servlet code indirectly (by means of utility classes)
- JavaBeans
- Servlet/JSP combo (MVC)
- MVC with JSP expression language
- Custom tags
- MVC with beans, custom tags, and a framework like Struts or JSF

Design strategy

Minimize the Java code in JSPs. It is better to create separate Java classes and use them in few JSP lines.

Advantages

- Development: You write the separate class in a Java environment (editor or IDE), not an HTML environment
- Debugging. If you have syntax errors, you see them at compile time.
- Testing. You can write a test routine with a loop that does many test and retry for any code change.
- Reuse. You can use the same class from multiple pages.

JSP Page Structure

Basic Components

- "Template" Data : HTML or XML content statically included in a JSP.
- Directives: JSP elements interpreted at compile time and used to:
  - indicate what files should be included during compilation,
  - define attributes in the JSP page,
  - define libraries of customized elements to be used in the page.
- Scripting elements : JSP elements used to introduce:
  - Variable declarations;
  - Method declarations,
  - Expressions for evaluation,
  - scriptlet: blocks of commands written in a language supported by container (usually Java),
- Active elements: commands similar to tags, but implemented in a transparent manner for the programmer in Java. Some "*standard actions*" are used to send requests to other resources o search and create Bean objects needed in the JSP. We can define customized actions by using tag extensions.
- From the active and scripting elements in the page the JSP programmer can access directly to the Java objects.

Scripting elements

1. Use **<% *java code* %>** for inclusion of valid Java code (*scriptlet*):

```
<%  String queryData = request.getQueryString();
  out.println("Query data: " + queryData);  %>
```

2. Use **<%! *java code* %>** for declaring variables and methods. Example:

```
<%! int x = 10; double y = 2.0; %>
```

   The declarations are directly inserted in the servlet code.

3. Use **<%=*expression* %>** for defining an expression, casting a result as a String and inserting it in the HTML page. Example:

```
<%= a+b %>
Current date: <%= new java.util.Date() %>
Host: <%= request.getRemoteHost() %>
```

4. Use the page directive **<%@page ... %>** for defining the scripting language. Example:

```
<%@page language="java" import="java.util.*" %>
```

5. Use the directive **<%@ include ... %>** for inserting in the main file the content of the other files. Example:

```
<%@include file="copyright.html" %>
```

6. XML compatible syntax. We can write JSP pages in XML:

```
<jsp:expression>espressione Java</jsp:expression>
<jsp:scriptlet>Java code </jsp:scriptlet>
<jsp:declaration>Java declarations</jsp:declaration>
```

JSP Expressions

Syntax: `<%= Java Expression %>`

Result: Expression evaluated, converted to String, and placed into HTML page at the place it occurred in JSP page.

By using JSP expressions in a JSP page we can:

- Write HTML code;
- Use user and standard Java classes;
- Access to `Session`, `Application` objects or to the request parameters.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
  <HEAD>
    <TITLE>Espressioni JSP </TITLE>
    <META NAME="keywords" CONTENT="JSP,expressions,JavaServer Pages">
    <META NAME="description" CONTENT="Un esempio semplice di espressioni JSP.">
    <LINK REL=STYLESHEET HREF="JSP-Styles.css" TYPE="text/css">
  </HEAD>
  <BODY>
    <H2>Espressioni JSP</H2>
    <UL>
      <LI>Data/ora corrente: <%= new java.util.Date() %>
      <LI>Server: <%=application.getServerInfo() %>
      <LI>ID Sessione: <%= session.getId() %>
      <LI>Il parametro <CODE>testParam</CODE> del form:
      <%= request.getParameter("testParam") %>
    </UL>
  </BODY>
</HTML>
```

JSP ←→ servlet

JSP Code:

```
<H1>A Random Number</H1>
<%= Math.random() %>
```

Representative resulting servlet code:

```
public void _jspService(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
      response.setContentType("text/html");
      HttpSession session = request.getSession(true);
      JspWriter out = response.getWriter();
      out.println("<H1>A Random Number</H1>");
      out.println(Math.random());
      ...
}
```

```
public class ThreeParams extends HttpServlet {
  public void doGet(HttpServletRequest request,
   HttpServletResponse response)
   throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String title = "Three parameters";
    out.println(ServletUtilities.headWithTitle(title)+
          "<BODY BGCOLOR=\"#FDF5E6\">\n" +
          "<H1 ALIGN=\"CENTER\">" + title + "</H1>\n"+
          "<UL>\n" +
          "  <LI><B>param1</B>: "
          + request.getParameter("param1") + "\n" +
          "  <LI><B>param2</B>: "
          + request.getParameter("param2") + "\n" +
          "  <LI><B>param3</B>: "
          + request.getParameter("param3") + "\n" +
          "</UL>\n" +
          "</BODY></HTML>");
  }
}
```

```
<!DOCTYPE …>
<HTML>
<HEAD>
<TITLE>Three parameters</TITLE>
<LINK REL=STYLESHEET
  HREF="JSP-Styles.css"
  TYPE="text/css">
</HEAD>
<BODY>
<H1>Three parameters</H1>
<UL>
<LI><B>param1</B>:
<%= request.getParameter("param1") %>
<LI><B>param2</B>:
<%= request.getParameter("param2") %>
<LI><B>param3</B>:
<%= request.getParameter("param3") %>
</UL>
</BODY></HTML>
```

JSP Scriplets (I)

Syntax: `<% Java Code %>`

Result: Code is inserted as it is into servlet's _jspService

Example

```
<%  String queryData = request.getQueryString();
out.println("Attached GET data: " + queryData); %>
<% response.setContentType("text/plain"); %>
```

XML-compatible syntax

```
<jsp:scriptlet>Java Code</jsp:scriptlet>
```

JSP Code

```
<H2>foo</H2>
<%= bar() %>
<% baz(); %>
```

Representative resulting servlet code

```
public void _jspService(HttpServletRequest request, HttpServletResponse
  response) throws ServletException, IOException {
   response.setContentType("text/html");
   HttpSession session = request.getSession();
   JspWriter out = response.getWriter();
   out.println("<H2>foo</H2>");
   out.println(bar());
   baz();
   ...
}
```

JSP Scriplets (II)

- Scriplets are inserted into servlet exactly as written
- Need not be complete Java expressions
- Complete expressions are usually clearer and easier to maintain, however

Example

```
<% if (Math.random() < 0.5 ) { %>
Have a <B>nice</B> day!
<%  } else { %>
Have a <B>lousy</B> day!
<% } %>
```

Representative result

```
if (Math.random() < 0.5 ) {
  out.println("Have a <B>nice</B> day!");
} else {
  out.println(" Have a <B>lousy</B> day!");
}

<TABLE BORDER='1' CELLSPACING='0' CELLPADDING='5'>
<TR><TH>number</TH><TH>squared</TH></TR>
<% for ( int i=0; i<10; i++ ) { %>
<TR><TD><%= i %></TD><TD><%= (i * i) %></TD></TR>
<% } %>
</TABLE>
```

JSP Declarations (I)

Format: `<%! Java Code %>`

Result: Code is inserted verbatim into servlet's class definition, outside of any existing methods.

Examples:

```
<%! private int aField = 5; %>
<%! private void aMethod(...) {...} %>
```

For methods, it is usually better to define the method in a separate Java class.

XML-compatible syntax

```
<jsp:declaration>Java Code</jsp:declaration>
```

Codice JSP

```
<H1>Some Heading</H1>
<%! private String randomHeading() {
return("<H2>" + Math.random() + "</H2>");
} %>
<%= randomHeading() %>
```

Codice del servlet equivalente

```
public class xxxx implements HttpJspPage {
  private String randomHeading() {
    return("<H2>" + Math.random() + "</H2>");
  }
  public void _jspService(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");
    HttpSession session = request.getSession(true);
    JspWriter out = response.getWriter();
    out.println("<H1>Some Heading</H1>");
    out.println(randomHeading());
    ...
  } ...
}
```

JSP Declarations (II)

- The JSP declarations tags become declarations in the servlet class.
- Any Java declaration can be used in JSPs: instance variables, instance methods, class variables, class methods, inner classes.

```
<!DOCTYPE …>
<HTML>
<HEAD>
  <TITLE>Declarations JSP</TITLE>
  <LINK REL=STYLESHEET HREF="JSP-Styles.css‟ TYPE="text/css">
</HEAD>
<BODY>
  <H1>JSP Declarations</H1>
  <%! private int accessCount = 0; %>
  <H2>Accesses to page since server reboot:
  <%= ++accessCount %></H2>
</BODY> </HTML>
```

Warning: the environment allows concurrent calls, so variables and methods should be designed for this .

- The predefined variables (request, response, out, session, etc.) are *local to the _jspService method. Thus, they are* not available to methods defined by JSP declarations or to methods in helper classes. What can you do about this? Pass them as arguments. E.g.

```
<%! private void someMethod (HttpSession s) {
    doSomethingWith(s);
}  %>
<% someMethod(session);  %>
```

`jspInit()` e `jspDestroy()`

Le pagine JSP come i servlet, possono aver bisogno di utilizzare `init()` e `destroy()`. Per evitare che anche il servlet che risulta dalla JSP abbia bisogno di `init()` e `destroy()`, e dunque riscriva i due metodi, è meglio utilizzare `jspInit()` e `jspDestroy()`. Il servlet generato chiamerà questi metodi da `init()` e `destroy()`, ma le versioni standard di `jspInit()` e `jspDestroy()` sono vuote.

```
<%!
   public void jspInit() {
     /* initialization code here */
   }
  public void jspDestroy() {
     /* clean up code here */
   }
%>
```

Non è possibile sovrascrivere il metodo `_jspService()` che viene creato dal motore JSP durante la traduzione in servlet.

Implicit Objects

- APIs for access to important objects for JSP lifetime, *implicit objects*, can be used for objects available in JSPs.
- The JSP container creates and exposes these standard objects as variables with predefined names.
- The implicit objects have a type given by a class or interface.

Object Name	ID in PageContext	Type/Super-Type
request	REQUEST	javax.servlet.ServletRequest
response	RESPONSE	javax.servlet.ServletResponse
out	OUT	javax.servlet.jsp.JspWriter
session	SESSION	javax.servlet.http.HttpSession
pageContext	PAGECONTEXT	javax.servlet.jsp.PageContext
page	PAGE	java.lang.Object
config	CONFIG	javax.servlet.ServletConfig
application	APPLICATION	javax.servlet.ServletContext
exception	EXCEPTION	javax.lang.Throwable

Direttive JSP

- Le direttive JSP contengono istruzioni di elaborazione per il container JSP e sono eseguite al momento della compilazione. Non agiscono direttamente sull'output, dunque sono indipendenti dalle richieste provenienti dal client. La sintassi normale è:

```
<%@ NomeTipoDirettiva attributo1="valore1" attributo2="valore2" ... %>
```

La sintassi per documenti JSP (formato XML):

```
<jsp:directive.NomeTipoDirettiva attributo1="valore1"
attributo2="valore2" ... />
```

- **Direttiva *include*:** inserisce un file in una JSP prima della compilazione della JSP. I file inclusi possono essere file di testo, file HTML o file JSP. La JSP e i file inclusi formano una "unità di compilazione". Forma generale:

```
<%@ include file="relativeURL" %>
```

Esempi:

```
<%@ include file="/sales/SalesTemplate.html" %>
<%@ include file="CommonTopArea.jsp" %> <%-- Relativo al file JSP corrente --%>
```

page Directive

- The attributes of the page directive are the followings:

language: java
extends: To specify parent class of servlet that will result from JSP page! Use with extreme caution
import: Generates import statements at top of servlet definition. (java.lang.*, javax.servlet.*, javax.servlet.jsp.*, javax.servlet.http.* are automatically imported)
session: true/false. Does this JSP partecipate in a session?
buffer: none/nkb. The minimum dimension of the output buffer.
autoFlush: true/false (default: true). When the output buffer is full an exception is thrown.
isThreadSafe: true/false (default: true). If the JSP can be used or not concurrently.
info: A string that describes the JSP. It can be obtained with Servlet.getServletInfo().
errorPage: Specifies an URL of a JSP that manages Throwable objects not captured by the current JSP.
isErrorPage: true/false (default: false). This JSP is indicated in an errorPage attribute of another page. If true, the implicit variable exception is available and is set with a Throwable object that contains the error.
pageEncoding: Specifies the character coding for the JSP (default: ISO-8859-1)
contentType: The MIME type of the response and its character coding (default: text/html;charset=ISO-8859-1)
isELIgnored: true/false (default: false per Servlet v2.4). Specifies if the expressions written in Expression Language (EL) must be ignored (true) or evaluated (false) by the container.

Exemples of the `page` Directive

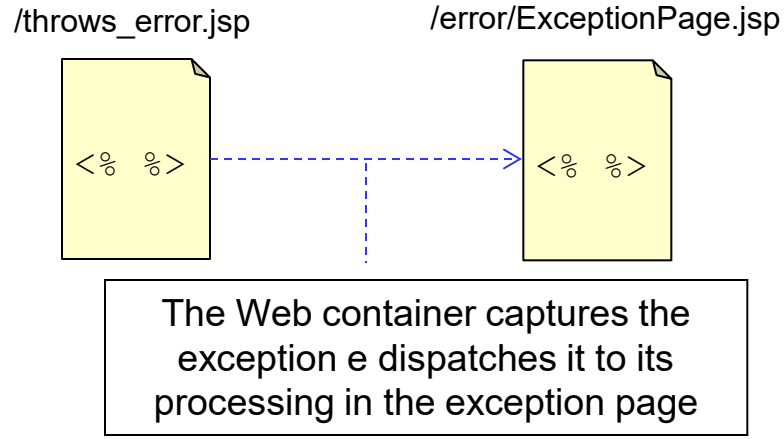
```
<%-- Classes to be imported --%>
<%@ page import="java.util.Date, java.util.Enumeration" %>

<%-- Specifies information the JSP and its error page --%>
<%@ page info="This JSP contains the site root page."
errorPage="ErrorDisplay.jsp" %>

<%-- Set the buffer minimum dimension and autoflushing --%>
<%@ page buffer="20kb" autoFlush="true" %>

<%-- Specifies the response type --%>
<%@ page contentType="text/plain;charset=UTF-8" %>
```

Exception Processing in JSPs



- An error page can be specified for each JSP.
- Declare an error page to be used each time the current JSP throws an exception:

```
<%@ page session="false" errorPage="error/ExceptionPage.jsp" %>
<%-- This page throws an exception "division by zero" --%>
<HTML><HEAD><TITLE>Error page test</TITLE></HEAD>
<BODY BGCOLOR='white'><OL>
<%
   for ( int i=10; i > -10; i-- ) {
%>
  <LI><%= 100/i %>
<%
   }
%>
</OL>
</BODY>
</HTML>
```

Develop an Error Page
Use the isErrorPage attribute in the `page` directive to indicate that this page can be used as an error page for other JSPs

Error Pages

```
…<BODY>
<%@ page errorPage="/WEB-INF/SpeedErrors.jsp" %>
<TABLE BORDER=5 ALIGN="CENTER">  <TR><TH CLASS="TITLE">Computing Speed</TABLE>
<%!
private double toDouble(String value) {
   return(Double.parseDouble(value));
}
%> ...
<%
double furlongs = toDouble(request.getParameter("furlongs"));
doutbolDeo ufbolret(nriegqhutess t=. getParameter("fortnights"));
double speed = furlongs/fortnights;
%>
<UL>
<LI>Distance: <%= furlongs %> furlongs.
<LI>Time: <%= fortnights %> fortnights.
<LI>Speed: <%= speed %> furlongs per fortnight.
</UL> </BODY></HTML>
```

Error page:

```
…<BODY>
<%@ page isErrorPage="true" %>
<TABLE BORDER=5 ALIGN="CENTER"> <TR><TH CLASS="TITLE"> Error Computing Speed</TABLE>
<P> ComputeSpeed.jsp reported the following error: <I><%= exception %></I>. This
  problem occurred in the following place:
<PRE>
<%@ page import="java.io.*" %>
<% exception.printStackTrace(new PrintWriter(out)); %>
</PRE>
</BODY></HTML>
```

isThreadSafe

Example of a Non-Threadsafe Code

```
<%! private int idNum = 0; %>
<%
String userID = "userID" + idNum;
out.println("Your ID is " + userID + ".");
idNum = idNum + 1;
%>
```

`isThreadSafe` is not needed. Synchronize normally:

```
<%! private int idNum = 0; %>
<%
synchronized(this) {
  String userID = "userID" + idNum;
  out.println("Your ID is " + userID + ".");
  idNum = idNum + 1;
}
%>
```

Using JavaBeans in JSPs

What Are Beans?

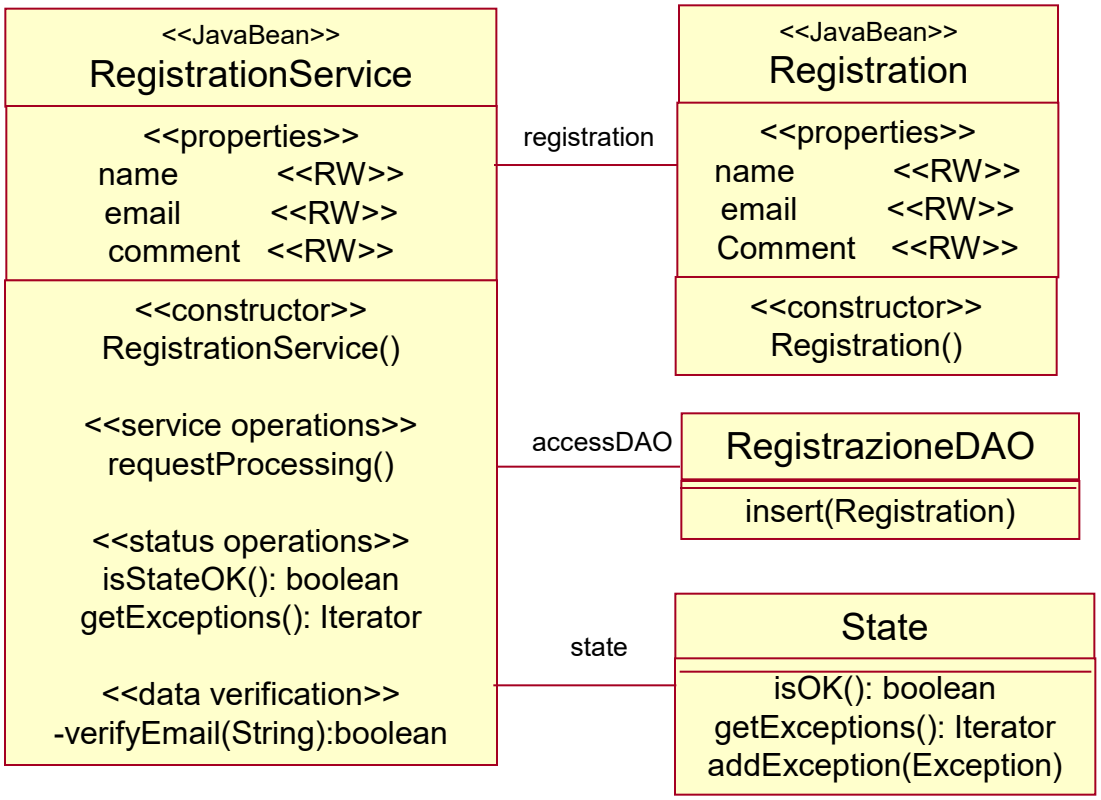
JavaBeans = Java classes that follow certain conventions:

1. Must have a zero-argument (empty) constructor. You can satisfy this requirement either by explicitly defining such a constructor or by omitting all constructors
2. Should have no public instance variables (fields).
3. Persistent values should be accessed through methods called *getXxx and setXxx.* If class has method *getTitle* that returns a String, class is said to have a String *property* named title*.* Boolean properties use *isXxx* instead of *getXxx.*

```
public double speed;

private double speed;
public double getSpeed() {
  return(speed);
}
public void setSpeed(double newSpeed) {
  speed = newSpeed;
}
public void setSpeed(double newSpeed) {
  if (newSpeed < 0) {
     sendErrorMessage(...);
     newSpeed = Math.abs(newSpeed);
  }
  speed = newSpeed;
}
```

Examples of JavaBeans



Standard Actions with JavaBeans

- JSP standard actions are tags as XML tags, that can be used in JSP pages for runtime actions.
- An empty standard standard action has the following syntax:

```
<jsp:action [attr="value"]* />
```

Exemples:

```
<jsp:useBean id="beanName" class="BeanClass" />
 <jsp:setProperty name="beanName" property="prop1" value="val" />
<jsp:getProperty name="beanName" property="prop1" />
```

- All JSP standard actions have the prefix `jsp`.

jsp:useBean

Element that builds a new bean. Exemple:

```
<jsp:useBean id="beanName" class="package.BeanClass" />
```

jsp:getProperty

Element that reads and provides the value of a property of the bean. Exemple:

```
<jsp:getProperty name="beanName" property="propertyName" />
```

jsp:setProperty

Element that modifies the value of a property of the bean. Exemple:

```
<jsp:setProperty name="beanName" property="propertyName"
value="propertyValue" />
```

jsp:useBean (I)

Exemple of bean creation with the action `<jsp:useBean ... >`. Syntax:

```
<jsp:useBean id="beanName"   class="fully_qualified_classname"
  scope="scope">
   <jsp:setProperty .../>
 </jsp:useBean>
```

Exemple:

```
<jsp:useBean id="regSvc" class="RegistrationService" scope="request">
<%
   regSvc.setName(request.getParameter("name"));
  regSvc.setEmail(request.getParameter("email"));
  regSvc.setComment(request.getParameter("comment"));
 %>
</jsp:useBean>
```

The example is equivalent to the code:

```
<%
   RegistrationService regSvc = new RegistrationService();
  regSvc.setName(request.getParameter("name"));
  regSvc.setEmail(request.getParameter("email"));
  regSvc.setComment(request.getParameter("comment"));
%>
```

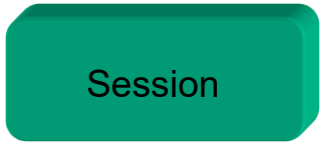
- `jsp:useBean` has two advantages with respect to scriptlets :
  - it is easier to extract values from the request parameters
  - it is easier to share objects between pages and servlets.

Access Scopes for `jsp:useBean`

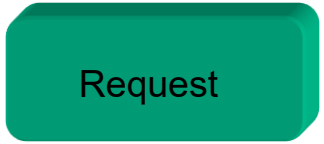
With the action jsp:useBean a JavaBean can created and stored e then found in one of four application scopes of a Web application: `page`, `request`, `session` or `application`.



The objects are accessible from pages that belong to the same application Web. They are stored in the `ServletContext` object.
(<jsp:useBean … scope="application"/>)



The objects are accessible from pages that belong to the same session where they were created. They are stored in `HTTPSession`.
(<jsp:useBean … scope="session"/>)



The objects are accessible from pages that process the same request. They are stored in `ServletRequest`.
(<jsp:useBean … scope="request"/>)



The objects are accessible from pages where they were created. They are stored in `PageContext`.
(<jsp:useBean … scope="page"/> or <jsp:useBean…>)

You can use the `scope` attribute to specify places where bean is stored.

```
<jsp:useBean id="…" class="…" scope="…" />
```

Lets multiple servlets or JSP pages share data

Also permits conditional bean creation: creates new object *only if it can't find existing one.*

jsp:useBean (II)

With `jsp:useBean`, a JSP page can create a bean and store it in the application scope. It can also obtain the reference to the bean from the same scope. `jsp:useBean` can be used without initialization code because the bean already exists.

Exemple

The bean `regSvc` was created in the `request` scope. When the page `registrazione.jsp` dispatches to an other page during the same call, the called page can access to the `regSvc` object of the same call.

```
<%-- Select the bean RegistrationService from the request domain --%>
<jsp:useBean id="regSvc" class="RegistrationService" scope="request" />
```

The action is equivalent to the following code:

```
<%
   // First, the bean is extracted from the request domain
  RegistrationService regSvc =
            (RegistrationService)request.getAttribute("regSvc");
   // If it does not exist, the bean is created and stored in the domain
   if ( regSvc == null ) {
      regSvc = new RegistrationService();
      request.setAttribute("regSvc", regSvc);
      // the useBean body, if any, is executed
      . . .
   }
%>
```

jsp:setProperty

- Exemple of initialization of the same instance with <jsp:setProperty …/>:

```
<jsp:useBean id="regSvc" class="RegistrationService" scope="request">
   <jsp:setProperty name="regSvc" property="name"/>
   <jsp:setProperty name="regSvc" property="email"/>
   <jsp:setProperty name="regSvc" property=comment"/>
</jsp:useBean>
```

The code is equivalent with:

```
<%
   regSvc.setNome(request.getParameter("name"));
  regSvc.setEmail(request.getParameter("email"));
  regSvc.setCommento(request.getParameter("comment"));
 %>
```

- The initialization with <jsp:setProperty . . ./> can be shortened as in the followings:

```
<jsp:useBean id="regSvc" class="RegistrationService" scope="request">
   <jsp:setProperty name="regSvc" property="*"/>
</jsp:useBean>
```

Assignments with `setProperty()`

Value Source	Element Syntax
String Constant	`<jsp:setProperty name="beanName" property="propName" value="string constant"/>`
Request parameter	`<jsp:setProperty name="beanName" property="propName" param="paramName"/>`
Request parameter that has the same name with the property	`<jsp:setProperty name="beanName" property="propName"/>` `<jsp:setProperty name="beanName" property="*"/>`
Expression	`<jsp:setProperty name="beanName" property="propName" value="expression"/>` `<jsp:setProperty name="beanName" property="propName" ><` `  <jsp:attribute name="value">` `  expression` `  </jsp:attribute>` `</jsp:setProperty>`

1. *beanName* has to be the same with the attribute name in the useBean element
2. A method *setPropName* should be in a JavaBeans component.
3. *paramName* has to be the request parameter name.

jsp:forward

```
<%
if ( request.getMethod().equals("POST") ) {
  regSvc.reuqestProcess();
  if ( regSvc.isStateOK() ) {
%>
   <%-- It follows the welcome page. --%>
   <jsp:forward page="welcome.jsp" />
<%
  } else {
%>
   <%-- An error occured and a message should be output. --%>
   <jsp:forward page="registrationError.jsp" />
<%  } // fine if (regSvc.isStateOK())
%>
<%} // fine if (request.getMethod().equals("POST") )
%>
```

The use of `<jsp:forward . . ./>` is equivalent with the call of the `forward()` method for a RequestDispatcher object :

```
RequestDispatcher view = request.getRequestDispatcher(page);
view.forward(request, response);
return;
```

jsp:getProperty

- It is possible to access the bean properties with `jsp:getProperty`.

Exemple

```
<%-- Select the RegistrationService bean from the request domain --%>
<jsp:useBean id="regSvc" class="RegistrationService" scope="request" />
<BR>Thanks, <jsp:getProperty name="regSvc" property="name"/>, for your
   registration.
```

For this action the equivalent case is:

```
<%= regSvc.getNome() %>
```

Using Beans

```
package core;
public class StringBean {
  private String message="No message specified";
  public String getMessage(){
   return message;
  }
  public void setMessage(String message) {
    this.message = message;
  }
}
```

```
<jsp:useBean id="stringBean" class="core.StringBean" />
<OL>
  <LI>Initial value (from jsp:getProperty): <I><jsp:getProperty
  name="stringBean" property="message" /></I>
  <LI>Initial value (from JSP expression):<I><%= stringBean.getMessage()
  %></I>
  <LI><jsp:setProperty name="stringBean" property="message" value="Best
  string bean: Fortex"/> Value after setting property with
  jsp:setProperty: <I><jsp:getProperty name="stringBean"
  property="message" /></I>
  <LI><% stringBean.setMessage ("My favorite: Kentucky Wonder"); %>Value
  after setting property with scriptlet:<I><%= stringBean.getMessage()
  %></I>
</OL>
```

Setting Bean Properties

```
<!DOCTYPE ...>
. . .
<jsp:useBean id="entry" class="core.SaleEntry" />
<%-- setItemID expects a String --%>
<jsp:setProperty name="entry" property="itemID"
value='<%= request.getParameter("itemID") %>' />
<%
int numItemsOrdered = 1;
try {
  numItemsOrdered = Integer.parseInt(request.getParameter("numItems"));
} catch(NumberFormatException nfe) {}
%>
<%-- setNumItems expects an int --%>
<jsp:setProperty name="entry" property="numItems" value="<%=
  numItemsOrdered %>" />
<%
double discountCode = 1.0;
try {
  String discountString = request.getParameter("discountCode");
  discountCode = Double.parseDouble(discountString);
} catch(NumberFormatException nfe) {}
%>
<%-- setDiscountCode expects a double --%>
<jsp:setProperty name="entry" property="discountCode" value="<%=
  discountCode %>" />
```

Conditional Bean Creation (I)

Bean conditionally created

– jsp:useBean results in new bean being instantiated only if no bean with same id and scope can be found.

– If a bean with same id and scope is found, the preexisting bean is simply bound to variable referenced by id.

• **Bean properties conditionally set**

– <jsp:useBean ... />

replaced by

<jsp:useBean ...>statements</jsp:useBean>

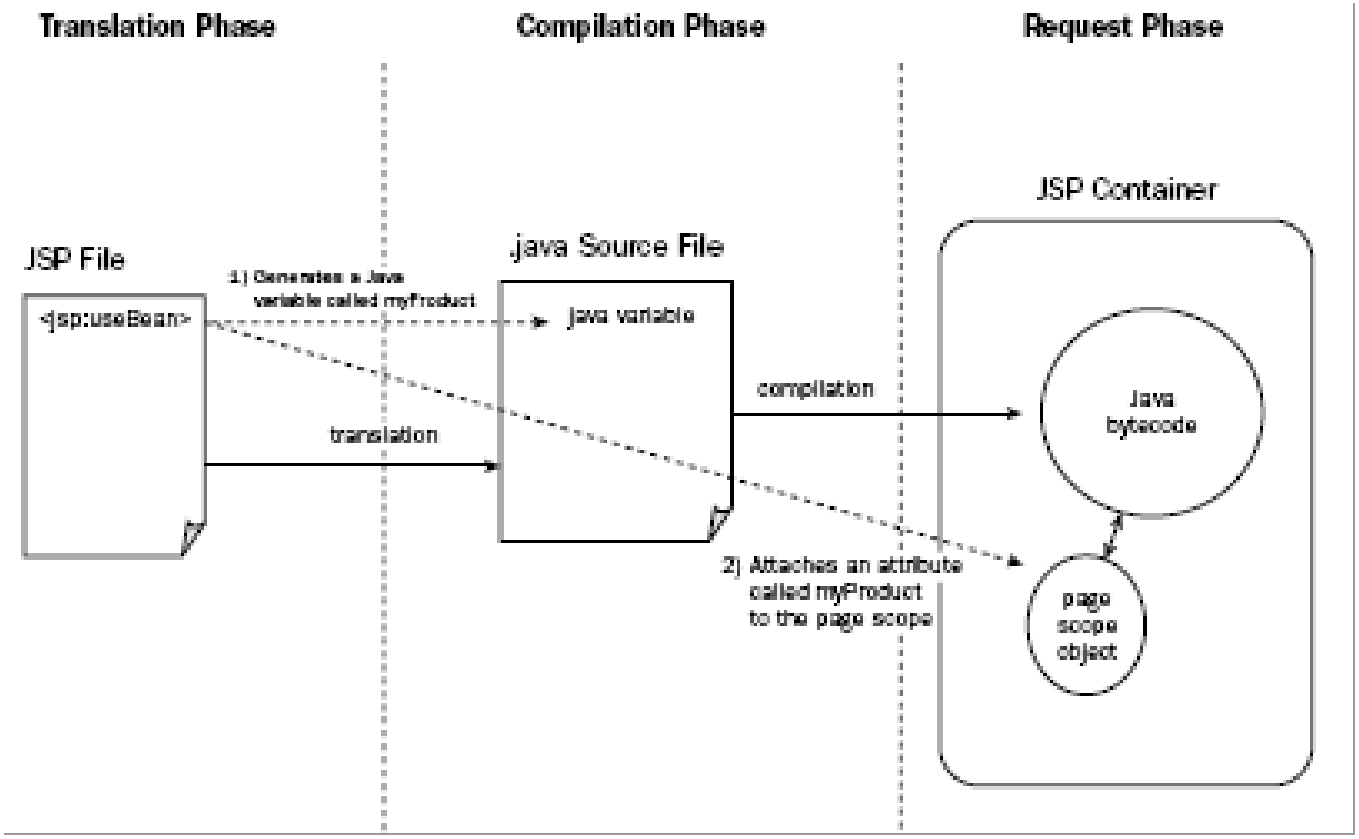
– The statements (jsp:setProperty elements) are executed *only if a new bean is created, not if an existing bean is* found.

Conditional Bean Creation (II)

```
public class AccessCountBean {
  private String firstPage;
  private int accessCount = 1;
  public String getFirstPage() {
    return(firstPage);
  }
  public void setFirstPage(String firstPage) {
    this.firstPage = firstPage;
  }
  public int getAccessCount() {
    return(accessCount);
  }
  public void setAccessCountIncrement(int increment) {
    accessCount = accessCount + increment;
  }
}
<jsp:useBean id="counter" class="AccessCountBean" scope="application">
  <jsp:setProperty name="counter" property="firstPage" value="SharedCounts1.jsp" />
</jsp:useBean>
Of SharedCounts1.jsp (this page),
<A HREF="SharedCounts2.jsp">SharedCounts2.jsp</A>, and
<A HREF="SharedCounts3.jsp">SharedCounts3.jsp</A>,
<jsp:getProperty name="counter" property="firstPage" />
 was the first page accessed.
<P> Collectively, the three pages have been accessed
<jsp:getProperty name="counter" property="accessCount" />
 times.
<jsp:setProperty name="counter" property="accessCountIncrement" value="1" />
```

Operation of the <java:usebean> standard action

- To create the Java language–accessible JavaBean reference called myProduct, it is necessary to generate custom code during the translation phase of JSP processing.



Organizing presentation fragments

- Presentation fragments = HTML or JSP files used in a web application. Usually they are gathered in a unique folder `/incl` on disk and extracted with:
  - `include` directive
  - `jsp:include` standard action

The `include` Directive

- Includes a fragment in the main JSP page during a servlet compilation. The syntax is:

```
<%@ include file="URLfragment" %>
```

- Exemple:

```
<!-- START of the side-bar -->
   <TD BGCOLOR='#CCCCFF' WIDTH='160' ALIGN='left'>
     <%@ include file="/incl/side-bar.jsp" %>
   </TD>
 <!-- END of the side-bar -->
```

- If the fragment contains other fragments, all these fragments are recursively inserted until all directives are eliminated.
- The path used in the directive can be relative to the folder containing the original page or absolute (when initiates with '/') that is relative to the root of the web application context.

jsp:include

- `jsp:include` allow us to include at *runtime* a text fragment in the HTTP response of the main JSP page. The syntax :

```
<jsp:include page="URLfragment" />
```

Exemple:

```
<!-- START of the banner -->
        <jsp:include page="/incl/banner.jsp" />
<!-- END of the banner -->
```

- The action `jsp:include` is equivalent to the include method called on the RequestDispatcher object:

```
RequestDispatcher fragment = request.getRequestDispatcher(page);
fragment.include(request, response);
```

Including Files at Request Time: jsp:include

Format

```
<jsp:include page="Relative URL" />
```

Purpose: To reuse JSP, HTML, or plain text content and to permit updates to the included content without changing the main JSP page(s)

Notes Relative URLs that starts with slashes are interpreted relative to the Web app, not relative to the server root.

```
<BODY>
<TABLE BORDER=5 ALIGN="CENTER"> <TR><TH CLASS="TITLE">
What's New at JspNews.com</TABLE>
<P> Here is a summary of our three most recent news stories:
<OL>
<LI><jsp:include page="/WEB-INF/Item1.html" />
<LI><jsp:include page="/WEB-INF/Item2.html" />
<LI><jsp:include page="/WEB-INF/Item3.html" />
</OL>
</BODY></HTML>
```

The page **Item1.html**

```
...
<B>Bill Gates acts humble.</B> In a startling and unexpected development,
  Microsoft big wig Bill Gates put on an open act of humility yesterday.
<A HREF="http://www.microsoft.com/Never.html">More details...</A>
```

Including Files at Page Translation Time: <%@ include … %>

Format: `<%@ include file="Relative URL" %>`

Purpose: To reuse JSP content in multiple pages, where JSP content affects main page.

Notes: Servers are not required to detect changes to the included file, and in practice they don't. Thus, you need to change the JSP files whenever the included file changes.

	jsp:include	<%@ include …%>
Basic syntax	`<jsp:include page="..."/>`	`<%@ include file="..." %>`
When inclusion occurs	Request time	Page translation time
What is included	Output of page	Contents of file
Number of resulting servlets	Two	One
Can included page set response headers that affect the main page?	No	Yes
Can included page define fields / methods that main page uses?	No	Yes
Does main page need to be updated when included page changes?	No	Yes

Including pages instead of forwarding to them

With the forward method of RequestDispatcher:
– Control is *permanently* transferred to new page
– Original page *cannot* generate any output
With the include method of RequestDispatcher:
– Control is *temporarily* transferred to new page
– Original page *can* generate output before and after the included page
– Original servlet does not see the output of the included page
– Useful for portals: JSP presents pieces, but pieces arranged in different orders for different users

Example

```
response.setContentType("text/html");
String firstTable, secondTable, thirdTable;
if (someCondition) {
  firstTable = "/WEB-INF/Sports-Scores.jsp";
  secondTable = "/WEB-INF/Stock-Prices.jsp";
  thirdTable = "/WEB-INF/Weather.jsp";
} else if (...) { ... }
RequestDispatcher dispatcher =
    request.getRequestDispatcher("/WEB-INF/Header.jsp");
dispatcher.include(request, response);
dispatcher = request.getRequestDispatcher(firstTable);
dispatcher.include(request, response);
dispatcher = request.getRequestDispatcher(secondTable);
dispatcher.include(request, response);
dispatcher = request.getRequestDispatcher(thirdTable);
dispatcher.include(request, response);
dispatcher = request.getRequestDispatcher("/WEB-INF/Footer.jsp");
dispatcher.include(request, response);
```

jsp:param (I)

- `jsp:include` allows us to parameterize inclusion in order to modify it with information at runtime.
- With `jsp:param` the action `jsp:include` allows to pass adding parameters to the presentation fragment.
- Syntax:

```
<jsp:include page="fragmentURL">
  <jsp:param name="paramName" value="paramValue"/>
</jsp:include>
```

- The action `jsp:param` is contained in the action `jsp:include`. A `jsp:include` tag can include any number of `jsp:param` actions.

```
<!-- START of the banner -->
  <jsp:include page="/incl/banner.jsp">
    <jsp:param name="subTitle" value="Thanks for subscription!"/>
  </jsp:include>
<!-- END of the banner -->
```

- The pair name-value specified in the action `jsp:param` is stored in the object `request` transmitted to the JSP page of the fragment as any form parameter. It can be accessed with `getParameter()`.

jsp:param (II)

```
<%@ page import="Championship" %>
<%-- Determines the title and subtitle of the page. --%>
<%   String titleBanner = "Football Championship";
   String subTitle = request.getParameter("subTitle");
   // Uses the selected championship title (if known)
   Championship championship =
  (Championship)request.getAttribute("championship");
   if (championship != null ) {
      bannerTitle = championship.getTitle();
   }
%>
    <FONT SIZE='5' FACE='Helvetica, san-serif'>
    <%= bannerTitle %>
    </FONT>
    <% if (bannerTitle != null ) { %>
    <BR><BR>
    <FONT SIZE='4' FACE='Helvetica, sans-serif'>
    <%= bannerTitle %>
    </FONT>
    <% } %>
```

jsp:plugin (I)

APPLET Tag

```
<APPLET CODE="MyApplet.class" WIDTH=475 HEIGHT=350>
</APPLET>
```

Equivalent jsp:plugin

```
<jsp:plugin type="applet" code="MyApplet.class" width="475" height="350">
</jsp:plugin>
```

Resultant HTML

```
<object classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
width="300" height="200"
codebase="http://java.sun.com/products/plugin/1.2.2/jinst all-1_2_2-
  win.cab#Version=1,2,2,0">
<param name="java_code" value="SomeApplet.class">
<param name="type" value="application/x-java-applet;">
<COMMENT>
<embed type="application/x-java-applet;" width="300" height="200"
  pluginspage="http://java.sun.com/products/plugin/"
  java_code="SomeApplet.class">
<noembed>
</COMMENT>
</noembed></embed>
</object>
```

jsp:plugin (II)

```
<jsp:plugin
   type="bean|applet"
   code="objectCode"
   codebase="objectCodebase"
   { align="alignment" }
   { archive="archiveList" }
   { height="height" }
   { hspace="hspace" }
   { jreversion="jreversion" }
   { name="componentName" }
   { vspace="vspace" }
   { width="width" }
   { nspluginurl="url" }
   { iepluginurl="url" } >
   { <jsp:params>
     { <jsp:param name="paramName" value= paramValue" /> }+
   </jsp:params> }
   { <jsp:fallback> arbitrary_text </jsp:fallback> }
 </jsp:plugin>
```

JSP 2.0. Expression Language

The JSP 2.0 EL provides concise, easy-to-read access to:

- Bean properties
- Collection elements
- Standard HTTP elements such as request parameters, request headers, and cookies

The JSP 2.0 EL works best with MVC

- Use only to output values created by separate Java code

Resist use of EL for business logic

Expression Language

- An expression language (EL) promotes JSP scoped attributes as the standard way to communicate information from business logic to JSP pages.
- It is not a general purpose programming language but a data access language, which makes it possible to access (and manipulate) application data without having to use scriptlets or request-time expression values.

`${name}` and `${name.address.zip}` are two possibilities to access variables or bean properties.

- The EL supports arithmetic, relational, and logical operators to handle the most common data manipulations.

```
<c:if test="${sessionScope.cart.numberOfItems > 0}">
   ...
</c:if>
```

- EL expressions are included between ${ e } .
- EL expressions are used in two situations:
  - In the template text (the expression value is evaluated and included in the current output);
  - in any standard or customized tag attribute that accepts an expression.

Setting values of attributes

1. Value of a unique expression:

   ```
   <xxx:tag value="${expr}"/>
   ```

   The expression is evaluated and the result is converted to the attribute type.

2. Value of one or more expressions separated of text:

   ```
   <some:tag value="some${expr}${expr}text${expr}"/>
   ```

   The expression is evaluated from left to right. Every expression is converted to a string and then all strings are concatenated with the alternate text. The resulting string is converted to the attribute type.

3. Value as a text:

   ```
   <some:tag value="sometext"/>
   ```

   The String string of the attribute is converted to the expected type of the attribute.

Exemple:

The expression `${1.2E4 + 1.4}` for a float attribute, undergoes the following conversions:

```
Float.valueOf("1.2E4 + 1.4").floatValue()
```

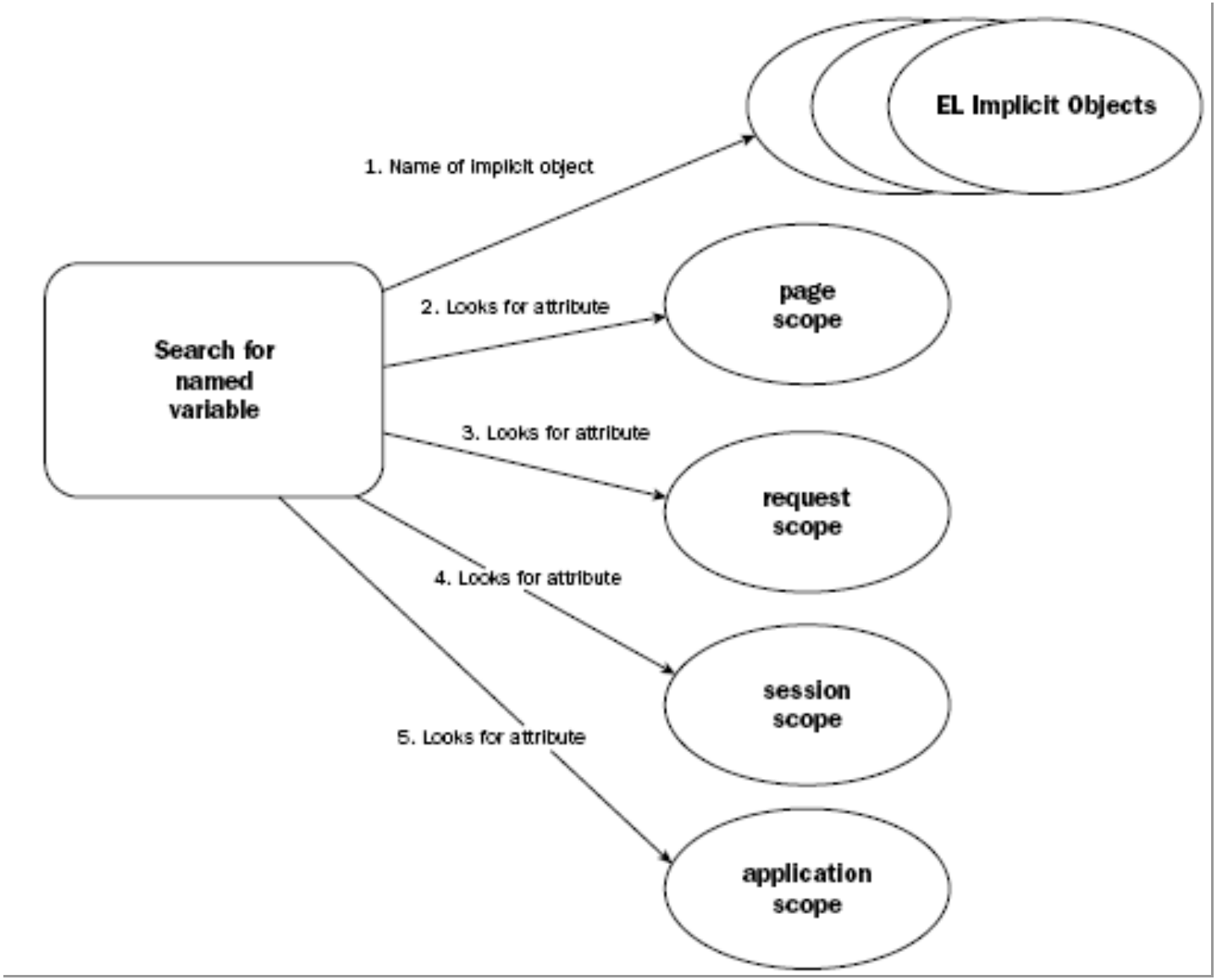
Variables

- You can access application data either as a property of an object, using the dot (.) operator, or an associative array element using the bracket `['name']` operator.
- The expression `${data}` represents the scoped variable named data.
- You can retrieve properties from collections using either the dot (.) or bracket ([]) operator. `${name.property}` and `${name["property"]}` are equivalent:
  - The dot (.) operator is used to retrieve a named property. For example, `${customer.name}`.
  - The bracket operator ([]) can be used to retrieve a named property, as in `${customer["name"]}` or `${customers[0]}` .

Evaluation of an expression `expA.expB`

- To evaluate `expA.expB` first are evaluated `expA` to `valA` and `expB` to `valB`.
  - If one of`valA` or `valB` is `null`, the whole expression is `null`, otherwise :
  - If `valA` is a `Map`, `valA.get(valB)` is returned (if `!valA.containsKey(valB)` is returned `null`);
  - If `valA` is a `List` or an array, `valB` is converted to an `int` and `valA.get(valB)` or `Array.get(valA, valB)` is returned
  - If `valA` is a javaBean object, the value `valB` is converted to a string. If `valB` is a readable property of `valA`, the result is that of `getXxx()`. Otherwise an exception is thrown in `getXxx()` and an error is reported.

Order of Searching for an EL named variable



Accessing Collections (I)

The expression `${attributeName[entryName]}` works for:

- Array. Equivalent to `theArray[index]`
- List. Equivalent to the `List.get(index)`
- Map. Equivalent to `theMap.get(keyName)`
- Equivalent forms (for HashMap): `${stateCapitals["maryland"]}` and `${stateCapitals.maryland}`, but `${listVar.2}` is illegal since 2 is not a legal var name.

Accessing Collections (II)

```
public class Collections extends HttpServlet {
 public void doGet(HttpServletRequest request, HttpServletResponse response)
  throws ServletException, IOException {
  String[] firstNames = { "Bill", "Scott", "Larry" };
  ArrayList lastNames = new ArrayList();
  lastNames.add("Ellison");
  lastNames.add("Gates");
  lastNames.add("McNealy");
  HashMap companyNames = new HashMap();
  companyNames.put("Ellison", "Sun");
  companyNames.put("Gates", "Oracle");
  companyNames.put("McNealy", "Microsoft");
  request.setAttribute("first", firstNames);
  request.setAttribute("last", lastNames);
  request.setAttribute("company", companyNames);
  RequestDispatcher dispatcher =
   request.getRequestDispatcher("/el/collections.jsp");
  dispatcher.forward(request, response);
 }
}
```

```
<!DOCTYPE …>
…
<BODY>
<TABLE BORDER=5 ALIGN="CENTER">
<TR><TH CLASS="TITLE"> Accessing Collections </TR></TH>
</TABLE>
<P>
<UL>
  <LI>${first[0]} ${last[0]} (${company["Ellison"]})
  <LI>${first[1]} ${last[1]} (${company["Gates"]})
  <LI>${first[2]} ${last[2]} (${company["McNealy"]})
</UL>
</BODY></HTML>
```

Implicit objects provided by EL

Implicit Object	Content
applicationScope	A collection of scoped variables from applications scope
cookie	A collection of all cookies
header	HTTP request headers as strings
headerValues	HTTP request headers as collections of strings
initParam	A collection of all application parameter names
pageContext	The javax.servlet.jsp.PageContext object for the current page
pageScope	A collection of all page scope objects
param	A collection of all request parameters as strings
paramValues	All request parameters as collections of strings
requestScope	A collection of all request scope objects
sessionScope	A collection of all session scope objects

Operators

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/ or div	Division
% or mod	Modulus (Remainder)
== or =	Equality
!= or !=	Inequality
< or lt	Less than
> or gt	Greater than
<= or le	Less than or equal to
>= or ge	Greater than or equal to
&& or and	Logical AND
|| or or	Logical OR
! or not	Boolean complement
empty	Check for empty value
a ? b : c	Conditional operator

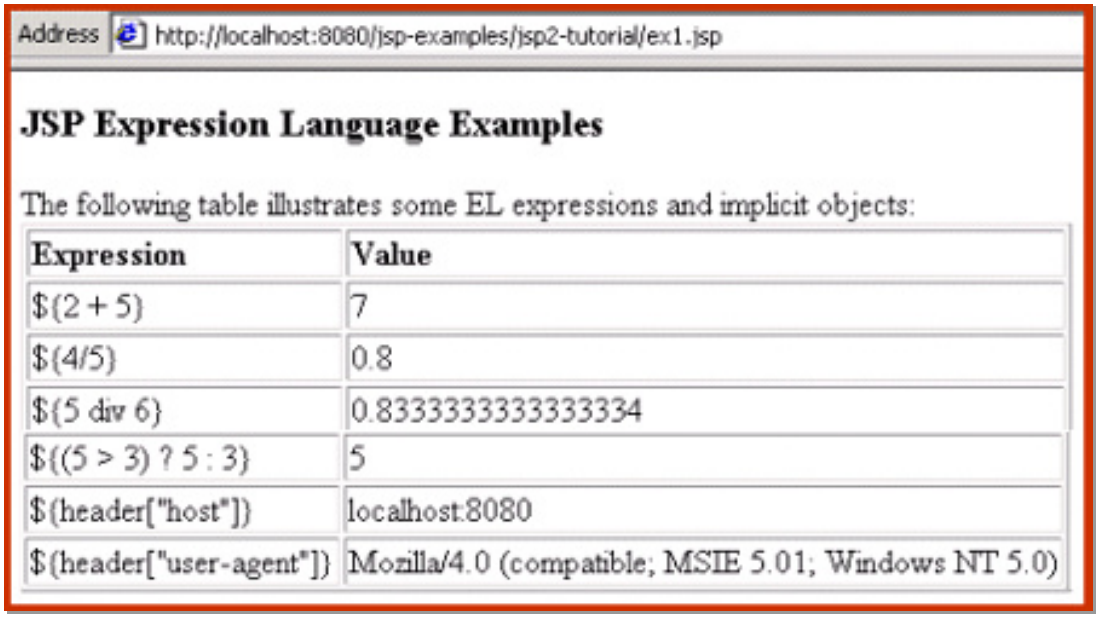
- The EL supports arithmetic, relational, and logical operators to handle the most common data manipulations.
- In addition, a special operator `empty` for testing if an object is empty is provided.
- You can use the empty operator to determine whether a collection or a string is empty or null.
- For example, `${empty param.name}` will be true only if the request parameter named `param` is not present

Examples of EL Expressions

EL Expression	Result
`${1 > (4/2)}`	false
`${4.0 >= 3}`	true
`${100.0 == 100}`	true
`${(10*10) ne 100}`	false
`${'a' < 'b'}`	true
`${'hip' gt 'hit'}`	false
`${4 > 3}`	true
`${1.2E4 + 1.4}`	12001.4
`${3 div 4}`	0.75
`${10 mod 4}`	2
`${!empty param.Add}`	True if the request parameter Add is not null or empty.
`${pageContext.request.contextPath}`	Context path
`${sessionScope.cart.numberOfItems}`	The value of the property numberOfItems of the artribute cart in the session scope.
`${param['mycom.productId']}`	The value of the request parameter mycom.productId
`${header["host"]}`	The host
`${departments[deptName]}`	Value of the entry deptName in the departments map.
`${requestScope['javax.servlet.forward.servlet_path']}`	The value of the attribute named javax.servlet.forward.servlet_path in the request scope

EL Expressions and Implicit Objects

```
<HTML>
<HEAD>
<TITLE>Expression Language Examples</TITLE>
</HEAD>
<BODY>
 <H3>JSP Expression Language Examples</H3>
 <P>mThe following table illustrates some EL expressions and implicit objects:
 <TABLE BORDER="1">
  <THEAD>
    <TD><B>Expression</B></TD>
    <TD><B>Value</B></TD>
  </THEAD>
  <TR>
    <TD>\${2 + 5}</TD>
    <TD>${2 + 5}</TD>
  </TD>
  <TR>
    <TD>\${4/5}</TD>
    <TD>${4/5}</TD>
  </TR>
  <TR>
    <TD>\${5 div 6}</TD>
    <TD>${5 div 6}</TD>
  </TR>
 <TR>
    <TD>\${(5 > 3) ? 5 : 3}</TD>
    <TD>${(5 > 3) ? 5 : 3}</TD>
  </TR>  <TR>
    <TD>\${header["host"]}</TD>
    <TD>${header["host"]}</TD>
  </TR>
  <TR>
    <TD>\${header["user-agent"]}</TD>
    <TD>${header["user-agent"]}</TD>
  </TR>
 </TABLE>
 </BODY>
</HTML>
```



$param[var]

```
<HTML>
<HEAD>
<TITLE>Form content</TITLE>
</HEAD>
 <BODY>
 <H3>Fill-out-form</H3>
<P>
<FORM action="/jsp-examples/jsp2/form.jsp" method="GET">
   Name = <input type="text" name="name" value="${param['name']}">
          <input type="submit" value="Submit name">
</FORM>
<P>
The name is: ${param.name}
</BODY>
</HTML>
```

JSP 1.2 ==> JSP 2.0

```
…
<BODY>
<jsp:useBean id="randomNum"
type="core.NumberBean" scope="request" />
<H2>Random Number:
<jsp:getProperty name="randomNum"
property="number" />
</H2>
</BODY></HTML>
```



```
…
<BODY>
<H2>Random Number:
${randomNum.number}
</H2>
</BODY></HTML>
```

```
…
<BODY>
<H1>Thanks for Registering</H1>
<jsp:useBean id="nameBean"
   type="core.NameBean" scope="session" />
<H2>First Name:
<jsp:getProperty name="nameBean"
   property="firstName" /></H2>
<H2>Last Name:
<jsp:getProperty name="nameBean"
   property="lastName" /></H2>
</BODY></HTML>
```



```
…
<BODY>
<H1>Thanks for Registering</H1>
<H2>First Name:
   ${nameBean.firstName}</H2>
<H2>Last Name:
   ${nameBean.lastName}</H2>
</BODY></HTML>
```

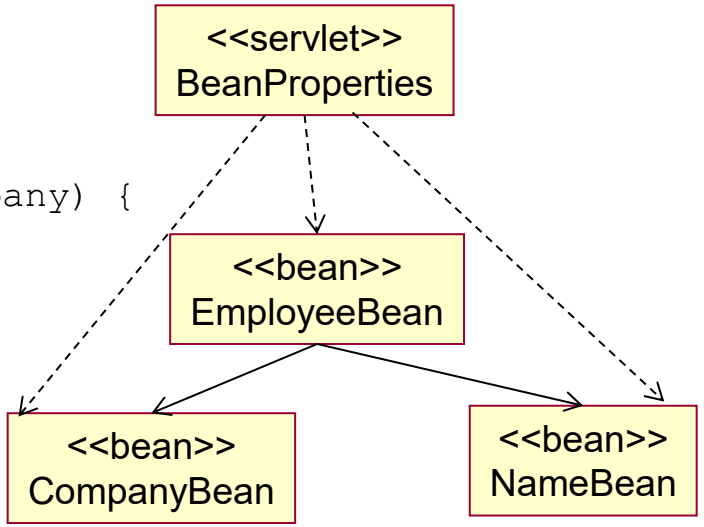
```
…
<BODY>
<H1>A Prime Number</H1>
<jsp:useBean id="primeBean"
   type="core.PrimeBean" scope="application"
   />
<jsp:getProperty name="primeBean"
   property="prime" />
</BODY></HTML>
```



```
…
<BODY>
<H1>A Prime Number</H1>
${primeBean.prime}
</BODY></HTML>
```

Accessing Bean Properties (I)

```
public class BeanProperties extends HttpServlet {
  public void doGet(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
    NameBean name = new NameBean("John", "Smith");
    CompanyBean company =
        new CompanyBean("core.com", "J2EE Training and Consulting");
    EmployeeBean employee = new EmployeeBean(name, company);
    request.setAttribute("employee", employee);
    RequestDispatcher dispatcher =
        request.getRequestDispatcher ("/el/bean-properties.jsp");
    dispatcher.forward(request, response);
  }
}
public class EmployeeBean {
  private NameBean name;
  private CompanyBean company;
  public EmployeeBean(NameBean name, CompanyBean company) {
    setName(name);
    setCompany(company);
  }
  public NameBean getName() { return(name); }
  public void setName(NameBean newName) {
    name = newName;
  }
  public CompanyBean getCompany() {return(company);}
  public void setCompany(CompanyBean newCompany) {
    company = newCompany;
  }
}
```



Accessing Bean Properties (II)

```
public class NameBean {
  private String firstName = "Missing first name";
  private String lastName = "Missing last name";
  public NameBean() {}
  public NameBean(String firstName, String lastName) {
    setFirstName(firstName);
    setLastName(lastName);
  }
  public String getFirstName() {
    return(firstName);
  }
  public void setFirstName(String newFirstName) {
    firstName = newFirstName;
  }…
}
public class CompanyBean {
  private String companyName;
  private String business;
  public CompanyBean(String companyName, String business) {
    setCompanyName(companyName);
    setBusiness(business);
  }
  public String getCompanyName() { return(companyName); }
  public void setCompanyName(String newCompanyName) {
    companyName = newCompanyName;
  }
  public String getBusiness() { return(business); }
  public void setBusiness(String newBusiness) {
    business = newBusiness;
  }
}
```

Accessing Bean Properties (III)

```
<!DOCTYPE …>
…
<UL>
  <LI><B>First Name:</B>
     ${employee.name.firstName}
  <LI><B>Last Name:</B>
     ${employee.name.lastName}
  <LI><B>Company Name:</B>
     ${employee.company.companyName}
  <LI><B>Company Business:</B>
     ${employee.company.business}
</UL>
</BODY>
</HTML>
```

JSTL

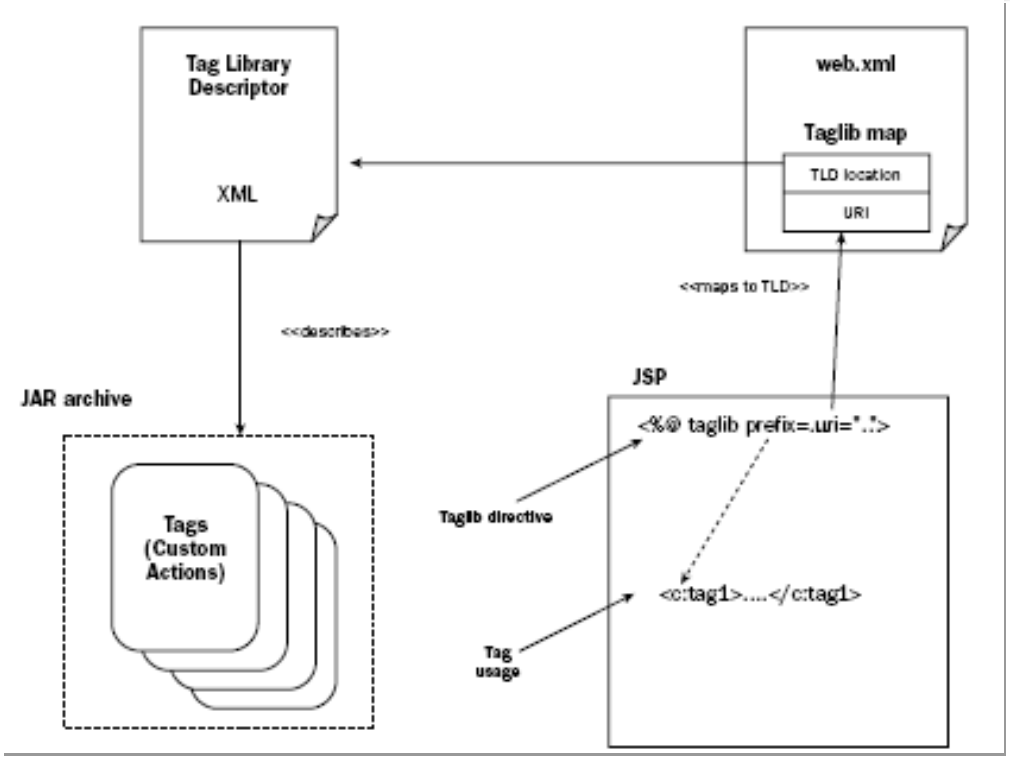
JSTL

- JavaServer Pages Standard Tag Library (JSTL) encapsulates common functionality, essential for many applications.
- JSTL offers standard tags that can be used in any container in optimized versions.
- JSTL offers tags that support common tasks as iteration, conditionals, for XML document processing, or relational database access.
- Five standard tag libraries . Their URLs are:

Area	Subfunction	Prefisso
Core	Variable Support	c
	Flow Control	
	URL Management	
	Miscellaneous	
XML	Core	x
	Flow Control	
	Transformation	
I18n	Locale	fmt
	Message formatting	
	Number and date formatting	
Database	SQL	sql
Functions	Collection length	fn
	String manipulation	

```
Core: http://java.sun.com/jsp/jstl/core
XML: http://java.sun.com/jsp/jstl/xml
Internationalization: http://java.sun.com/jsp/jstl/fmt
SQL: http://java.sun.com/jsp/jstl/sql
Functions: http://java.sun.com/jsp/jstl/functions
```

Anatomy of a Tag Library



- A Tag Library Descriptor file (TLD) is included with most tag libraries. The content of this file describes the tags and their parameterization to the container (or a development tool). TLD files are text files created in the XML format.
- The actual implementation of the tags can reside in:

1. the `<application directory>/WEB-INF/lib` directory,
2. the `<application directory>/WEB-INF/classes` directory, or
3. JSP-based tag files in a directory below `<application directory>/WEB-INF`.

Tags in the `core` library

Area	Function	Tag	Prefix
Core	**Variable support**	remove set	c
	Flow control	choose when otherwise forEach forTokens if	
	URL management	import param redirect param url param	
	Various	catch out	

- Tags for variable support:

```
<c:set var="foo" scope="session" value=".."/>
<c:set var="foo">   ... </c:set>
<c:set var="bookId" value="${param.Remove}"/>
```

- Tags for execution flow control:

Iteration tags:

```
<c:forEach var="item"
   items="${sessionScope.cart.items}">
    ...
    <tr>
      <td align="right" bgcolor="#ffffff">
      ${item.quantity}
    </td>
    ...
</c:forEach>
```

Looping Tags

Looping with explicit numeric values

```
<c:forEach var="name" begin="x" end="y" step="z">
  words, words <c:out value="${name}"/>
</c:forEach>
```

Looping over data structures. Can loop down arrays, strings, collections, maps

```
<c:forEach var="name" items="array-or-collection">
  words, words <c:out value="${name}"/>
</c:forEach>
```

Looping down delimited strings

```
forTokens
```

JSP without JSTL

```
<UL>
<%
for(int i=0;i<messages.length;i++){
  String message = messages[i];
%>
<LI><%= message %>
<% } %>
</UL>
```

JSP with JSTL

```
<UL>
<c:forEach var="message" items="${messages}">
<LI><c:out value="${message}"/>
</c:forEach>
</UL
```

Looping (I)

Looping with Simple Numeric Values

```
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>
<UL>
  <c:forEach var="i" begin="1" end="10">
    <LI><c:out value="${i}"/>
  </c:forEach>
</UL>
```

Looping with a Designated Step Size

```
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>
<UL>
<c:forEach
   var="seconds"
   begin="0"
   end="${pageContext.session.maxInactiveInterval}"
   step="100">
  <LI><c:out value="${seconds}"/> seconds.
</c:forEach>
  <LI>Timeout exceeded.
</UL>
```

Looping with a Designated Arbitrarily Delimited Strings

```
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>
<UL>
<c:forTokens var="color"
   items="(red (orange) yellow)(green)((blue) violet)"
   delims="()">
  <LI><c:out value="${color}"/>
</c:forTokens>
</UL>
```

Looping (II)

Looping Down Arrays

```
<% String[] words = { "foo", "bar", "baz"};
    pageContext.setAttribute("words", words); %>
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>
<H2>Key Words:</H2>
<UL>
<c:forEach var="word" items="${words}">
  <LI><c:out value="${word}"/>
</c:forEach>
</UL>
<H2>Values of the test Parameter:</H2>
<UL>
<c:forEach var="val" items="${paramValues.test}">
  <LI><c:out value="${val}"/>
</c:forEach>
</UL>
```

Looping Down Comma-Delimited Strings

```
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>
<UL>
<c:forEach
   var="country"
   items="Australia,Canada,Japan,Philippines,USA">
  <LI><c:out value="${country}"/>
</c:forEach>
</UL>
```

Conditional Evaluation Tags

One choice: if

```
<c:if test="${someTest}">
  Content
</c:if>
```

Lots of choices: choose

```
<c:choose>
<c:when test="test1">Content1</c:when>
<c:when test="test2">Content2</c:when>
...
<c:when test="testN">ContentN</c:when>
<c:otherwise>Default Content</c:otherwise>
</c:choose>
```

Example

```
<%@ taglib prefix="c"
   uri="http://java.sun.com/jstl/core"%>
<UL>
<c:forEach var="i" begin="1" end="10">
  <LI><c:out value="${i}"/>
     <c:if test="${i > 7}">
     (greater than 7)
     </c:if>
</c:forEach>
</UL>
```

Example

```
<%@ taglib prefix="c"
  uri="http://java.sun.com/jstl/core"%>
<UL>
<c:forEach var="i" begin="1" end="10">
  <LI><c:out value="${i}"/>
    <c:choose>
      <c:when test="${i < 4}">
         (small)
      </c:when>
      <c:when test="${i < 8}">
         (medium)
      </c:when>
      <c:otherwise>
         (large)
      </c:otherwise>
    </c:choose>
</c:forEach>
</UL>
```

Examples

1.

```
<c:if test="${!empty param.Add}">
   <c:set var="bid" value="${param.Add}"/>
   <jsp:useBean id="bid" type="java.lang.String" />
    <sql:query var="books"
     dataSource="${applicationScope.bookDS}">
     select * from PUBLIC.books where id = ?
     <sql:param value="${bid}" />
    </sql:query>
    <c:forEach var="bookRow" begin="0" items="${books.rows}">
      <jsp:useBean id="bookRow" type="java.util.Map" />
      <jsp:useBean id="addedBook" class="database.BookDetails" scope="page" />
     ...
    <% cart.add(bid, addedBook); %>
 ...
</c:if>
```

2.

```
<c:choose>
   <c:when test="${count == 0}" >
     No records matched your selection.
   </c:when>
   <c:otherwise>
     ${count} records matched your selection.
   </c:otherwise>
 </c:choose>
```

Database Access Tags

<sql:setDataSource> Specifies data source (can also be set by config settings)

```
<%@ taglib prefix="sql" uri="http://java.sun.com/jstl/sql" %>
<sql:setDataSource
  driver="sun.jdbc.odbc.JdbcOdbcDriver"
  url="jdbc:odbc:Northwind"
  user=""
  password=""/>
```

<sql:query> Queries database and stores `ResultSet` in variable

Form 1: use the sql attribute

```
<sql:query var="results" sql="SELECT * FROM …"/>
```

Form 2: put the query in the body of the tag

```
<sql:query var="results">
   SELECT * FROM …
</sql:query>
```

<sql:update>

<sql:param>

<sql:dateParam>

<sql:transaction> Performs the enclosed `<sql:query>` and `<sql:update>` actions as a single transaction

Example

```
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>
<%@ taglib prefix="sql" uri="http://java.sun.com/jstl/sql" %>
<sql:setDataSource
  driver="sun.jdbc.odbc.JdbcOdbcDriver"
  url="jdbc:odbc:Northwind"
  user=""
  password=""/>
<sql:query var="employees">
   SELECT * FROM employees
</sql:query>
<UL>
<c:forEach var="row" items="${employees.rows}">
   <LI><c:out value="${row.firstname}"/>
       <c:out value="${row.lastname}"/>
</c:forEach>
</UL>
```

Various Tags

URL Handling Tags

`<c:import>` Read content from arbitrary URLs. Insert into page, store in variable, or make accessible via a reader

`<c:redirect>` Redirects response to specified URL

`<c:param>` Encodes a request parameter and adds it to a URL

Formatting Tags

`<fmt:formatNumber>` Formats a numeric value as a number, currency value, or percent, in a locale-specific manner

`<fmt:parseNumber>` Reads string representations of number, currency value, or percent

`<fmt:formatDate>`

`<fmt:parseDate>`

`<fmt:timeZone>`

`<fmt:setTimeZone>`

XML Manipulation Tags

Core

`<x:parse>`

Xpath

`<x:if>` `<x:choose>` `<x:when>` `<x:otherwise>` `<x:forEach>`

XSLT

`<x:transform>` `<x:param>`

Inter-tags Collaborations

- The tags usually implicitly or explicitly collaborate with the environment.
- Implicitly, through an interface exposed by the tag-father to the sub-tags. The implicit collaboration is used by the conditional tags.
- Explicitly, when a tag exposes informations like EL variables. JSTL requires to use the `var` name for any attribute that exports information on the tag. For instance the `forEach` tag exports the current item of the cart:

```
<c:forEach var="item" items="${sessionScope.cart.items}">
   ...
</c:forEach>
```

- When a tag exposes more than one information the `var` name is used for the main element we want to expose. For secondary elements, other names should be used. For instance, information on the iteration state can be exported by the `forEach` tag via the `status` attribute.
- When an EL expression exposed by a JSTL tag should be used in a scriptlet of the page a `jsp:useBean` tag must be used for the scripting variable:

```
<c:set var="bookId" value="${param.Remove}"/>
<jsp:useBean id="bookId" type="java.lang.String" />
<% cart.remove(bookId); %>
<sql:query var="books"  dataSource="${applicationScope.bookDS}">
   select * from PUBLIC.books where id = ?
   <sql:param value="${bookId}" />
</sql:query>
```

The `url` tag

- L'elemento `jsp.include` permette l'inclusione di risorse statiche e dinamiche trovate nello stesso contesto con la pagina.
- Il tag `import` della libreria di tag standard `core` è il più semplice mezzo da utilizzare per accedere alle risorse specificate con il loro URL e che possono essere incluse ed elaborate della pagina.

Esempio in cui è letto un documento XML con informazioni sui libri è assegnato il suo contenuto alla variabile `xml`:

```
<c:import url="/books.xml" var="xml" />
<x:parse doc="${xml}" var="booklist" scope="application" />
```

Esempio in cui l'URL viene rescritto con il tag `url`. Il tag include l'ID della sessione nell'URL solo se i cookie sono disabilitati. Altrimenti restituisce l'URL come era. Nell'esempio l'URL deve essere relativo. Il tag `url` utilizzato ha dei sottotag `param` che includono i parametri nell'URL restituito.

```
<c:url var="url" value="/catalog" >
   <c:param name="Add" value="${bookId}" />
</c:url>
<p><strong><a href="${url}">
```

The taglib map in the web.xml DD

- Additional mapping information is required to locate the actual TLD file. The mapping is called the *taglib map. The taglib map* consists of a <taglib> element in the application's web.xml deployment descriptor file. For example, the following is a <taglib> element specifies that the URI `http://www.XXXX.com/begjsp/eshop-functionstaglib` should map to the taglib described by the `/WEB-INF/jsp/eshop-taglib.tld` TLD file :

```
<taglib>
  <taglib-uri>
    http://www.XXXX.com/begjsp/eshop-functions-taglib
  </taglib-uri>
  <taglib-location>
     /WEB-INF/jsp/eshop-taglib.tld
  </taglib-location>
</taglib>
```

- Through the taglib map in the web.xml file, the JSP container locates the actual TLD file. It can then read the TLD file and determine the following:

❑ The location of the implementation code for the tags
❑ The name of the tags
❑ The attributes of each tag and their expected data type
❑ Whether the attribute can be dynamic (and take runtime expressions)
❑ Information interesting to the container, including how to load and run the tags, and whether certain tags work in collaboration with others.

Introduction to Ajax

Ajax, which consists of HTML, JavaScript™ technology, DHTML, and DOM, is an outstanding approach that helps you transform clunky Web interfaces into interactive Ajax applications.

XMLHttpRequest (I)

Using XMLHttpRequest

There are just a *few* of the methods and properties you'll use on this object:

- open(): Sets up a new request to a server.
- send(): Sends a request to a server.
- abort(): Bails out of the current request.
- readyState: Provides the current HTML ready state.
- responseText: The text that the server sends back to respond to a request.

Create an instance of the XMLHttpRequest object

The best approach is to create this object and verify if something goes wrong (many older browsers don't support XMLHttpRequest or Microsoft-IE uses an other object than XMLHttpRequest).

Sandbox

Ajax has a sandbox security model. As a result, your Ajax code (and specifically, the XMLHttpRequest object) can only make requests to the same domain on which it's running. You'll learn lots more about security and Ajax in an upcoming article, but for now realize that code running on your local machine can only make requests to server-side scripts on your local machine.

XMLHttpRequest (II)

Setting the server URL

The first thing you need to determine is the URL of the server to connect to. In most applications, you'll construct this URL from some set of static data combined with data from the form your users work with.

Opening the request

With a URL to connect to, you can configure the request. You'll accomplish this using the open() method on your XMLHttpRequest object. This method takes as many as five parameters:

request-type: The type of request to send. Typical values are GET or POST, but you can also send HEAD requests.

url: The URL to connect to.

asynch: True if you want the request to be asynchronous and false if it should be a synchronous request. This parameter is optional and defaults to true.

username: If authentication is required, you can specify the username here. This is an optional parameter and has no default value.

password: If authentication is required, you can specify the password here. This is an optional parameter and has no default value.

Sending the request

send() takes only a single parameter, the content to send. When you don't need to pass data along through send(), then just pass null .

XMLHttpRequest (III)

Specifying a callback method

When a request is set up and then made the JavaScript method will not wait for the server. So the control will return to the form in the HTML page. Users can keep entering information and the application isn't going to wait on the server.

What happens when the server has finished processing the request?
This is where that onreadystatechange property comes into play. This property allows you to specify a *callback method* that allows the server to *call back* into the Web page's code. The server looks in the XMLHttpRequest object and specifically at the onreadystatechange property. Whatever method is specified by that property is then invoked.

You must set the onreadystatechange property before the request is sent, so the server can look up the property when it finishes answering a request.

Response Processing (I)

Once finished with a request, looks up what method to call in the onreadystatechange property of XMLHttpRequest.

HTTP ready states

An HTTP ready state indicates the state or status of a request. It's helpful in determining whether it's safe to read whatever response text or data that a server might have supplied. The XMLHttpRequest object has a property readystate with five possible values:

- **0**: The request is uninitialized (before you've called open()).
- **1**: The request is set up, but hasn't been sent (before you've called send()).
- **2**: The request was sent and is being processed (you can usually get content headers from the response at this point).
- **3**: The request is being processed; often some partial data is available from the response, but the server hasn't finished with its response.
- **4**: The response is complete; you can get the server's response and use it.

HTTP status codes

What if the server responds to your request and finishes processing, but reports an error? We also need to check the HTTP status.

Response Processing (II)

Reading the response text

We can finally deal with the data sent back by the server using the responseText property of the XMLHttpRequest object when the response is a text (probably a HTML page) or responseXML in the event that the server chooses to respond with XML. Dealing with an XML response is quite different than dealing with plain text and involves parsing, the Document Object Model (DOM), and several other considerations.

How to make a HTTP Request from Javascript (I)

- In order to make an HTTP request to the server using JavaScript:

1. You need an instance of a class that provides you this functionality. Such a class was originally introduced in Internet Explorer as an ActiveX object, called XMLHTTP. Then Mozilla, Safari and other browsers followed, implementing an XMLHttpRequest class that supports the methods and properties of Microsoft's original ActiveX object. As a result, in order to create a cross-browser instance (object) of the required class, you can do:

```
if (window.XMLHttpRequest) { // Mozilla, Safari, ...
   http_request = new XMLHttpRequest();
} else if (window.ActiveXObject) { // IE
  try {
    http_request = new ActiveXObject("Msxml2.XMLHTTP");
  } catch (e) {
     try {
        http_request = new ActiveXObject("Microsoft.XMLHTTP");
     } catch (e2) {
        http_request = false;
     }
  }
}
```

Some versions of some Mozilla browsers won't work properly if the response from the server doesn't have an XML mime-type header. You can use an method call to override the header sent by the server, in case it's not text/xml.

```
http_request = new XMLHttpRequest();
http_request.overrideMimeType('text/xml');
```

How to make a HTTP Request from Javascript (II)

2. You have to tell what you want to do after you receive the server response to your request you must pass a JS function to the `onreadystatechange` property.
```
http_request.onreadystatechange = nameOfTheFunction;
```
3. Next, you need to actually make the request. For this call the open() and send() methods of the HTTP request class, like this:
```
http_request.open('GET', 'http://www.example.org/some.file', true);
http_request.send(null);
```
   - The first parameter of the call to `open()` is the HTTP request method – GET, POST, HEAD etc..
   - The second parameter is the URL of the page you're requesting. As a security feature, *you cannot call pages on 3rd-party domains*.
   - The third parameter sets whether the request is asynchronous (the execution of the function will continue while the response of the server has not yet arrived (Asinchronously as A-jax).

   The parameter to the `send()` method can be any data you want to send to the server if POST-ing the request. It should be in the form of a query string, like:
```
name=value&anothername=othervalue&so=on
```
4. If you want to POST data, you have to change the MIME type of the request using the following line:
```
http_request.setRequestHeader('Content-Type','application/x-www-form-urlencoded');
```
   Otherwise, the server will discard the POSTed data.

Handling the Server Response

Let's see what the function we provided in the HTTP Request should do.

```
http_request.onreadystatechange = nameOfTheFunction;
```

1. The function needs to check for the state of the request. If the state has the value of 4, that means that the full server response is received and it's OK for you to continue processing it.

```
if (http_request.readyState == 4) {
   // everything is good, the response is received
 } else {
 // still not ready
}
```

The full list of the readyState values is as follows:

```
0 (uninitialized); 1 (loading); 2 (loaded); 3 (interactive); 4 (complete);
```

2. Then the function needs to check is the status code of the HTTP server response. In this case we are only interested in 200 OK response.

```
if (http_request.status == 200) {
 // perfect!
} else {
 // there was a problem with the request, for example the response may be
 // a 404 (Not Found) or 500 (Internal Server Error) response codes
}
```

3. You can do whatever you want with the data the server has sent to you. You have two options to access that data:

`http_request.responseText` – will return the server response as a string of text

`http_request.responseXML` – will return the response as an XMLDocument object

“Make a request”

```
<script type="text/javascript" language="javascript">
function makeRequest(url) {
  var http_request = false;
  if (window.XMLHttpRequest) { // Mozilla, Safari,...
     http_request = new XMLHttpRequest();
     if (http_request.overrideMimeType) {
        http_request.overrideMimeType('text/xml');
        // See note below about this line
     }
  } else if (window.ActiveXObject) { // IE
     try {
        http_request = new ActiveXObject("Msxml2.XMLHTTP");
     } catch (e) {
        try {
           http_request = new ActiveXObject("Microsoft.XMLHTTP");
        } catch (e) {}
     }
     }
     if (!http_request) {
        alert('Giving up :( Cannot create an XMLHTTP instance');
        return false;
     }
     http_request.onreadystatechange = function() { alertContents(http_request); };
     http_request.open('GET', url, true);
     http_request.send(null);
}
```

"Alert contents"

```
function alertContents(http_request) {
   if (http_request.readyState == 4) {
     if (http_request.status == 200) {
       alert(http_request.responseText);
     } else {
       alert('There was a problem with the request.');
     }
    }
}
</script>
<span style="cursor: pointer; text-decoration: underline"
  onclick="makeRequest('test.html')">
      Make a request
 </span>
```

1. The user clicks the link "Make a request" in the browser;
2. This calls the `makeRequest()` function with a parameter – the name test.html of an HTML file in the same directory;
3. The request is made and then (`onreadystatechange`) the execution is passed to `alertContents()`;
4. alertContents() checks if the response was received and it's an OK and then `alert()s` the contents of the `test.html` file.

Working with the XML Response

Let's create a valid XML document that we'll request later on. The document (test.xml) contains the following:

```
<?xml version="1.0" ?>
 <root> I'm a test.
</root>
```

In the script we only need to change the request line with:

```
 ...
onclick="makeRequest('test.xml')">
...
```

Then in `alertContents()` we need to replace the `alert()`-ing line `alert(http_request.responseText);` with:

```
var xmldoc = http_request.responseXML;
var root_node = xmldoc.getElementsByTagName('root').item(0);
alert(root_node.firstChild.data);
```

This way we took the XMLDocument object given by responseXML and we used DOM methods to access some data contained in the XML document.

Filters

New capability in version 2.3 of the servlet API : define filters for servlets and JSP pages.

Introduction to Filters

Filters provide a powerful and standard alternative to the nonstandard “servlet chaining” supported by some early servers.

A filter is a program that runs on the server before the servlet or JSP page with which it is associated. A filter can be attached to one or more servlets or JSP pages and can examine the request information going into these resources.

After doing so, it can choose among the following options.

- Invoke the resource (i.e., the servlet or JSP page) in the normal manner.
- Invoke the resource with modified request information.
- Invoke the resource but modify the response before sending it to the client.
- Prevent the resource from being invoked and instead redirect to a different resource, return a particular status code, or generate replacement output.

Benefits

- It lets you encapsulate common behavior in a modular and reusable manner. Do you have 30 different servlets or JSP pages that need to compress their content to decrease download time? Make a single compression filter.
- It lets you separate high-level access decisions from presentation code. For example, do you want to block access from certain sites without modifying the individual pages to which these access restrictions apply? Create an access restriction filter.
- It lets you apply wholesale changes to many different resources. Do you have a bunch of existing resources that should remain unchanged except that the company name should be changed? Make a string replacement filter.

Creating Filters

Creating a filter involves five basic steps:

1. **Create a class that implements the Filter interface.** Your class will need three methods: doFilter, init, and destroy. The doFilter method contains the main filtering code (see Step 2), the init method performs setup operations, and the destroy method does cleanup.
2. **Put the filtering behavior in the doFilter method.** The first argument to the doFilter method is a ServletRequest object. This object gives your filter full access to the incoming information, including form data, cookies, and HTTP request headers. The second argument is a ServletResponse; it is mostly ignored in simple filters. The final argument is a FilterChain; it is used to invoke the servlet or JSP page as described in the next step.

- 3. **Call the doFilter method of the FilterChain object.** The doFilter method of the Filter interface takes a FilterChain object as one of its arguments. When you call the doFilter method of that object, the next associated filter is invoked. If no other filter is associated with the servlet or JSP page, then the servlet or page itself is invoked.
- 4. **Register the filter with the appropriate servlets and JSP pages.** Use the filter and filter-mapping elements in the deployment descriptor (*web.xml).*
- 5. **Disable the invoker servlet.** Prevent users from bypassing filter settings by using default servlet URLs.

Filter Interface

All filters must implement `javax.servlet.Filter`. This interface comprises three methods: `doFilter`, `init`, and `destroy`.

1. `public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) throws ServletException, IOException`

The `doFilter` method is executed each time a filter is invoked (i.e., once for each request for a servlet or JSP page with which the filter is associated).

The first argument is the `ServletRequest` associated with the incoming request. For simple filters, most of your filter logic is based on this object. Cast the object to `HttpServletRequest` if you are dealing with HTTP requests and you need access to methods such as `getHeader` or `getCookies` that are unavailable in `ServletRequest`.

The final argument to `doFilter` is a `FilterChain` object. You call `doFilter` on this object to invoke the next filter that is associated with the servlet or JSP page. In no other filters are in effect, then the call to `doFilter` invokes the servlet or JSP page itself.

2. `public void init(FilterConfig config) throws ServletException`

The `init` method is executed only when the filter is first initialized. It is common to use `init` to store the `FilterConfig` object in a field so that the `doFilter` method can access the servlet context or the filter name.

3. `public void destroy()`

Use this method for cleanup tasks like closing files or database connection pools that are used by the filter.

Register the filter with servlets and JSPs

- Version 2.3 of the deployment descriptor introduced two elements for use with filters: filter and filter-mapping. The filter element registers a filtering object with the system. The filter-mapping element specifies the URLs to which the filtering object applies.
- The `filter` element goes near the top of deployment descriptor (*web.xml), before* any filter-mapping, servlet, or servlet-mapping elements and contains six possible subelements: `icon`, `filter-name`, `display-name`, `description`, `filter-class`, `init-param`.
- The filter-mapping element goes in the *web.xml* file after the filter element but before the servlet element. It contains three possible subelements: filter-name, url-pattern, servlet-name.

Example

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application
  2.3//EN‟ "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
  <filter>
    <filter-name>MyFilter</filter-name>
    <filter-class>myPackage.FilterClass</filter-class>
  </filter>
<!-- ... -->
  <filter-mapping>
    <filter-name>MyFilter</filter-name>
    <url-pattern>/someDirectory/SomePage.jsp</url-pattern>
  </filter-mapping>
</web-app>
```

Example: A Reporting File

Write a simple filter that prints a message to standard output whenever the as

```
package core.filters;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;
public class ReportFilter implements Filter {
  public void doFilter(ServletRequest request, ServletResponse response, FilterChain
   chain) throws ServletException, IOException {
      HttpServletRequest req = (HttpServletRequest)request;
      System.out.println(req.getRemoteHost() + " tried to access " +
            req.getRequestURL() + " on " + new Date() + ".");
      chain.doFilter(request,response);
}
public void init(FilterConfig config) throws ServletException {}
public void destroy(){}
}
```

web.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
<!-- Register the name "Reporter" for ReportFilter. -->
  <filter>
     <filter-name>Reporter</filter-name>
     <filter-class>core.filters.ReportFilter</filter-class>
  </filter>
  <!-- ... -->
  <!-- Apply the Reporter filter to home page. -->
  <filter-mapping>
     <filter-name>Reporter</filter-name>
     <url-pattern>/index.jsp</url-pattern>
   </filter-mapping>
<!-- Also apply the Reporter filter to the servlet named "TodaysSpecial". -->
  <filter-mapping>
     <filter-name>Reporter</filter-name>
     <servlet-name>TodaysSpecial</servlet-name>
  </filter-mapping>
  <!-- ... -->
  <!-- Give a name to the servlet so that filters can be applied to it. -->
  <servlet>
    <servlet-name>TodaysSpecial</servlet-name>
    <servlet-class>core.TodaysSpecialServlet</servlet-class>
</servlet>
<!-- ... -->
<!-- Make /TodaysSpecial invoke the servlet named TodaysSpecial . -->
  <servlet-mapping>
    <servlet-name>TodaysSpecial</servlet-name>
    <url-pattern>/TodaysSpecial</url-pattern>
  </servlet-mapping>
  <!-- Turn off invoker. Send requests to index.jsp. -->
  <servlet-mapping>
    <servlet-name>Redirector</servlet-name>
    <url-pattern>/servlet/*</url-pattern>
  </servlet-mapping>
  <!-- ... -->
</web-app>
```

Accessing the Servlet Context

The only solution to access application-wide parameters or ServletContext is to use getServletContext on FilterConfig in the init method of the Filter interface:

```
public class LogFilter implements Filter {
  protected FilterConfig config;
  private ServletContext context;
  private String filterName;
  public void init(FilterConfig config)throws ServletException {
  // In case it is needed by subclass.
    this.config = config;
    context = config.getServletContext();
    filterName = config.getFilterName();
  }
  public void doFilter(ServletRequest request, ServletResponse response,
        FilterChain chain) throws ServletException, IOException {
    HttpServletRequest req = (HttpServletRequest)request;
  context.log(req.getRemoteHost()+" tried to access "+req.getRequestURL()
  +" on " + new Date() + ". " + "(Reported by " + filterName + ".)");
    chain.doFilter(request,response);
  }
  public void destroy() {}
}
```

web.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC
  "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
  "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
<!-- ... -->
  <!-- Register the name "Logger" for LogFilter. -->
  <filter>
    <filter-name>Logger</filter-name>
    <filter-class>core.filters.LogFilter</filter-class>
  </filter>
<!-- ... -->
<!-- Apply the Logger filter to all servlets and JSP pages. -->
  <filter-mapping>
    <filter-name>Logger</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>
<!-- ... -->
</web-app>
```

Blocking the Response with a Filter

- If your filter detects an unusual situation and wants to prevent the original resource from being invoked you can omit the call to the doFilter method of the FilterChain object. Instead, the filter can redirect the user to a different page (e.g., with a call to response.sendRedirect) or generate the response itself (e.g., by calling get-Writer on the response and sending output, just as with a regular servlet). Don't forget to cast `ServletRequest` and `ServletResponse` with `HttpServletRequest` and `HttpServletResponse`.

```
public void doFilter(ServletRequest request, ServletResponse response,
     FilterChain chain) throws ServletException, IOException {
  HttpServletRequest req = (HttpServletRequest)request;
  HttpServletResponse res = (HttpServletResponse)response;
  if (isUnusualCondition(req)) {
    res.sendRedirect("http://www.somesite.com");
  } else {
    chain.doFilter(req,res);
  }
}
```

Modifying the Response with a Filter

- Filters can block access to resources or invoke them normally, but also can change the response that a resource generates.
- The second argument to `doFilter` (the `ServletResponse`) gives the filter a way to send new output to a client, but it doesn't give the filter access to the output of the servlet or JSP pag: when the `doFilter` method is first invoked, the servlet or JSP page hasn't even executed yet. Once you call the `doFilter` method of the `FilterChain` object, it appears to be too late to modify the response - data has already been sent to the client.
- The solution is to change the response object that is passed to the `doFilter` method of the `FilterChain` object. You typically create a version that buffers up all the output that the servlet or JSP page generates. The servlet 2.3 API provides a useful resource for this purpose: the `HttpServletResponseWrapper` class. This class implements `HttpServletResponse`.

Five steps:

1. Create a response wrapper by extending HttpServletResponseWrapper.
2. Provide a PrintWriter that buffers output by overriding the get-Writer method to return a PrintWriter that saves everything sent to it and stores that result in a field that can be accessed later.
3. Pass that wrapper to doFilter.
4. Extract and modify the output. After the call to the doFilter method of the FilterChain, the output of the original resource is available to you through whatever mechanism you provided in Step 2. You can modify or replace it as appropriate for your application.

Defining Custom JSP Tag Libraries

JSP 1.1 introduced an extremely valuable new capability: the ability to create your own JSP tags. You define how a tag, its attributes, and its body are interpreted, then group your tags into collections called *tag libraries* that can be used in any number of JSP files.

Tag Files

- A tag file is a special kind of custom action that is developed much like a JSP page. Unlike JSP pages, tag files can be invoked from within JSP pages in the same way Java Standard Tag Library (JSTL) tags are used.
- Tag files, for instance, can be passed parameters to use, and they can expose values to the calling JSP page, and invoke fragments of JSP code.
- A developer implements a tag file using a text file and places it within a special directory of the Web application.
- From there, its invocation and usages are much like any other custom tag an application may use.

Example

1.The JSP page that invokes the tag file :

```
<%@ taglib prefix="beginjsp" tagdir="/WEB-INF/tags" %>
<beginjsp:todaysdate/>
```

2.The `taglib` directive , using the `tagdir` attribute, also identifies the folder in which the tag files can be found：`/WEB-ING/tags`. To create the tag itself, a file called `todaysdate.tag` is placed in the `/WEB-INF/tags` folder of the Web application. Here is the file content:

```
<%@ tag body-content="empty" %>
<jsp:useBean id="today" scope="page" class="java.util.Date"/>
Today's date is: ${today.date} / ${today.month} / ${today.year + 1900}
```

3. Behind the scenes, the tag file is converted by the servlet container a Java source file, and then compiles it, much as it would a normal JSP page.

More About Tag Files

Advantages of Tag Files

- ❑ Code reuse
- ❑ Hiding complexity
- ❑ Separation of concerns
- ❑ Simplicity
- ❑ Flexible packaging

Scope and implicit objects

- JSP files and the tag files they invoke share the `application`, `session`, and `request` scopes, while each has their own `page` scope.
- The `page` scope variables of the JSP page cannot be used inside the tag file, and vice versa.

Using directives in tag files

- A number of directives can be used within JSP tag files, such as `tag`, `taglib`, `include`, `attribute`, and variable.
- As in normal JSP files, directives are `included` to affect the overall processing of the page being constructed.
- You have already seen the tag directive being used in a preceding example. Some of these directives are used in standard JSP pages; others, such as `attribute` and `variable`, can only be used within tag file custom actions.

Example

1. The address field in a form to be generated can be set with scripting elements extracted from an `address` parameter as in the followings:

```
<TD ALIGN='right'>Address:</TD>
<TD>
 <% String addressVal =request.getParameter("address");
    if (addressVal == null ) addressVal = ""; %>
 <INPUT TYPE='text' NAME='address' VALUE='<%=addressVal %>' SIZE='50'>
</TD>
```

2. With custom tags this code can be substitute with a reference to a unique, empty tag as in the followings:

```
<TD ALIGN='right'>Address:</TD>
  <TD>
     <INPUT TYPE='text' NAME='address' VALUE='<order:getReqParam name="address"/>'
          SIZE='50'>
  </TD>
```

The `tag` Directive

The tag directive plays a similar role to the page directive used in JSP pages.

```
<%@ tag body-content="empty" import="java.util.ArrayList" %>
```

- It handles a number of useful attributes that affect the processing and presentation of the tag file generally, and how it is used in a JSP file

Attribute Description

`body-content` optional, it indicates the nature of the body of the tag when it is invoked. Values can be empty, tagdependent, scriptless. Default: scriptless

`description` An optional description for the tag.

`display-name` An optional simple name for the tag. May be used by IDE or XML tools. The default value is the name of the file without the .tag extension

`example` A simple illustration of how the tag functions.

`import` An optional attribute used to import packages or classes into the scope of the tag.

`isElIgnored` Indicates that EL expressions will be ignored. Default : false.

`language` The scripting language used in the tag. The default is Java.

`large-icon` Optional, it can be used to relate an image file with the tag.

`pageEncoding` The character encoding for this file. IANA encoding names must be used.

`small-icon` Optional attribute used to relate an image file with the tag.

`dynamic-attributes` Indicates that dynamic attributes can be passed to this tag, and the value of this attribute names a page scope Map object where their name and value pairs are stored.

The `taglib` Directive

- A tag file can also make use of the taglib directive in order to make use of other custom actions. This is extremely useful where you need to use JSTL custom tags within a tag file.The directive `taglib` indicates to the JSP compiler to include the library in the current compilation unit. The directive associates to the library an ID as a tag prefix.
- The prefix is defined with the `prefix` attribute. `jsp`, `jspx`, `java`, `javax`, `servlet`, `sun` are `sunw` reserved attributes. An *uri* attribute is used to indicate where the library with the specified prefix is located. You can use the `tagdir` attribute in order to specify a tag library in the standard directory `/WEB-INF/tags`.

Examples:

```
<%--- a directive taglib that uses the uri attribute --%>
<%@ taglib uri="http://www.datamat.com/tags" prefix="data" %>

<%--- a directive taglib that uses the tagdir attribute --%>
<%@ taglib tagdir="/WEB-INF/tags/ecommerce" prefix="data" %>
```

- In the JSP document you must use the `<jsp:root>` element for including a definition in the namespace `xmlns`:

Example:

```
<jsp:root ... xmlns:data="http://www.datamat.com/tags" ... >
...
</jsp:root>
```

The `include` Directive

- The include directive works in much the same way as the include directive in a normal JSP file. It is used to include static and dynamic content. For instance, within a tag file, the include directive could be used to include the content from an HTML file, or the dynamic result of a tag file fragment.

```
<%@ tag body-content=”empty” %>
<%@ include file=”/includeme.html” %>
<P>
<%@ include file=”/WEB-INF/tagf/includedmetoo.tagf” %>
```

- Each of these files would be included in turn and represent the output of the tag file. The tag file fragment, `includedmetoo.tagf`, is treated as a segment of the tag file; it can therefore include any part of the tag file as long as the result of the tag file itself and its fragments make up a legitimate tag file.

The `attribute` Directive

- The attribute directive allows tag files to be passed values from the invoking JSP page. This gives real power to the use of tag files, as it does to other custom actions. When a tag file uses the attribute directive, JSP pages can pass values through to the tag file for processing.
- A simple “hello, world” example of a new tag file called `printstring.tag`:

```
<%@ tag body-content=”empty” %>
<%@ taglib prefix=”c” uri=”http://java.sun.com/jsp/jstl/core” %>
<%@ attribute name=”value” required=”true”%>
<c:out value=”${value}”/>
```

- An invoking JSP file may appear as follows:

```
<%@ taglib prefix=”beginjsp” tagdir=”/WEB-INF/tags” %>
<beginjsp:printstring value=”hello, world”/>
```

The attribute directive can use several attributes.

`fragment` This attribute indicates whether the attribute being passed is a fragment attribute (true). The default value of this optional attribute is false.

`name` The name that the value this attribute represents will take within the tag file’s scope. Within the tag file, the value of this attribute is referred to by this name. It must be unique. This value also forms the name of the tag attribute that will provide the value to the tag file when it is called in the JSP page. This attribute is required.

`required` Indicates that the attribute is required (true) when the custom action is being invoked in the JSP page. The default value of this optional attribute is false.

`type` The type of the attribute that is being passed. The value of this attribute is a fully qualified class name. The default is java.lang.String. This attribute is optional.

The `variable` Directive

- The variable directive is used to expose values of the tag file to the calling JSP page. This becomes extremely useful when the result of a tag file is not to be simply displayed.
- Think of the variable directive as similar to a value being returned from the tag file, much like a value is returned from a normal Java method.

- Attributes in a `variable` directive

`name-given` or `name-from-attribute` One of these tags must be used in the variable directive, but they cannot be used together. The name-given attribute gives the returning value a name that the calling JSP page can use when referencing it. The name-from-attribute attribute also gives the returning value a name the calling JSP page can use when referencing it, but the name is that of an attribute for this custom action.

`alias` Used in conjunction with the name-from-attribute attribute to synchronize names used between a tag file and a JSP page. The alias forms the tag's name for the variable, as opposed to the name provided by the attribute in name-from-attribute. This attribute is required when name-from-attribute is used.

`variable-class` The class name of the variable being returned. This attribute is optional and its default value is java.lang.String.

`declare` A message to the container to declare the returning value in the JSP page. The default is true. This attribute is optional.

`scope` The scope of the returning variable, with respect to the tag itself. Values here are AT_BEGIN, NESTED, and AT_END. The default is NESTED. This attribute is optional.

Example of `variable` Directive

- A String will be created in a tag file and displayed on the calling JSP page. The tag file `returnvalue.tag` contains the following:

```
<%@ tag body-content="empty" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ variable name-given="foo" scope="AT_END"%>
<c:set var="foo" value="this is from the tag"/>
```

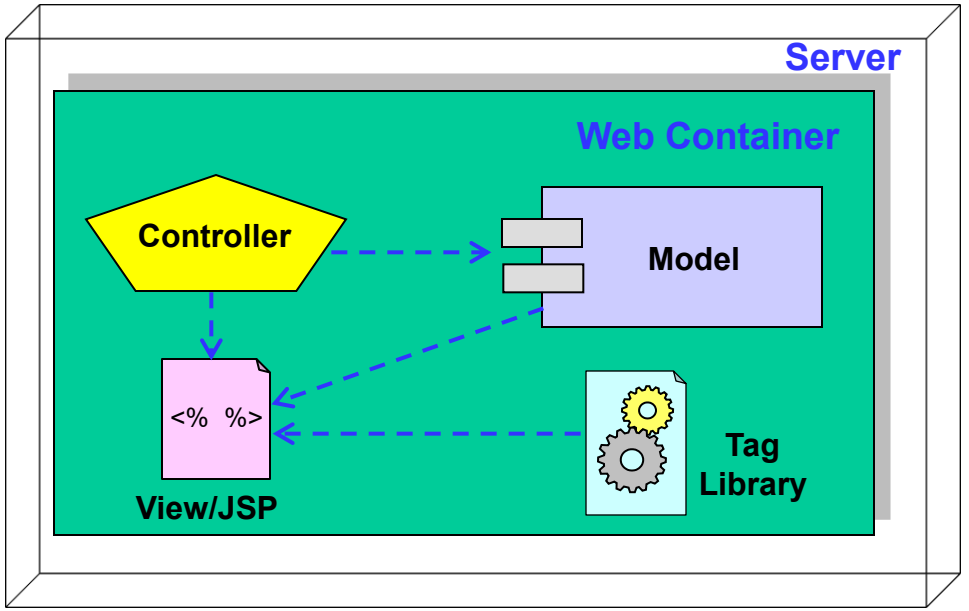
- The variable directive here defines the variable that will be provided to the calling JSP page. This variable, `foo`, will be available to the JSP page after the tag has been executed due to the scope attribute value of `AT_END`. The `scope` value here refers to the scope of the variable with respect to the tag itself and its invocation on the calling JSP page.
- The variable `foo` is then declared and assigned a value using the JSTL `c:set` tag. By default, variables created with the c:set tag are placed in the page scope. As the variable directive has been used here, and the declare attribute has been left as true (the default), the container has declared a variable named `foo` in the page scope of the JSP page.
- The JSP page can then print out this value using the following:

```
<%@ taglib prefix="beginjsp" tagdir="/WEB-INF/tags" %>
<beginjsp:returnvalue/>
This is the value of foo:${pageScope.foo}
```

Result: `This is the value of foo: this is from the tag`

Custom Tag Libraries

- A custom tag library is a collection of tag handlers plus a library descriptor.
- A custom tag library is a Web component, part of a Web application. The library can access objects in the application where a View can access them.
- The tag libraries are delivered with a documentation that specifies each tag with such the following attributes:
  - scope,
  - behavior,
  - content type,
  - attributes
  - attribute description.



Syntax Rules for Tags

1. Standard syntax of tags:

   <prefix:name {attribute={"value"|'value'}}*> body</prefix:name>

Example:

```
<calcio:intestazione alignment="center" color="blue">
    Form di registrazione del campionato di calcio </calcio:intestazione>
```

2. Empty tags:

   <prefix:name {attribute={"value"|'value'}}* />

Esempio:

```
<calcio:getReqParam name="indirizzo" />
```

3. I nomi dei tag, degli attributi e dei prefissi sono case sensitive. Ad esempio:

   `<calcio:intestazione>` è diverso da `<calcio:Intestazione>`

   `<calcio:intestazione>` è diverso da `<Calcio:intestazione>`

   `<calcio:intestazione color="blue">` è diverso da `<calcio:intestazione COLOR="blue">`

4. I tag devono seguire regole di aniddamento:

   <tag1> <tag2> </tag2></tag1>

Esempio :

```
<calcio:verifStato>
   <calcio:scorreErrori>
      <%-- codice JSP  --%>
   </calcio:scorreErrori>
</calcio:verifStato>
```

Libreria di tag. Esempio (I)

Tag `getReqParam`

Inserisce il valore di un parametro della richiesta nell'output. Se il parametro non esiste, si utilizza sia un valore predefinito (se è fornito) o la stringa vuota.

Contenuto: Tag vuoto.

Attributo: `name`

Obbligatorio. Rappresenta il nome del parametro della richiesta.

Attributo: `default`

Opzionale. Rappresenta il valore predefinito del parametro della richiesta.

Esempio:

```
<calcio:getReqParam name="codicePaese" default="IT"/>
```

Tag `intestazione`

Genera una tabella HTML nascosta che crea un'intestazione colorata alla pagina.

Contenuto: Contiene codice JSP.

Attributo: `alignment`

Obbligatorio. Specifica l'allineamento del testo dell'intestazione. Valori accettate: `left`, `center`, `right`.

Attributo: `color`

Obbligatorio. Specifica il colore della casella di contorno dell'intestazione. Valori accettate: nomi di colori HTML o RGB.

Esempio:

Libreria di tag. Esempio (II)

Tag *verifStato*

Crea un contenuto condizionato se l'attributo "stato" è impostato e se l'oggetto Stato indica "fallito". Utilizzato quando una pagina contenente un form è riproposta dopo un errore.

Contenuto: Contiene codice JSP.

Esempio:

```
<calcio:verifStato>
   <%-- codice JSP  --%>
</calcio:verifStato>
```

Tag *scorreErrori*

Scorre tra oggetti eccezione in un oggetto Stato. Deve essere utilizzato dentro il tag `verifStato`.

Contenuto: Contiene codice JSP.

Esempio:

```
<calcio:verifStato>
   <calcio:scorreErrori>
      <%-- codice JSP  --%>
   </calcio:scorreErrori>
</calcio:verifStato>
```

Tag *getMessCorrente*

Scrive messaggi di errore. Deve essere utilizzato dentro il tag `scorreErrori`.

Contenuto: Tag vuoto.

Esempio:

```
<calcio:verifStato>
   <calcio:scorreErrori>
      <calcio:getMessCorrente/>
   </calcio:scorreErrori>
</calcio:verifStato>
```

Sviluppare JSP con una libreria di tag

- Una libreria di tag personalizzati ha due parti:
  - File `.jar` con le classi che gestiscono i tag
  - Tag Library Descriptor (TLD), un file XML con estensione .tld in cui i tag ricevono un nome e una struttura.
- Per utilizzare una libreria di tag in un'applicazione Web si deve dichiarare il TLD nel Deployment Descriptor (DD) dell'applicazione.

Esempio di dichiarazione di una libreria tag nel DD che mappa il file fisico TLD in un URI arbitrario:

```
<taglib>
        <taglib-uri>http://www.calcio.it/taglib</taglib-uri>
        <taglib-location>/WEB-INF/taglib.tld</taglib-location>
</taglib>
```

- Il file `.tld` si memorizza nella directory `WEB-INF`, invece il file `.jar` si memorizza in `WEB-INF\lib` .
- La pagina JSP può utilizzare una libreria di tag indicando con la direttiva `taglib` al compilatore JSP di accedere al file TLD. La direttiva include l'URI e il prefisso dei tag personalizzati.

Esempio

```
<%@ page session="false" %>
<%@ taglib uri="http://www.calcio.it/taglib" prefix="calcio" %>
<HTML><HEAD><TITLE> Form di Registrazione </TITLE></HEAD>
<BODY BGCOLOR='white'>
```

Utilizzo dei custom tag nel codice JSP

Utilizzo dei tag personalizzati condizionali (verifica una condizione per decidere se deve o no inserire un certo contenuto HTML).

Esempio con scriptlet:

```
<%  if ( (stato != null) && !stato.isSuccessful() ) {
%>
   <%-- codice JSP per messaggi di errore --%>
<%
  } // end IF stato
%>
```

Esempio equivalente con tag personalizzati:

```
<calcio:verifStato>
   <%-- codice JSP per messaggi di errore --%>
</calcio:verifStato>
```

Utilizzo dei tag personalizzati iterativi (per liste UL o OL o righe in tabelle)

Esempio con scriptlet:

```
Ci sono problemi con la sua richiesta:
<UL><%
    Iterator errori = stato.getExceptions();
    while ( errori.hasNext() ) {
      Exception ex = (Exception) errori.next();
%>
    <LI><%= ex.getMessage() %>
<%
    } // end WHILE
%></UL>
```

Esempio equivalente con tag personalizzati:

```
Ci sono problemi con la sua richiesta:
<UL>
  <calcio:scorreErrori>
    <LI><calcio:getMessCorrente/>
  </calcio:scorreErrori></UL>
```

The “Model 2” Architecture

Why Combine Servlets and JSPs?

Typical picture: use JSP to make it easier to develop and maintain the HTML content

- For simple dynamic code, call servlet code from scripting elements
- For slightly more complex applications, use custom classes called from scripting elements
- For moderately complex applications, use beans and custom tags

But, that's not enough

- For complex processing, starting with JSP is awkward

The assumption behind JSP is that a *single page gives a single basic look.*

Possibilities to handle a single request:

1. **Servlet only. Works well when:**

– Output is a binary type. E.g.: an image
– There is *no* output. E.g.: you are doing forwarding or redirection.
– Format/layout of page is highly variable. E.g.: portal.

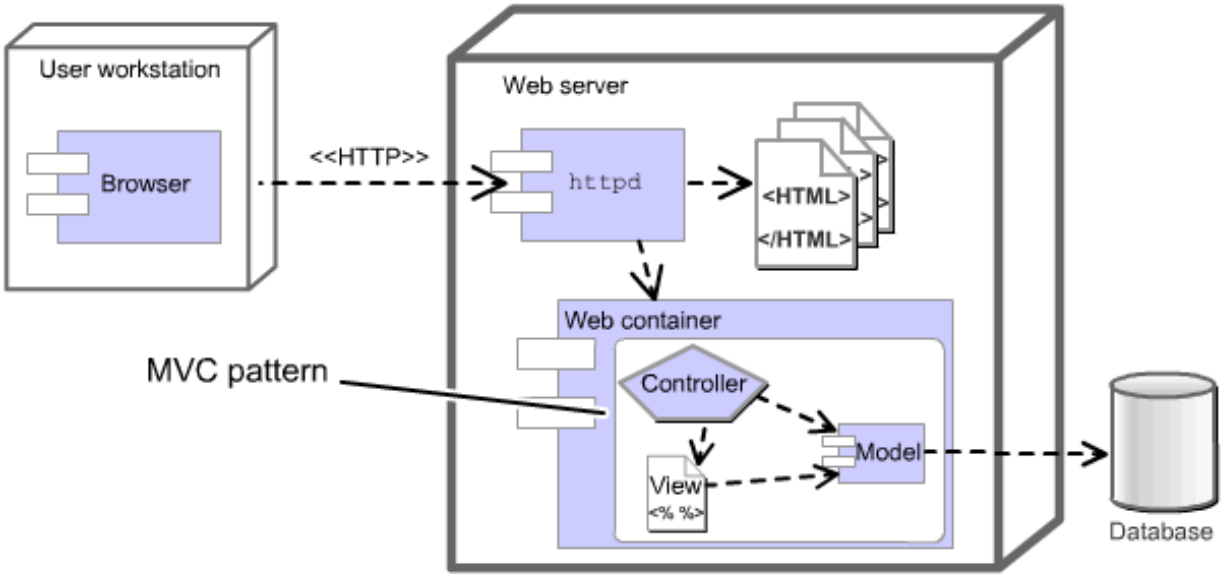
2. **JSP only. Works well when:**

– Output is mostly character data. E.g.: HTML
– Format/layout mostly fixed.

3. **Combination (MVC architecture). Needed when:**

– A single request will result in multiple substantially different looking results.
– You have a large development team with different team members doing the Web development and the business logic.
– You perform complicated data processing, but have a relatively fixed layout.

Web Server Architecture Using the Model 2 Design



Using servlets and JSP pages together facilitates proper separation of concerns using a variation on the Model-View-Controller (MVC) design pattern.

- A servlet acts as the Controller, which verifies form data, updates the Model with the form data, and selects the next View as the response.
- A JSP page acts as the View, which renders the HTML response, retrieving data from the Model necessary to generate the response, and provides HTML forms to permit user interaction.
- Java technology classes act as the Model, which implements the business logic of the Web application. It can include DAO (Data Access Objects) objects that are hidden from web components.

Implementing MVC with RequestDispatcher

1. **Define beans to represent the data**
2. **Use a servlet to handle requests**

Servlet reads request parameters, checks for missing and malformed data, etc.

3. **Populate the beans**

   The servlet invokes business logic (application-specific code) or data-access code to obtain the results. Results are placed in the beans that were defined in step 1.

4. **Store the bean in the request, session, or servlet context**

   The servlet calls `setAttribute` on the request, session, or servlet context objects to store a reference to the beans that represent the results of the request.

5. **Forward the request to a JSP page.**

   The servlet determines which JSP page is appropriate to the situation and uses the forward method of `RequestDispatcher` to transfer control to that page.

6. **Extract the data from the beans.**

   The JSP page accesses beans with `jsp:useBean` and a scope matching the location of step 4. The page then uses `jsp:getProperty` to output the bean properties.

   The JSP page does not create or modify the bean; it merely extracts and displays data that the servlet created.

Request Forwarding Example

```
public void doGet(HttpServletRequest request, HttpServletResponse
   response) throws ServletException, IOException {
   String operation = request.getParameter("operation");
   if (operation == null) {
      operation = "unknown";
   }
   String address;
   if (operation.equals("order")) {
      address = "/WEB-INF/Order.jsp";
   } else if (operation.equals("cancel")) {
      address = "/WEB-INF/Cancel.jsp";
   } else {
      address = "/WEB-INF/UnknownOperation.jsp";
   }
   RequestDispatcher dispatcher = request.getRequestDispatcher(address);
   dispatcher.forward(request, response);
}
```

In the MVC model, the servlet, not the JSP page, should create all the data objects. You should use `<jsp:useBean ... type="package.Class" />` instead of `<jsp:useBean ... class="package.Class" />`

In the MVC model, the JSP page should not modify the objects

You should use `jsp:getProperty`, but not `jsp:setProperty`.

jsp:useBean in MVC vs. in Standalone JSP Pages

1. **The JSP page should not create the objects.** The servlet, not the JSP page, should create all the data objects. So, to guarantee that the JSP page will not create objects, you should use:

```
<jsp:useBean ... type="package.Class" />
```

instead of:

```
<jsp:useBean ... class="package.Class" />
```

2. **The JSP page should not modify the objects.** So, you should use `jsp:getProperty` but not `jsp:setProperty`.

Scope Alternative:

Request

```
<jsp:useBean id="..." type="..." scope="request" />
```

Session

```
<jsp:useBean id="..." type="..." scope="session" />
```

Application

```
<jsp:useBean id="..." type="..." scope="application" />
```

Page. This scope is not used in MVC (Model 2) architecture

```
<jsp:useBean id="..." type="..." scope="page" />
```

or just

```
<jsp:useBean id="..." type="..." />
```

Request and Session -Based Data Sharing

Request -Based Data Sharing

Servlet

```
ValueObject value = new ValueObject(...);
request.setAttribute("key", value);
RequestDispatcher dispatcher = request.getRequestDispatcher
("/WEB-INF/SomePage.jsp");
dispatcher.forward(request, response);
```

JSP 1.2

```
<jsp:useBean id="key" type="somePackage.ValueObject‟ scope="request" />
<jsp:getProperty name="key" property="someProperty" />
```

JSP 2.0

```
${key.someProperty}
```

Session -Based Data Sharing

Servlet

```
ValueObject value = new ValueObject(...);
HttpSession session = request.getSession();
session.setAttribute("key", value);
RequestDispatcher dispatcher = request.getRequestDispatcher
("/WEB-INF/SomePage.jsp");
dispatcher.forward(request, response);
```

JSP 1.2

```
<jsp:useBean id="key" type="somePackage.ValueObject‟ scope="session" />
<jsp:getProperty name="key" property="someProperty" />
```

JSP 2.0

```
${key.someProperty}
```

ServletContext-Based Data Sharing

Servlet

```
synchronized(this) {
  ValueObject value = new ValueObject(...);
  getServletContext().setAttribute("key", value);
  RequestDispatcher dispatcher = request.getRequestDispatcher
    ("/WEB-INF/SomePage.jsp");
  dispatcher.forward(request, response);
}
```

JSP 1.2

```
<jsp:useBean id="key" type="somePackage.ValueObject"
   scope="application" />
<jsp:getProperty name="key" property="someProperty" />
```

JSP 2.0

```
${key someProperty}
```

Relative URLs in JSP Pages

Issue: Forwarding with a request dispatcher is transparent to the client. *Original URL* is only URL browser knows about.

Why does this matter?

What will browser do with tags like the following:

```
<IMG SRC="foo.gif" …>
<LINK REL=STYLESHEET HREF="JSP-Styles.css" TYPE="text/css">
<A HREF="bar.jsp">…</A>
```

Answer: browser treats them as relative to *servlet URL*

Simplest solution:

Use URLs that begin with a slash.

Bank Account Balances

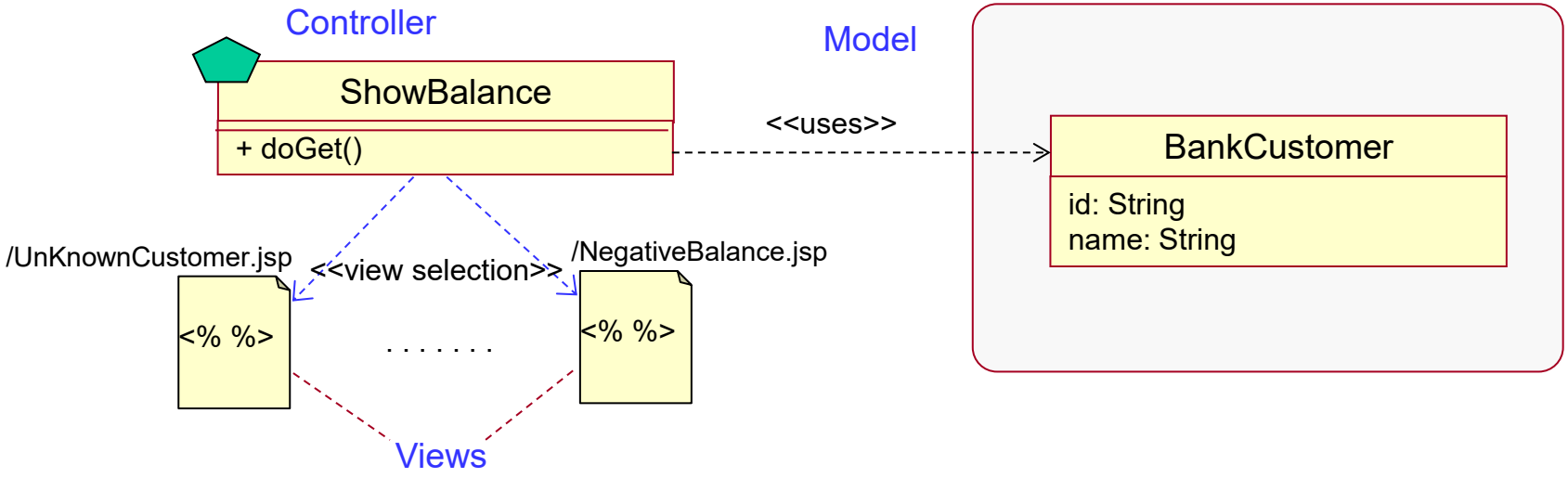
Bean: BankCustomer

Servlet that populates bean and forwards to appropriate JSP pages:

- Reads customer ID
- Calls data-access code to populate BankCustomer
- Uses current balance to decide on appropriate result page

JSP pages to display results

- Negative balance: warning page
- Regular balance: standard page
- High balance: page with advertisements added
- Unknown customer ID: error page



MVC Implementation

ShowBalance.java

```
public class ShowBalance extends HttpServlet {
 public void doGet(HttpServletRequest request, HttpServletResponse response) throws
   ServletException, IOException {
  BankCustomer customer = BankCustomer.getCustomer(request.getParameter("id"));
  String address;
  if (customer == null) {
    address = "/WEB-INF/bank-account/UnknownCustomer.jsp";
  } else if (customer.getBalance() < 0) {
    address = "/WEB-INF/bank-account/NegativeBalance.jsp";
    request.setAttribute("badCustomer", customer);
  }  …
  RequestDispatcher dispatcher = request.getRequestDispatcher(address);
  dispatcher.forward(request, response);
   …
 }
```

Bank Account Balances: JSP 1.2 Code (Negative Balance)

NegativeBalance.jsp

```
<BODY>
<TABLE BORDER=5 ALIGN="CENTER">
<TR><TH CLASS="TITLE"> We Know Where You Live!</TABLE>
<P>
<IMG SRC="/bank-support/Club.gif" ALIGN="LEFT">
Watch out, ${badCustomer.firstName}, we know where you live.
<P>
Pay us the $${badCustomer.balanceNoSign} you owe us before it is too
  late!
</BODY></HTML>
```

Bank Account Balances: JSP 2.0 Code (Negative Balance)

```
<BODY>
<TABLE BORDER=5 ALIGN="CENTER">
<TR><TH CLASS="TITLE">
We Know Where You Live!</TABLE>
<P>
<IMG SRC="/bank-support/Club.gif" ALIGN="LEFT">
Watch out,
${badCustomer.firstName},
we know where you live.
<P>
Pay us the $${badCustomer.balanceNoSign}
you owe us before it is too late!
</BODY></HTML>
```

Comparing Data-Sharing Approaches: Request

- Goal: Display a random number to the user
- Type of sharing: each request should result in a new number, so request-based sharing is appropriate.

Java Bean

```
public class NumberBean {
  private double num = 0;
  public NumberBean(double number) {
    setNumber(number);
  }
  public double getNumber() {
    return(num);
  }
  public void setNumber(double number) {
    num = number;
  }
}
```

Request-Based Sharing: Servlet and jsp

Servlet

```
public class RandomNumberServlet extends HttpServlet {
  public void doGet(HttpServletRequest request, HttpServletResponse
  response) throws ServletException, IOException {
    NumberBean bean = new NumberBean(Math.random());
    request.setAttribute("randomNum", bean);
    String address = "/WEB-INF/mvc-sharing/RandomNum.jsp";
    RequestDispatcher dispatcher = request.getRequestDispatcher(address);
    dispatcher.forward(request, response);
  }
}
```

Request-Based Sharing: JSP 1.2

```
<BODY>
<jsp:useBean id="randomNum" type="NumberBean"scope="request" />
<H2>Random Number: <jsp:getProperty name="randomNum" property="number" />
</H2>
</BODY></HTML>
```

Request-Based Sharing: JSP 2.0

```
…
<BODY>
<H2>Random Number: ${randomNum.number}
</H2>
</BODY></HTML>
```

Session-Based Sharing

- **Goal:** Display users' first and last names. If the users fail to tell us their name, we want to use whatever name they gave us previously. If the users do not explicitly specify a name and no previous name is found, a warning should be displayed.
- **Type of sharing:** Data is stored for each client, so session-based sharing is appropriate.

Java Bean

```
public class NameBean {
  private String firstName = "Missing first name";
  private String lastName = "Missing last name";
  public NameBean(String firstName, String lastName) {
    setFirstName(firstName);
    setLastName(lastName);
  }
  public String getFirstName() {
    return(firstName);
  }
  public void setFirstName(String firstName) {
    if (!isMissing(firstName)) {
      this.firstName = firstName;
  }
}
  ... // getLastName, setLastName
  private boolean isMissing(String value) {
    return((value == null) || (value.trim().equals("")));
  }
}
```

Session-Based Sharing: servlet

```
public class RegistrationServlet extends HttpServlet {
  public void doGet(HttpServletRequest request, HttpServletResponse
  response) throws ServletException, IOException {
   HttpSession session = request.getSession();
   NameBean nameBean = (NameBean)session.getAttribute("nameBean");
   if (nameBean == null) {
     String firstName = request.getParameter("firstName");
     String lastName = request.getParameter("lastName");
     nameBean = new NameBean(firstName, lastName);
     session.setAttribute("nameBean", nameBean);
  }
  String address = "/WEB-INF/mvc-sharing/ShowName.jsp";
  RequestDispatcher dispatcher = request.getRequestDispatcher(address);
  dispatcher.forward(request, response);
 }
}
```

Session-Based Sharing: jsp

JSP 1.2

```
. . .
<BODY>
<H1>Thanks for Registering</H1>
<jsp:useBean id="nameBean" type="NameBean" scope="session" />
<H2>First Name:
<jsp:getProperty name="nameBean" property="firstName" /></H2>
<H2>Last Name:
<jsp:getProperty name="nameBean" property="lastName" /></H2>
</BODY></HTML>
```

JSP 2.0

```
. . .
<BODY>
<H1>Thanks for Registering</H1>
<H2>First Name:
${nameBean.firstName}</H2>
<H2>Last Name:
${nameBean.lastName}</H2>
</BODY></HTML>
```

ServletContext-Based Sharing: Bean

Goal: Display a prime number of a specified length. If the user fails to tell us the desired length, we want to use whatever prime number we most recently computed for *any user.*

- **Type of sharing** Data is shared among multiple clients, so application-based sharing is appropriate.

Java Bean

```
import java.math.BigInteger;
public class PrimeBean {
  private BigInteger prime;
  public PrimeBean(String lengthString) {
    int length = 150;
    try {
      length = Integer.parseInt(lengthString);
    } catch(NumberFormatException nfe) {}
    setPrime(Primes.nextPrime(Primes.random(length)));
  }
  public BigInteger getPrime() {
    return(prime);
  }…
}
```

ServletContext-Based Sharing: Servlet

```
public class PrimeServlet extends HttpServlet {
  public void doGet(HttpServletRequest request, HttpServletResponse
  response) throws ServletException, IOException {
    String length = request.getParameter("primeLength");
    ServletContext context = getServletContext();
    synchronized(this) {
      if((context.getAttribute("primeBean")==null)||(length != null)) {
        PrimeBean primeBean = new PrimeBean(length);
        context.setAttribute("primeBean", primeBean);
      }
    String address = "/WEB-INF/mvc-sharing/ShowPrime.jsp";
    RequestDispatcher dispatcher = request.getRequestDispatcher(address);
    dispatcher.forward(request, response);
   }
  }
}
```

ServletContext-Based Sharing: jsp

JSP 1.2

```
…
<BODY>
<H1>A Prime Number</H1>
<jsp:useBean id="primeBean" type="PrimeBean" scope="application" />
<jsp:getProperty name="primeBean" property="prime" />
</BODY></HTML>
```

JSP 2.0

```
<BODY>
<H1>A Prime Number</H1>
${primeBean.prime}
</BODY></HTML>
```

Forwarding from JSP Pages

```
<% String destination;
   if (Math.random() > 0.5) {
      destination = "/examples/page1.jsp";
   } else {
      destination = "/examples/page2.jsp";
  }
%>
<jsp:forward page="<%= destination %>" />
```

Legal, but bad idea. Better:

– Business and control logic belongs in servlets
– Keep JSP focused on presentation.

With the forward method of RequestDispatcher:

– Control is *permanently transferred to new page*
– Original page *cannot generate any output*

• **With the include method of RequestDispatcher:**

– Control is *temporarily transferred to new page*
– Original page *can generate output before and after the* included page
– Original servlet does not see the output of the included page (for this, see later topic on servlet/JSP filters)
– Useful for portals: JSP presents pieces, but pieces arranged in different orders for different users.

Including Pages Instead of Forwarding to Them

```
response.setContentType("text/html");
String firstTable, secondTable, thirdTable;
if (someCondition) {
   firstTable = "/WEB-INF/Sports-Scores.jsp";
   secondTable = "/WEB-INF/Stock-Prices.jsp";
   thirdTable = "/WEB-INF/Weather.jsp";
} else if (...) { ... }
RequestDispatcher dispatcher = request.getRequestDispatcher("/WEB-
  INF/Header.jsp");
dispatcher.include(request, response);
dispatcher = request.getRequestDispatcher(firstTable);
dispatcher.include(request, response);
dispatcher = request.getRequestDispatcher(secondTable);
dispatcher.include(request, response);
dispatcher = request.getRequestDispatcher(thirdTable);
dispatcher.include(request, response);
dispatcher = request.getRequestDispatcher("/WEB-INF/Footer.jsp");
dispatcher.include(request, response);
```

Part 3. Web Services

Introduction to Web Services in Java

Servizi Web

Web Service = A software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically Web Services Description Language -WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.

Web services provide a standard means of interoperating between different software applications, running on a variety of platforms and/or frameworks.

A Web service is an abstract notion that must be implemented by a concrete *agent*. Web services are frequently just Web APIs that can be accessed over a network, such as the Internet, and executed on a remote system hosting the requested services (agent). The agents may change but the service remains the same!

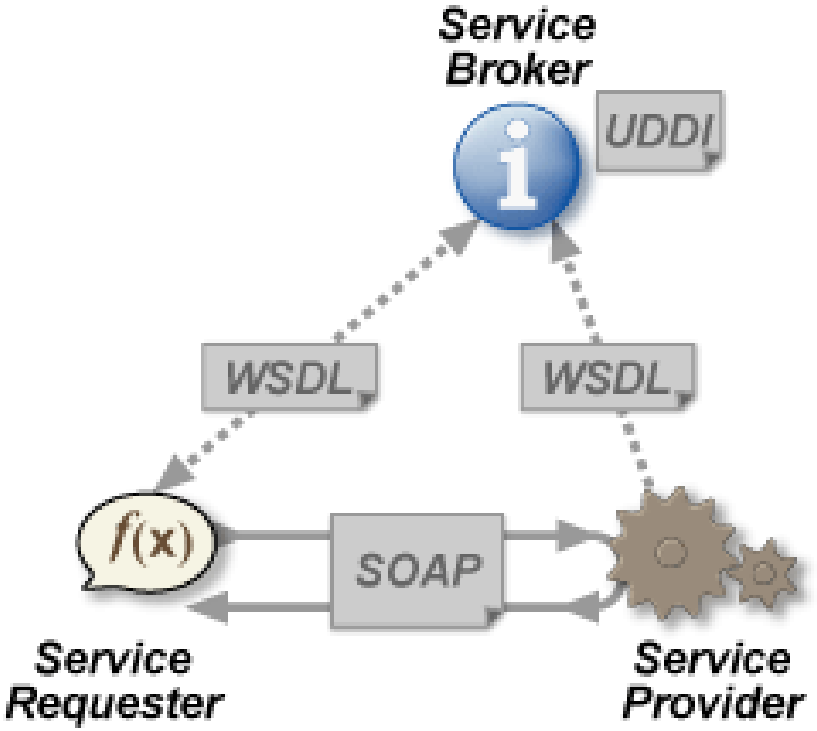
Web Service Architecture (WSA) provides a conceptual model and a context for understanding Web services and the relationships between the components of this model. The Web services architecture is an *interoperability* architecture: it identifies those global elements of the global Web services network that are required in order to ensure interoperability between Web services.

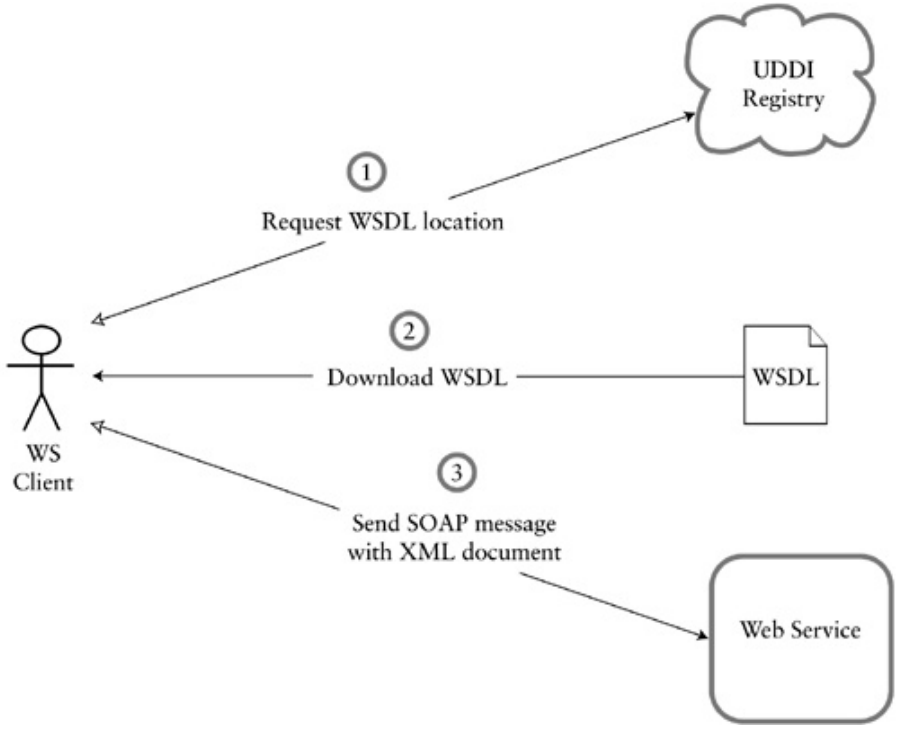
Web Service Styles

- **Web services** are a set of tools that can be used in a number of ways. The three most common styles of use are: RPC, SOA, and REST.
- **Remote procedure calls** (XML-RPC)
  RPC Web services present a distributed function (or method) call interface. Typically, the basic unit of RPC Web services is the WSDL operation. The first Web services tools were focused on RPC, and as a result this style is widely deployed and supported.
  Drawback: It is not loosely coupled, because it was often implemented by mapping services directly to language-specific functions or method calls. Many vendors pushed for RPC to be disallowed in the WS-I Basic Profile.
- **Service-oriented architecture (SOA)**
  Web services can also be used to implement a SOA where the basic unit of communication is a message, rather than an operation. This is often referred to as *"message-oriented" services*. Unlike RPC Web services, loose coupling is more likely, because the focus is on the "contract" that WSDL provides, rather than the underlying implementation details.
- **Representational state transfer (REST)**
  RESTful Web services attempt to emulate HTTP by constraining the interface to a set of well-known, standard operations (e.g., GET, PUT, DELETE). Here, the focus is on interacting with stateful resources, rather than messages or operations.

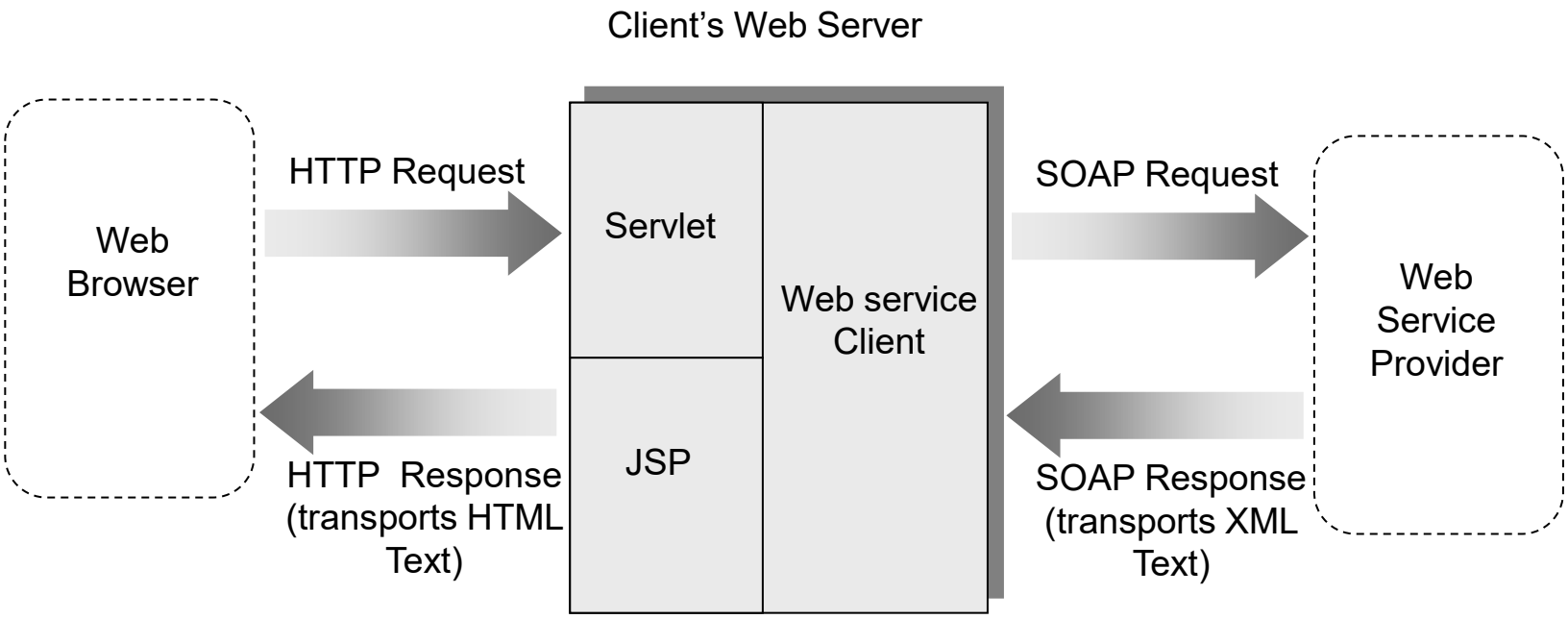
Standards for Web Services

XML (eXtensible Markup Language),
SOAP (Simple Object Access Protocol),
WSDL (Web Services Description Language)
UDDI (Universal Description, Discovery, and Integration)





Architecture for a WS-based Application

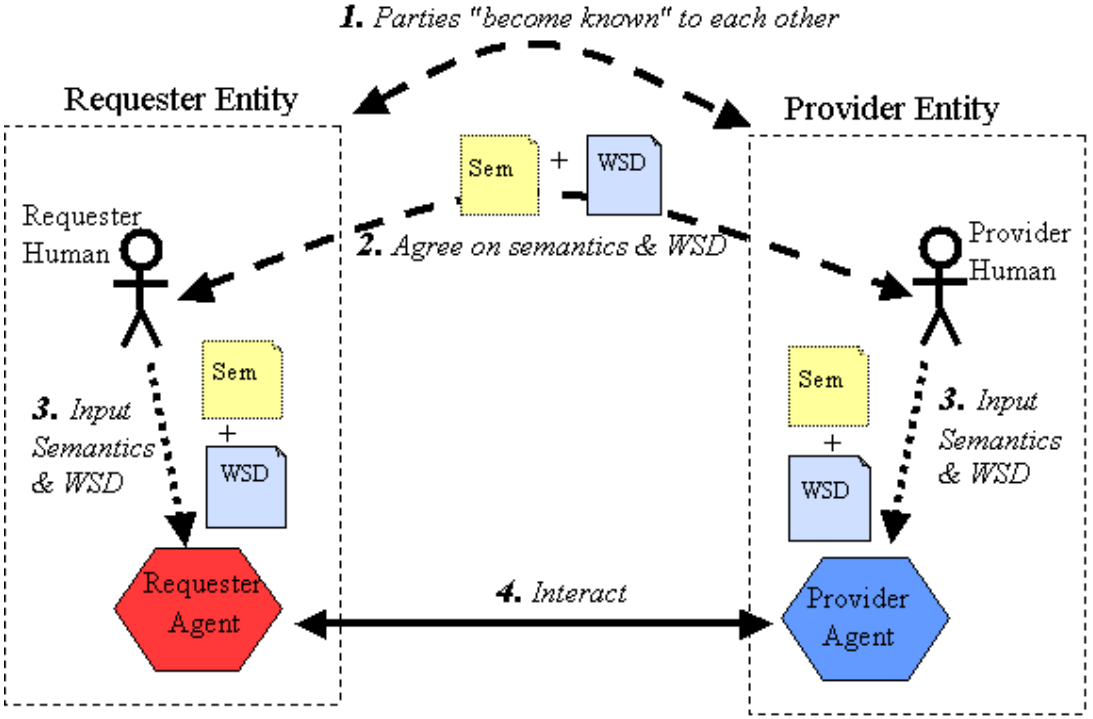


Requesters and Providers

A Web service provides some functionality on behalf of its owner -- a person or organization, such as a business or an individual. The *provider entity* is the person or organization that provides an appropriate agent to implement a particular service.

A *requester entity* is a person or organization that wishes to make use of a provider entity's Web service. It will use a *requester agent* to exchange messages with the provider entity's *provider agent*.

In order for this message exchange to be successful, the requester entity and the provider entity must first *agree* on both the semantics and the mechanics of the message exchange.
This means for both to have a congruent and non-conflictual view on the semantics of the service description.



Engaging a Web Service

- There are many ways that a requester entity might engage and use a Web service. In general, the following broad steps are required:

1. the requester and provider entities become known to each other (or at least one becomes know to the other);
2. the requester and provider entities somehow agree on the service description and semantics that will govern the interaction between the requester and provider agents;
3. the service description and semantics are realized by the requester and provider agents; and
4. the requester and provider agents exchange messages, thus performing some task on behalf of the requester and provider entities. (I.e., the exchange of messages with the provider agent represents the concrete manifestation of interacting with the provider entity's Web service.)

Some of these steps may be automated, others may be performed manually.

Web Service Description & Semantics

Web Service Description (WSD) documents the mechanics of the message exchange . It is a machine-processable specification of the Web service's interface, written in WSDL.

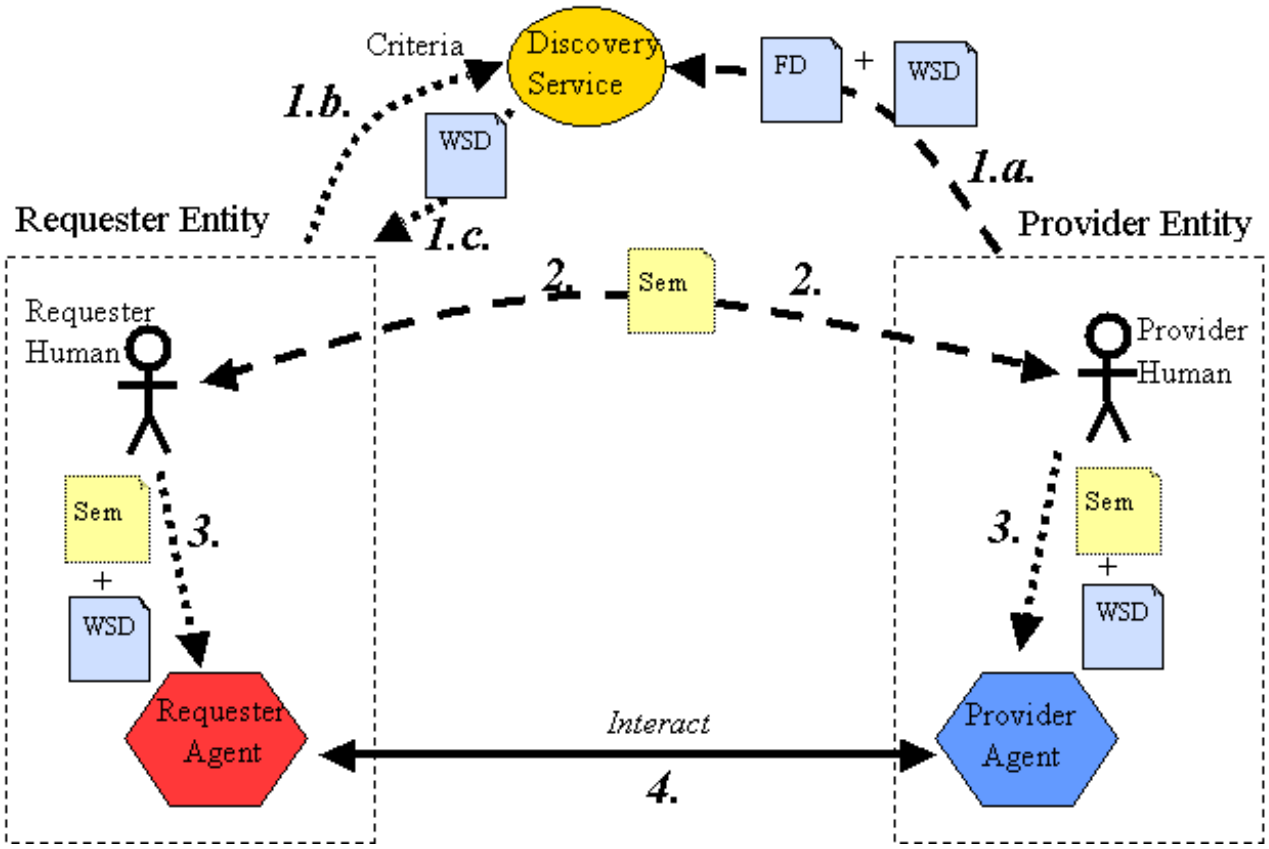
- It defines the message formats, datatypes, transport protocols, and transport serialization formats that should be used between the requester agent and the provider agent.
- It also specifies one or more network locations at which a provider agent can be invoked, and may provide some information about the message exchange pattern that is expected.

In essence, the service description represents an agreement governing the mechanics of interacting with that service.

The **semantics of a Web service** is the shared expectation about the behavior of the service, in particular in response to messages that are sent to it

This should be the "contract" between the requester entity and the provider entity regarding the purpose and consequences of the interaction. Although this contract represents the overall agreement between the requester entity and the provider entity on how and why their respective agents will interact, it is not necessarily written or explicitly negotiated. It may be explicit or implicit, oral or written, machine processable or human oriented, and it may be a legal agreement or an informal (non-legal) agreement.

Service Discovery Process



The functional description ("FD") is a machine-processable description of the functionality (or partial semantics) of the service that the provider entity is offering. It could be as simple as a few words of meta data or a URI, or it may be more complex, such as a TModel (in UDDI) or a collection of RDF, DAML-S or OWL-S statements.

Steps in Service Descovery

1. The requester and provider entities "become known to each other":
   1.a. The discovery service somehow obtains both the Web service description ("WSD") and an associated functional description ("FD") of the service. The WS architecture does not specify or care how the discovery service obtains the service description or functional description.
   1.b. The requester entity supplies criteria to the discovery service to select a Web service description based on its associated functional description, capabilities and potentially other characteristics. One might locate a service having certain desired functionality or semantics; however, it may be possible to specify "non-functional" criteria related to the provider agent, such as the name of the provider entity, performance or reliability criteria, or criteria related to the provider entity, such as the provider entity's vendor rating.
   1.c. The discovery service returns one or more Web service descriptions (or references to them) that meet the specified criteria.
2. The requester and provider entities agree on the semantics ("Sem") of the desired interaction.
3. The service description and semantics are input to, or embodied in, both the requester agent and the provider agent, as appropriate.
4. The requester agent and provider agent exchange SOAP messages on behalf of their owners.

Manual Versus Autonomous Discovery

Under *manual discovery*, a requester *human* uses a discovery service (typically at design time) to locate and select a service description that meets the desired functional and other criteria.

Under *autonomous discovery*, the requester *agent* performs this task, either at design time or run time.

The constraints and needs are significantly different, such as:

- *Interface requirements.* The requirements for something that is intended for human interaction are very different from the requirements for something that is intended for machine interaction.
- *Need for standardization.* There is far less need to standardize an interface or protocol that humans use than those that machines are intended to use.
- *Trust.* People do not necessarily trust machines to make decisions that may have significant consequences.

In the case of autonomous discovery, the need for machine-processable semantics is greatly increased.

One situation in which autonomous discovery is often needed is when the requester agent has been interacting with a particular provider agent, but for some reason needs to refresh its choice of provider agent, either because the previous provider agent is no longer available, or other reasons.

Discovery: Registry, Index or Peer-to-Peer?

Three viewpoints on how a discovery service should be conceived: as a *registry*, as an *index*, or as a *peer-to-peer* system.

1. A *registry* is an authoritative, centrally controlled store of information.
   - Publishing a service description requires an active step by the provider entity. In the case of a registry the registry owner decides *who* has authority to place information into, or update, the registry.
   - The registry owner decides *what* information is placed in the registry. Others cannot independently augment that information.

UDDI is an example of the registry approach, but it can be used as an index.

2. An *index* is a compilation or guide to information that exists elsewhere. It is not authoritative and does not centrally control the information.
   - Publishing is passive: the provider entity exposes the service and functional descriptions on the Web, and those who are interested collect them without the provider entity's specific knowledge.
   - Anyone can create their own index.
   - The information contained in an index could be out of date.
   - An index could include third-party information.
   - Different indexes could provide different kinds of information.
   - Free-market forces determine which index people will use to discover the information that they seek.

Google is often cited as an example of the index approach.

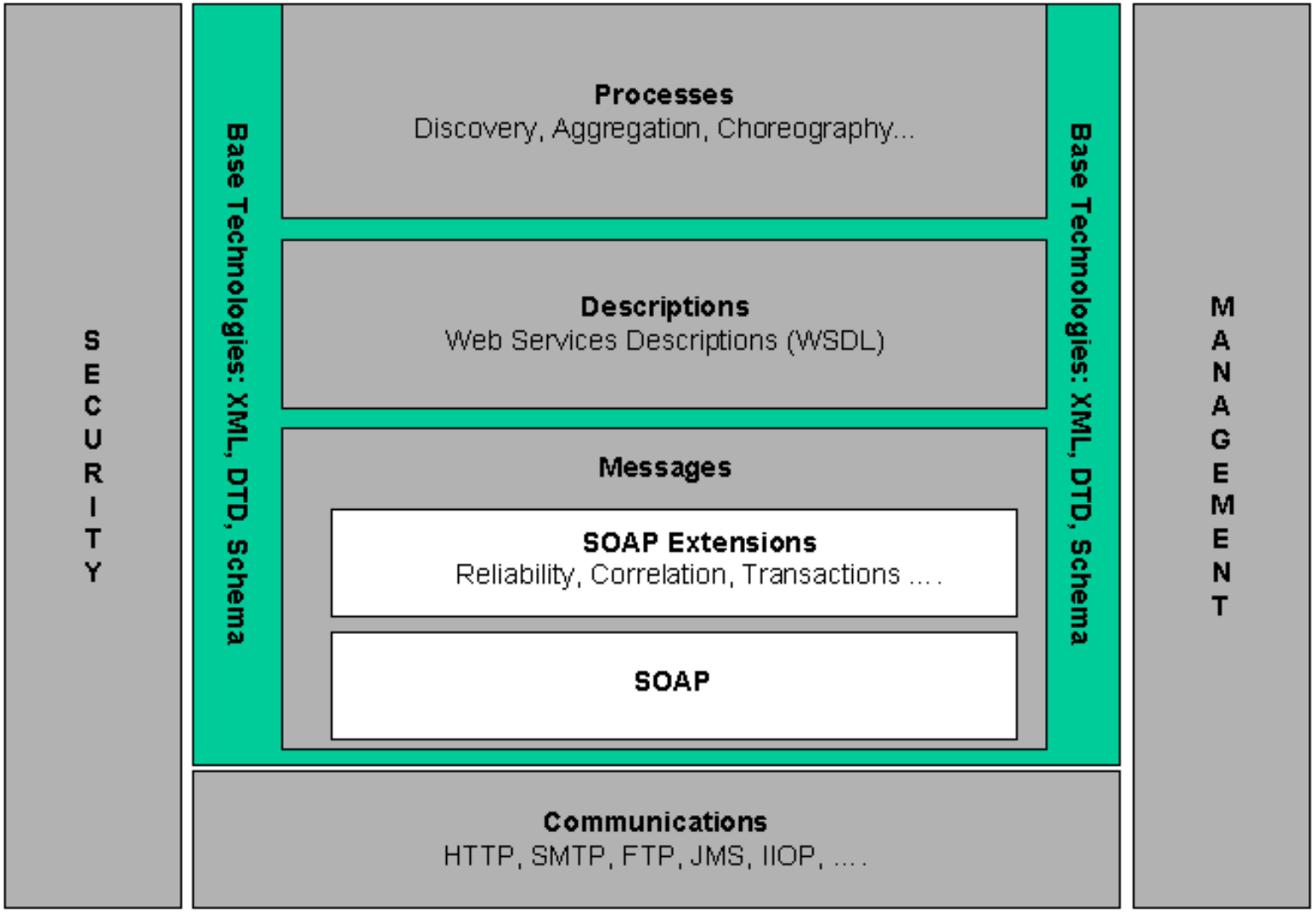
Discovery: Registry, Index or Peer-to-Peer?

3. Peer-to-Peer (P2P) Discovery

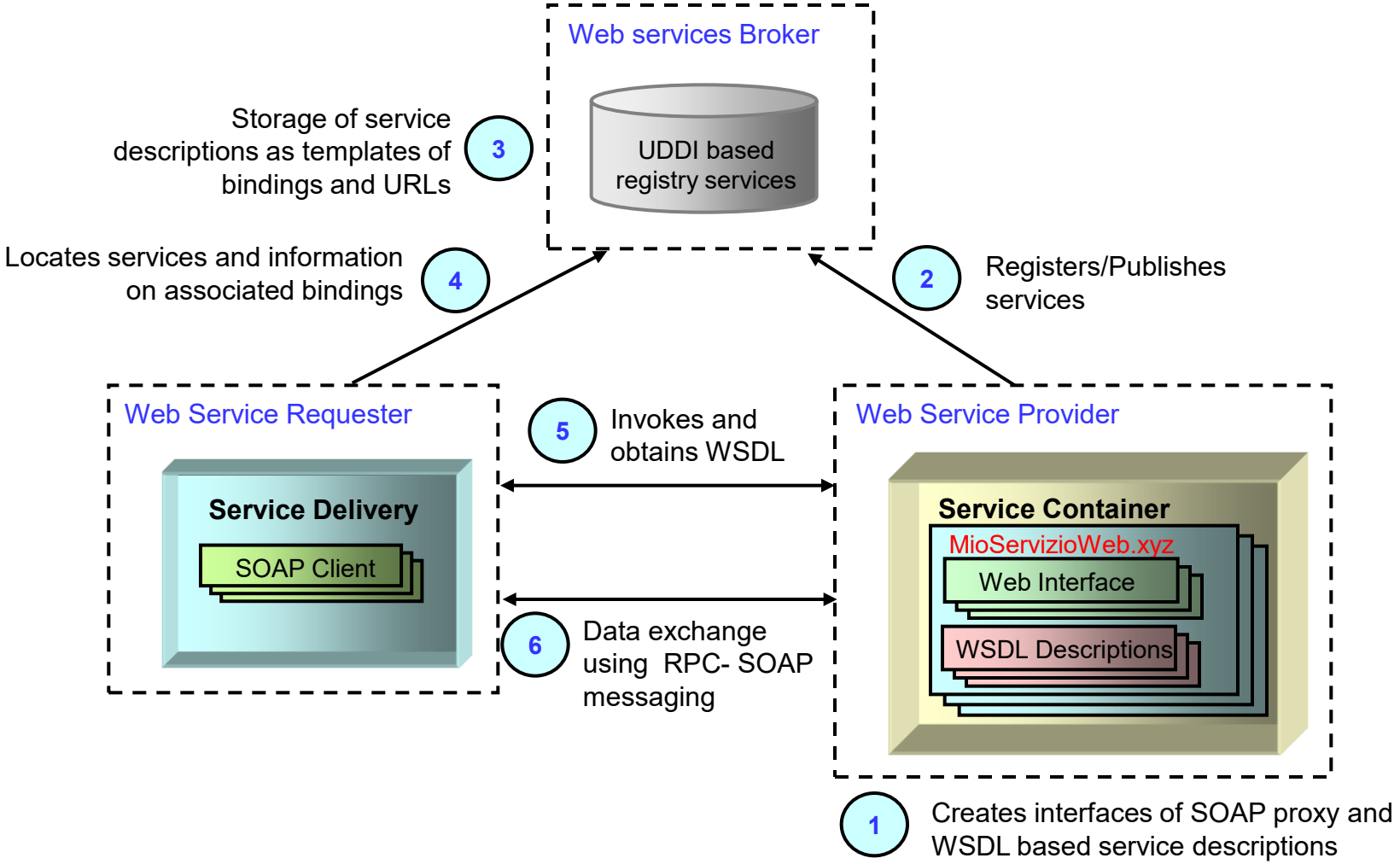
P2P computing provides an alternative that does not rely on centralized registries; rather it allows Web services to discover each other dynamically. Under this view, a Web service is a node in a network of peers, which may or may not be Web services. At discovery time, a requester agent queries its neighbors in search of a suitable Web service. If any one of them matches the request, then it replies. Otherwise each queries its own neighboring peers and the query propagates through the network until a particular hop count or other termination criterion is reached.

- Peer-to-peer architectures do not need a centralized registry, since any node will respond to the queries it receives.
- P2P architectures do not have a single point of failure, such as a centralized registry. Furthermore, each node may contain its own indexing of the existing Web services.
- Finally, nodes contact each other directly, so the information they they receive is known to be current.
- The reliability provided by the high connectivity of P2P systems comes with performance costs and lack of guarantees of predicting the path of propagation. There may be no guarantee that a request will spread across the entire network, therefore there is no guarantee to find the providers of a service.

Web Services Architecture Stack



Using Web Services



Standards for Web Services

XML

XML is a standard for inter-platform, extensible and plain-text based representation of data

XML

- **eXtensible Markup Language (XML)** is a meta-language, that is a language for defining other languages, defined by a specification.
- As a specification, XML defines the rules for creating XML markup languages.
- An **XML markup language** defines a set of **tags** that are used to organize and describe text. Tags are usually paired; together, a start tag, an end tag, and everything between them are called an **element**.
- A specific XML markup language (also known as an XML application[1]) describes which element names are used and how they are organized. The number of XML markup language is potentially infinite.
- Some XML markup languages are de facto standards within their industries: CML (chemistry), MathML (mathematics), DocBook (publishing), SVG (multimedia), XHTML (Web pages), etc.
- XML is used for organizing business documents and data. The XML files can be stored and transmited among aplications in net.
- XML is fundamental for web services technologies.
- XML documents are composed of Unicode text, so people as well as software can understand them: you can open an XML document and read it in any text editor. Because XML's syntactical rules are strict, however, you can also parse and manipulate it with a variety of software tools.

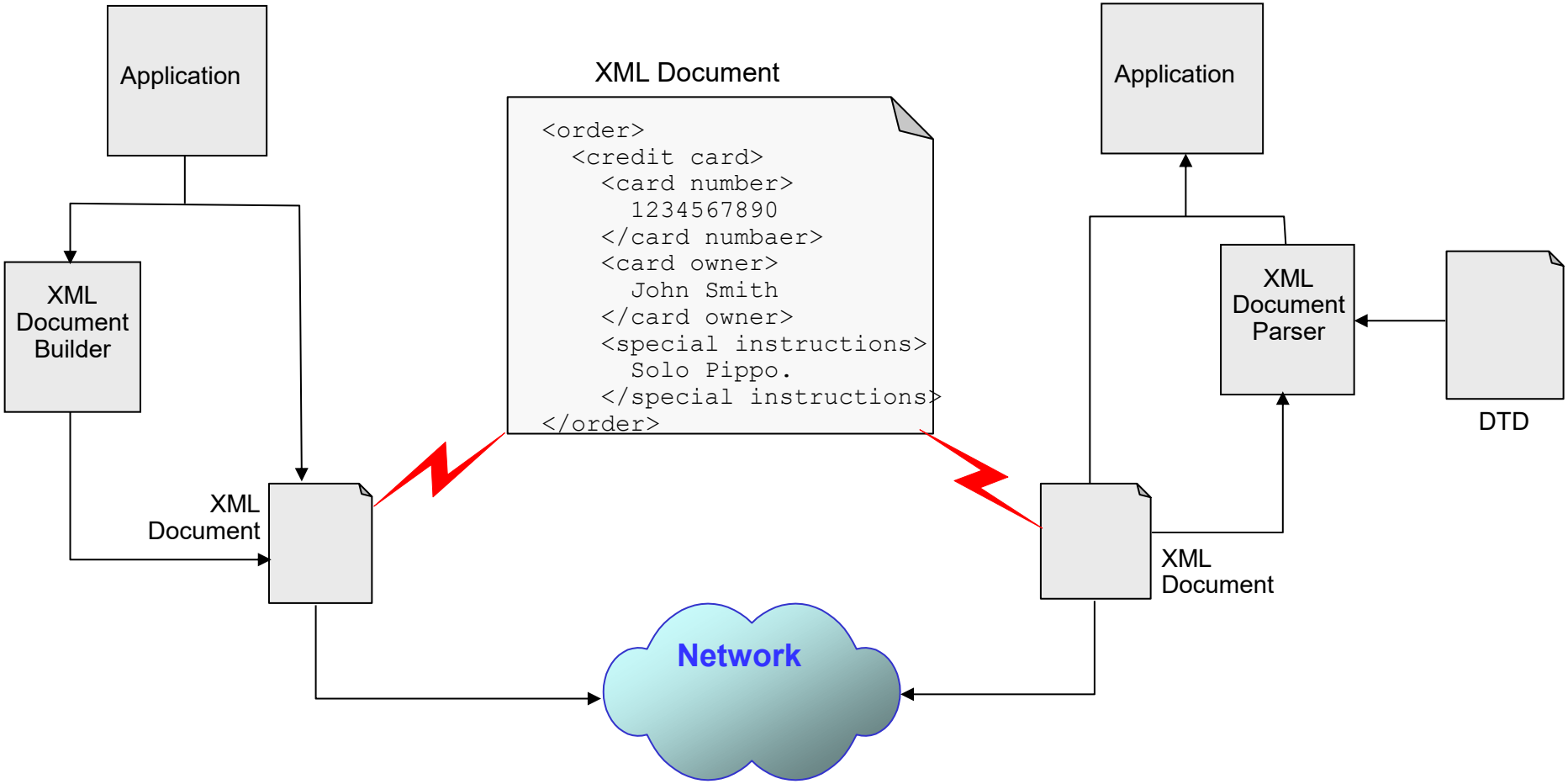
Example: Address Book Markup Language

```
<?xml version="1.0" encoding="UTF-8"?>
  <addresses>
     <address category="friend">
       <name>Bill Frankenfiller</name>
       <street>3243 West 1st Ave.</street>
       <city>Madison</city>
       <state>WI</state>
       <zip>53591</zip>
     </address>
     <address category="business">
       <name>Amazon.com</name>
       <street>1516 2nd Ave</street>
       <city>Seattle</city>
       <state>WA</state>
       <zip>90952</zip>
     </address>
</addresses>
```

XML is used for two different purposes:

- *document-oriented markup languages* (XHTML,DocBook) are focused on the format and presentation of literature. They describe how books, magazines, and business and scientific papers are organized, and how they look.
- *data-oriented applications markup languages* focus on how data is organized and typed; they define a schema for storing and exchanging data between software applications. Some XML markup languages are industry standards, like SOAP and XHTML.

How XML is used?



XML Document

```
<?xml version="1.0"?>
<purchaseOrder orderDate="1999-10-20">
  <shipTo country="US">
    <name>Alice Smith</name>
    <street>123 Maple Street</street>
    <city>Mill Valley</city>
    <state>CA</state>
    <zip>90952</zip>
  </shipTo>
  <billTo country="US">
    <name>Robert Smith</name>
    <street>8 Oak Avenue</street>
    <city>Old Town</city>
    <state>PA</state>
    <zip>95819</zip>
  </billTo>
  <comment>Hurry, my lawn is going wild!</comment>
  <items>
    <item partNum="872-AA">
       <productName>Lawnmower</productName>
       <quantity>1</quantity>
       <USPrice>148.95</USPrice>
       <comment>Confirm this is electric</comment> </item>
   <item partNum="926-AA">
     <productName>Baby Monitor</productName>
     <quantity>1</quantity>
     <USPrice>39.98</USPrice>
     <shipDate>1999-05-21</shipDate>
    </item>
  </items>
</purchaseOrder>
```

XML Document Structure

Elements

```
<?xml version="1.0" encoding="UTF-8" ?>
<address>                ←----- root element
  <name>Amazon.com</name>
  <street>1516 2nd Ave</street>
  <city>Seattle</city>
  <state>WA</state>
  <zip>90952</zip>
</address>
```

- Six elements: `address`, `name`, `street`, `city`, `state`, and `zip`. Except `address`, all contain text.

Attributes

An element may have one or more attributes. You use an attribute to supplement the data contained by an element, to provide information about it not captured by its contents.

```
<address category="business" >
```

Each attribute is a name-value pair. The value must be in single or double quotes. You can define any number of attributes for an element. Attributes cannot be nested like elements.

Empty-element tags may be used when the attributes contain all the data:

```
<phone countrycode="01" areacode="715" number="55529482" ext="341" />
```

Using attributes instead of nested elements is considered a matter of style.

Processing XML Documents

- A software application that reads and manipulates XML documents will use an XML parser. An XML parser can read an XML document and parse its contents according to the XML syntax.
- Parsers usually provide a programming API that allows developers to access elements, attributes, text, and other constructs in XML documents.
- Two standard kinds of XML parser APIs: *SAX* and *DOM*.
- SAX (Simple API for XML). SAX2, the latest version, is based on an event model. As the SAX2 parser reads an XML document, starting at the beginning, it fires off events every time it encounters a new element, attribute, piece of text, or other component. SAX2 parsers are generally very fast because they read an XML document sequentially and report on the markup as it's encountered.
- DOM (Document Object Model) is maintained by the W3C. DOM 2 presents the programmer with a generic, object-oriented model of an XML document. Elements, attributes, and text values are represented as objects organized into a hierarchical tree structure that reflects the hierarchy of the XML document being processed. DOM 2 allows an application to navigate the tree structure, modify elements and attributes, and generate new XML documents in memory. It's slow compared to SAX2, and consumes a lot more memory.
- JAXP (Java API for XML Processing), which is part of the J2EE platform, is not a parser. It is a set of factory classes and wrappers for DOM 2 and SAX2 parsers.

Declaring Namespaces in XML

With namespaces we indicate that the address element should conform to the Address Markup Language, while the rest of the elements conform to the Purchase Order Markup Language. The `xmlns` declarations define the default namespace for the element and all its descendants.

```
<?xml version="1.0" encoding="UTF-8" ?>
<purchaseOrder orderDate="2008-02-22"
  xmlns="http://www.MySite.com/jwsbook/PO">
  <accountName>Amazon.com</accountName>
  <accountNumber>923</accountNumber>
  <address xmlns="http://www.MySite.com/jwsbook/ADDR">
    <name>AMAZON.COM</name>
    <street>1850 Mercer Drive</street>
    <city>Lexington</city>
    <state>KY</state>
    <zip>40511</zip>
  </address>
  <book>
    <title>J2EE Web Services</title>
    <quantity>300</quantity>
    <wholesale-price>29.99</wholesale-price>
  </book>
  <total>8997.00</total>
</purchaseOrder>
```

Declaring and Using Namespaces Prefixes in XML

To simplify things, XML Namespaces defines a shorthand notation for associating elements and attributes with namespaces. You can assign an XML namespace to a prefix, then use that prefix to fully qualify each element name.

```
<?xml version="1.0" encoding="UTF-8" ?>
<po:purchaseOrder orderDate="2003-09-22"
  xmlns:po="http://www.MySite.com/jwsbook/PO"
  xmlns:addr="http://www.MySite.com/jwsbook/ADDR">
  <po:accountName>Amazon.com</po:accountName>
  <po:accountNumber>923</po:accountNumber>
  <addr:address>
     <addr:name>AMAZON.COM</addr:name>
     <addr:street>1850 Mercer Drive</addr:street>
     <addr:city>Lexington</addr:city>
     <addr:state>KY</addr:state>
     <addr:zip>40511</addr:zip>
  </addr:address>
  <po:book>
     <po:title>J2EE Web Services</po:title>
     <po:quantity>300</po:quantity>
     <po:wholesale-price>29.99</po:wholesale-price>
  </po:book>
  <po:total>8997.00</po:total>
</po:purchaseOrder>
```

Avoiding Element Name Collisions by Using XML Namespaces

`addr:address` is called a **Qname**. Two parts: *XML namespace* : *the local name*.

Namespaces help avoid name collisions, where two elements from different markups share a common local name. Two elements are named `address`, they belong to different namespaces (`soap` and `addr`) with different QNames, so there is no name conflict.

```
<?xml version="1.0" encoding="UTF-8" ?>
<definitions name="Address-Update"
  targetNamespace="http://www.MySite.org/jwsbook/Address-Update"
  xmlns:tns="http://www.MySite.org/jwsbook/Address-Update"
  xmlns:addr="http://www.MySite.com/jwsbook/ADDR"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns="http://schemas.xmlsoap.org/wsdl/">
  ...
 <!-- message elements describe the paramters and return values -->
 <message name="AddressMessage">
     <part name="address" element="addr:address" />
 </message>
  ...
 <!-- service tells us the Internet address of a Web service -->
 <service name="AddressUpdateService">
  <documentation>Update a customers mailing address</documentation>
  <port name="AddressUpdate_Port" binding="tns:AddressUpdate_Binding">
     <soap:address location="http://www.MySite.org/jwsbook/BookPrice" />
  </port>
</service>
```

XML Schema

- A XML schema describes an XML markup language. Specifically it defines which elements and attributes are used in a markup language, how they are ordered and nested, and what their data types are.
- XML schema is actually an XML document.
- The root element of a schema document is always the schema element. Nested within the schema element are element and type declarations.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:mh="http://www.MySite.com/jwsbook"
  targetNamespace="http://www.MySite.com/jwsbook">
    <element name="address" type="mh:USAddress" />
    <complexType name="USAddress">
      <sequence>
         <element name="name" type="string" />
         <element name="street" type="string" />
         <element name="city" type="string" />
         <element name="state" type="string" />
         <element name="zip" type="string" />
      </sequence>
    </complexType>
  ...
</schema>
```

- The schema declares a complex type named `USAddress` as a sequence of elements, and an element of that type named `address`.

The schema of an XML document

```
<xsd:schema
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
       Purchase order schema. Copyright 2000.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:element name="purchaseOrder"
   type="PurchaseOrderType"/>
  <xsd:element name="comment" type="xsd:string"/>
  <xsd:complexType name="PurchaseOrderType">
    <xsd:sequence>
      <xsd:element name="shipTo" type="USAddress"/>
      <xsd:element name="billTo" type="USAddress"/>
      <xsd:element ref="comment" minOccurs="0"/>
      <xsd:element name="items" type="Items"/>
     </xsd:sequence>
     <xsd:attribute name="orderDate" type="xsd:date"/>
   </xsd:complexType>
   <xsd:complexType name="USAddress">
     <xsd:sequence>
       <xsd:element name="name" type="xsd:string"/>
       <xsd:element name="street" type="xsd:string"/>
       <xsd:element name="city" type="xsd:string"/>
       <xsd:element name="state" type="xsd:string"/>
       <xsd:element name="zip" type="xsd:decimal"/>
     </xsd:sequence>
     <xsd:attribute name="country" type="xsd:NMTOKEN"
   fixed="US"/>
 </xsd:complexType>
<xsd:complexType name="Items">
  <xsd:sequence>
   <xsd:element name="item" minOccurs="0"
   maxOccurs="unbounded">
    <xsd:complexType>
     <xsd:sequence>
      <xsd:element name="productName"
         type="xsd:string"/>
      <xsd:element name="quantity">
       <xsd:simpleType>
        <xsd:restriction
            base="xsd:positiveInteger">
         <xsd:maxExclusive value="100"/>
        </xsd:restriction>
       </xsd:simpleType>
      </xsd:element>
      <xsd:element name="USPrice"
         type="xsd:decimal"/>
      <xsd:element ref="comment"
         minOccurs="0"/>
      <xsd:element name="shipDate"
         type="xsd:date" minOccurs="0"/>
     </xsd:sequence>
     <xsd:attribute name="partNum"
         type="SKU" use="required"/>
    </xsd:complexType>
   </xsd:element>
  </xsd:sequence>
 </xsd:complexType>
 <!-- Stock Keeping Unit, a code for
   identifying products -->
 <xsd:simpleType name="SKU">
  <xsd:restriction base="xsd:string">
   <xsd:pattern value="\d{3}-[A-Z]{2}"/>
  </xsd:restriction>
 </xsd:simpleType>
</xsd:schema>
```

XML Schema Built-in Simple Types

- A simple type resembles a Java primitive type in that both are atomic; they cannot be broken down into constituent parts: a simple element type does not contain other elements; it will contain only data.
- The XML schema specification defines many standard simple types, called built-in types. The built-in types are the standard building blocks of an XML schema document. They are members of the XML schema namespace, `http://www.w3.org/2001/XMLSchema`

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
   xmlns:mh="http://www.MySite.com/jwsbook"
   targetNamespace="http://www.MySite.com/jwsbook">
 ...
<complexType name="PurchaseOrder">
  <sequence>
    <element name="accountName" type="string"/>
    <element name="accountNumber" type="integer"/>
    <element name="total" type="float"/>
    <!-- More stuff follows -->
  </sequence>
</complexType>
```

```
package com.mysite.jwsbook;

public class PurchaseOrder {

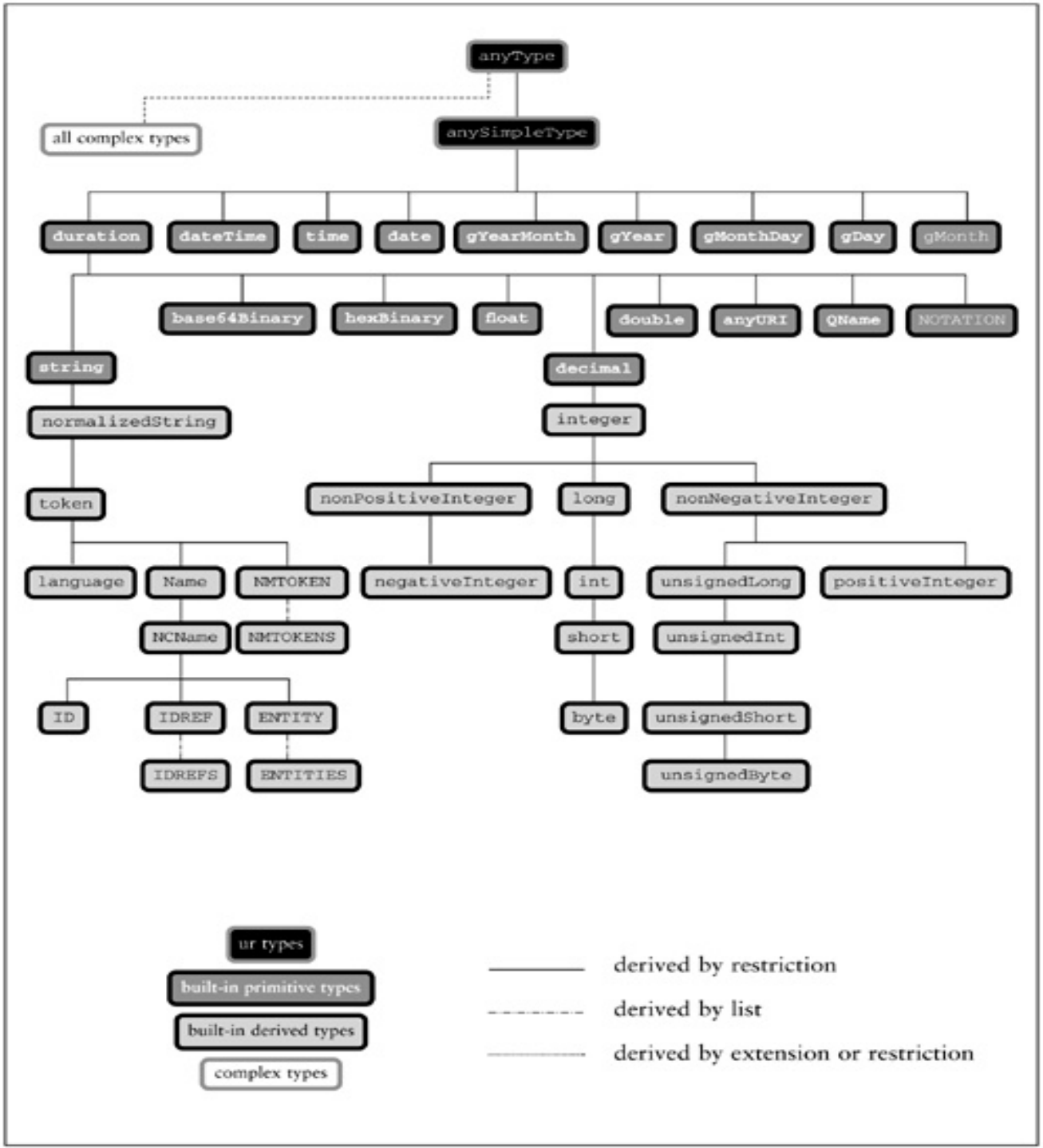
  String accountName;
  int accountNumber;
  float total;
  // more stuff follows

}
```

XML Schema Built-in Simple Types

Simple Type	Definition
string	A sequence of characters conforming to UCS
normalizedString	A string without carriage returns, line feeds, or tabs
token	A string without spaces, line feeds, or tabs
NMTOKEN	A token used in attributes
byte	A non-decimal number between –128 and 127
unsignedByte	A non-decimal number between 0 and 255
base64Binary	Base64-encoded binary data (RFC 2045)[a]
hexBinary	Hex-encoded binary data[b]
integer	A base-10-integer number of any size (…)[c]
positiveInteger	A base-10 integer greater then zero (1, 2, …)
negativeInteger	A base-10 integer less then zero (…, –2, –1)
int	A base-10 integer between –2.147.483.648 and 2.147.483.647 (–2 billion and 2 billion)
unsignedInt	A base-10 integer between 0 and 4.294.967.295
long	A base-10 integer between $-9{,}223*10^{18}$ and $9{,}223*10^{18}$
unsignedLong	A base-10 integer between 0 e $18{,}446*10^{18}$
short	A base-10 integer between –32.767 and 32.767
unsignedShort	A base-10 integer between 0 and 65.535
decimal	A decimal number of any precision and size
float	A decimal number conforming to the IEEE single-precision 32-bit floating-point type[d]
double	A decimal number conforming to the IEEE double-precision 64-bit floating-point type[d]
boolean	A boolean value of "true" or "false" can also use the values of "0" (false) or "1" (true);
time	A time in hours, minutes, seconds, and milliseconds formatted as hh:mm:ss.sss
date	A Gregorian date in centuries, years, months, and days[]
dateTime	A Gregorian date measured in centuries, years, months, and days
duration	A span of time measured in years, months, days, and seconds. The value must always start with the letter P.[f]

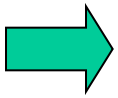
XML Schema Type Hierarchy



Complex Types

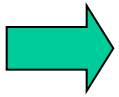
XML Schema: Complex Type

```
<complexType name="USAddress">
  <sequence>
     <element name="name" type="string"/>
     <element name="street" type="string"/>
     <element name="city" type="string"/>
     <element name="state" type="string"/>
     <element name="zip" type="string"/>
  </sequence>
</complexType>
```



Java Class Definition

```
public class USAddress {
  public String name;
  public String street;
  public String city;
  public String state;
  public String zip;
}
```



XML Document

```
<address>
  <name>Amazon.com</name>
  <street>1516 2nd Ave</street>
  <city>Seattle</city>
  <state>WA</state>
  <zip>90952</zip>
</address>
```

`<sequence>` = An element that lists one or more element definitions. The element definitions tells us which elements are nested in the type, the order in which they appear, and the kind of data each element contains.

Complex type declarations can build on other complex type definitions to create richer types that easily describe very complex XML structures.

In addition to sequences of elements, a complex type may also define its own attributes.

A complex Type in an XML Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:mh="http://www.MySite.com/jwsbook"
  targetNamespace="http://www.MySite.com/jwsbook" >

  <element name="purchaseOrder" type="mh:PurchaseOrder" />
  <element name="address" type="mh:USAddress" />
  <complexType name="PurchaseOrder">
    <sequence>
     <element name="accountName" type="string”/>
     <element name="accountNumber" type="unsignedShort"/>
     <element name="shipAddress" type="mh:USAddress”/>
     <element name="billAddress" type="mh:USAddress”/>
     <element name="book" type="mh:Book”/>
     <element name="total" type="float"/>
    </sequence>
    <attribute name="orderDate" type="date"/>
  </complexType>
<complexType name="USAddress">
    <sequence>
     <element name="name" type="string"/>
     <element name="street" type="string
     <element name="city" type="string”/
     <element name="state" type="string"
     <element name="zip" type="string”/>
    </sequence>
  </complexType> . . . .
</schema>
```

The XML document contains all the elements and the `orderDate` and would be verifiable against that type.

```
<?xml version="1.0" encoding="UTF-8"?>
<po:purchaseOrder
  orderDate="2003-09-22"
  xmlns:po=
"http://www.MySite.com/jwsbook">
<accountName>Amazon.com</accountName>
<accountNumber>923</accountNumber>   .
. .
```

Occurrence Constraints

The *multiplicity* of an element, the number of times it occurs in an instance document, is controlled by occurrence constraints, which are declared by the **maxOccurs** and **minOccurs** attributes.

```
<complexType name="USAddress">
  <sequence>
    <element name="name" type="string" />
    <element name="street" type="string" minOccurs="1" maxOccurs="2" />
    <element name="city" type="string" />
    <element name="state" type="string" />
    <element name="zip" type="string" />
  </sequence>
</complexType>
```

Constraints to constraints:

```
minOccurs>=0
maxOccurs>=minOccurs
```

Values of occurrence constraints:

predefined: `maxOccurs = 1 minOccurs = 1`

```
<element name="billAddress" type="mh:USAddress" minOccurs="0" />
<element name="book" type="mh:Book" maxOccurs="unbounded" />
```

That is: **`minOccurs="0"`** : `billAddress` is optional, `maxOccurs` is illimited, but at least 1 (because `minOccurs="1"` is predefined)

Occurrence Constraints

Attributes also have occurrence constraints, but they are different from those of elements. Instead of `maxOccurs` and `minOccurs`, attribute types declare the use occurrence constraint, which may be :

- `required`
- `optional`
- `prohibited`

The predefined value is `optional`. An attribute might be `prohibited` if you want to stop the use of a particular attribute, perhaps one that is inappropriate or no longer in use.

Exemple:

```
<complexType name="PurchaseOrder">
  <sequence>
    <element name="accountName" type="string" />
    <element name="accountNumber" type="unsignedShort" />
    <element name="shipAddress" type="mh:USAddress" />
    <element name="billAddress" type="mh:USAddress" minOccurs="0" />
    <element name="book" type="mh:Book" maxOccurs="unbounded" />
    <element name="total" type="float" />
  </sequence>
  <attribute name="orderDate" type="date" use="required" />
</complexType>
```

Declaring the Default Value of an Attribute

- The default attribute can be used only when the `use` attribute is `optional`.
- If the `use` is `prohibited`, the attribute's not allowed so there is no sense having a default value.

Example:

```
<complexType name="USAddress">
  <sequence>
    <element name="name" type="string" />
    <element name="street" type="string" />
    <element name="city" type="string" />
    <element name="state" type="string" />
    <element name="zip" type="string" />
  </sequence>
  <attribute name="category" type="string" default="business" />
</complexType>
```

- An attribute may also be declared `fixed`. A fixed value is assigned to the attribute no matter what value appears in the XML instance document.

all

Occasionally we may want to use the all element. Unlike `sequence`, which defines the exact order of child elements, the XML schema all element allows the elements in it to appear in any order. Each element in an all group may occur once or not at all; no other multiplicity is allowed. In other words, `minOccurs` is always "0" and `maxOccurs` is always "1".

Example:

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:mh="http://www.MySite.com/jwsbook"
        targetNamespace="http://www.MySite.com/jwsbook" > ...
   <complexType name="USAddress">
      <all>
         <element name="name" type="string" />
         <element name="street" type="string" />
         <element name="city" type="string" minOccurs="0"/>
         <element name="state" type="string" minOccurs="0"/>
         <element name="zip" type="string" />
      </all>
   </complexType> . . .
```

`street`, `name` and `zip` have to appear at least once, but `city` and `state` can miss. If they occur, this will be exactly once. The order is any:

```
<addr:address xmlns:addr="http://www.MySite.com/jwsbook" >
   <zip>90952</zip>
   <street>1516 2nd Ave</street>
   <name>Amazon.com</name>
</addr:address>  . . .
```

Declaring Global Elements in a Schema

A schema may also declare global elements, which XML instance documents can refer to directly. Global elements are declared as direct children of the schema element, rather than children of a complex type.

```
<?xml version="1.0" encoding="UTF-8"?>
  <schema xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:mh="http://www.MySite.com/jwsbook"
    targetNamespace="http://www.MySite.com/jwsbook">
    <element name="address" type="mh:USAddress"/>
    <element name="purchaseOrder" type="mh:PurchaseOrder"/>
    <complexType name="PurchaseOrder">
       <sequence>
       ...
       </sequence>
       <attribute name="orderDate" type="date"/>
     </complexType>
     <complexType name="USAddress">
       <sequence>
       ...
       </sequence>
     </complexType>
  ...
  </schema>
```

The schema allows to create XML documents in which the `purchaseOrder` or `address` element are in the root..

```
<?xml version="1.0" encoding="UTF-8"?>
<po:purchaseOrder orderDate="2003-09-22"
    xmlns:addr="http://www . . .
    <accountName>Amazon.com</accountName>
    <accountNumber>923</accountNumber>
    <shipAddress>
    ...
</po:purchaseOrder>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<addr:address
   xmlns:addr="http://www . . .
   <name>AMAZON.COM</name>
   <street>1850 Mercer Drive</street>
   <city>Lexington</city>
   <state>KY</state>
   <zip>40511</zip>
 </addr:address>
```

Qualifying Elements in an XML Document

Elements in an XML document may or may not require `QName` prefixes.

Rules for qualifying elements:

1. Global elements and attributes must always be qualified, which means that in an XML instance you must prefix them to form a `QName`. Exception: when a global element is a member of the default namespace; all unqualified elements are assumed to be part of the default namespace.
2. local elements may not need to be qualified. XML schema defines two attributes, `elementsFormDefault` and `attributesFormDefault`, that determine whether local elements in an XML instance need to be qualified with a prefix or not. When the `formElementDefault` attribute is set to `unqualified`, in any XML instance all the local elements in the `targetNamespace` must be qualified with a prefix. If, on the other hand, the value for `formAttributeDefault` is `unqualified`, only the global elements must be qualified.

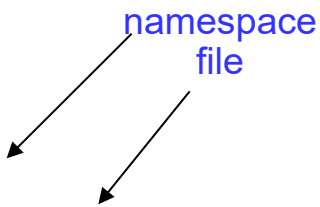
Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:mh="http://www.MySite.com/jwsbook"
    targetNamespace="http://www.MySite.com/jwsbook"
    elementFormDefault="qualified" >
  <element name="address" type="mh:USAddress" />
  <complexType name="USAddress">
    <sequence>
      ...
    </sequence>
  </complexType>
</schema>
```

schemaLocation

- To validate an XML document against one or more schemas, you need to specify which schemas to use. You do so by identifying the schemas' locations, using the `schemaLocation` attribute, which is an *XML schema-instance attribute*.

```
<?xml version="1.0" encoding="UTF-8"?>
<purchaseOrder orderDate="2003-09-22"
  xmlns="http://www.MySite.com/jwsbook"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.MySite.com/jwsbook
                      http://www.MySite.com/po.xsd">
  <accountName>Amazon.com</accountName>
  <accountNumber>923</accountNumber>
  . . .
```



`xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"` introduces `xsi` as the namespace of the XML schema instance, as it is defined by `XMLSchema`. `XMLSchema-instance` explicitly defines a few attributes belonging to this namespace, which can be used in XML documents, including the `xsi:schemaLocation` attribute

The `xsi:schemaLocation` attribute helps an XML processor locate the actual physical schema document used by the XML instance. Each schema is listed in an `xsi:schemaLocation` attribute as a namespace-location pair, which associates a namespace with a physical URL..

SOAP

SOAP (Simple Object Access Protocol) is the standard messaging protocol used by Web Services, and is the de facto standard for Web services in general. SOAP's primary application is Application-to-Application (A2A) communication. Specifically, it's used in Business-to-Business (B2B) and Enterprise Application Integration (EAI), which are two sides of the same coin: Both focus on integrating software applications and sharing data.

The SOAP standard

- SOAP is a XML-based protocol that defines a standard packaging format for messages. With this format the messages can be carried by any transport carrier able to carry a byte stream .
  - usually, the SOAP messages are exchanged between a client and services container in web containers;
  - the most common way to exchange SOAP messages is via HTTP. But you don't view SOAP messages with a browser as you do HTML. SOAP messages are exchanged between applications on a network and are not meant for human consumption. HTTP is just a convenient way of sending and receiving SOAP messages.

  Alternative carrying mechanisms could be used as FTP, SMTP or JMS.
  A SOAP message consists from:
  - an "envelope" that can contain any number of headers,
  - a body.
- XML elements called `<Envelope>`, `<Header>` and `<Body>` belong to the namespace used by the SOAP standard.
- The SOAP standard says nothing about the application-dependent information contained in the message except the fact that this content should be a valid XML message.

Example of SOAP message

- The SOAP messages are envelopes for documents and data exchanges in the net..
- Having a unique standard for data packaging and exchange web service applications become more interoperable, easily implementable and understandable.

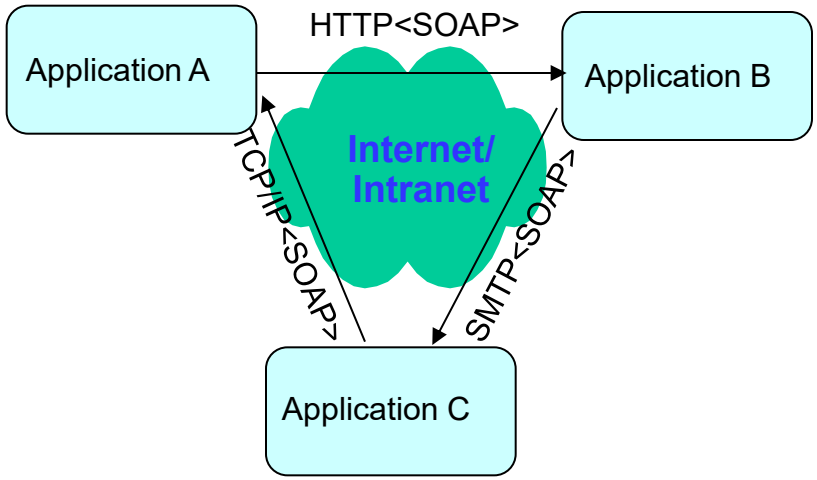
Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
  xmlns:soap=http://schemas.xmlsoap.org/soap/envelope/
  xmlns:addr="http://www.datamat.com/documenti/HTTR" >
    <soap:Body>
      <addr:address>
        <addr:name>Amazon.com</addr:name>
        <addr:street>1516 2nd Ave</addr:street>
        <addr:city>Seattle</addr:city>
        <addr:state>WA</addr:state>
        <addr:zip>90952</addr:zip>
       </addr:address>
     </soap:Body>
</soap:Envelope>
```

Another example of SOAP message

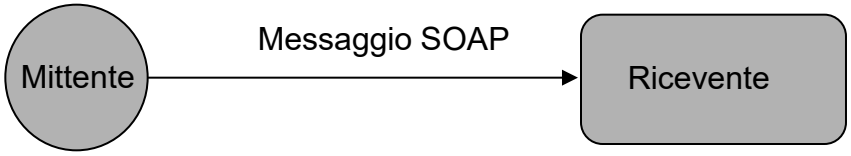
```
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:enc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:ns0= "urn:jwsnut.chapter6.headerbookservice/types/HeaderBookQuery"
  xmlns:ns1= "urn:jwsnut.chapter6.headerbookservice/wsdl/HeaderBookQuery"
  env:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
 <env:Header>
  <ns1:auth xsi:type="ns0:Authentication">
        <UserName xsi:type="xsd:string">JWSUserName</UserName>
        <Password xsi:type="xsd:string">JWSPassword</Password>
  </ns1:auth>
 </env:Header>
 <env:Body>
  <ns1:getBookAuthor>
        <String_1 xsi:type="xsd:string">Java in a Nutshell</String_1>
        <String_2 xsi:type="xsd:string" xsi:nil="1"/>
  </ns1:getBookAuthor>
 </env:Body>
</env:Envelope>
```

SOAP Messaging

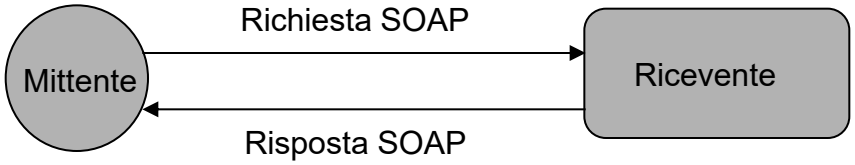


Web services can use One-Way messaging or Request/Response messaging. In the former, SOAP messages travel in only one direction, from a sender to a receiver. In the latter, a SOAP message travels from the sender to the receiver, which is expected to send a reply back to the sender.

One-Way



Request/Response.



The Basic Structure of SOAP

A SOAP message is analogous to an envelope used in traditional postal service. The next message is an example of a SOAP message that contains an arbitrary XML element, the purchaseOrder element.

è opzionale

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <po:purchaseOrder orderDate="2003-09-22"
     xmlns:po="http://www.MySide.com/PO">
      <po:accountName>Amazon.com</po:accountName>
      <po:accountNumber>923</po:accountNumber>
      <po:address>
        <po:name>AMAZON.COM</po:name>
        <po:street>1850 Mercer Drive</po:street>
        <po:city>Lexington</po:city>
        <po:state>KY</po:state>
        <po:zip>40511</po:zip>
      </po:address>
      <po:book>
        <po:title>J2EE Web Services</po:title>
        <po:quantity>300</po:quantity>
        <po:wholesale-price>24.99</po:wholesale-price>
      </po:book>
    </po:purchaseOrder>
  </soap:Body>
</soap:Envelope>
```

Envelope è l'elemento root in un messaggio SOAP.
Envelope **deve** contenere un elemento Body.
Prima di Body, l'Envelope **può** contenere un elemento Header.
L'elemento Body delimita, in un formato XML, il contenuto, i dati che si devono scambiare tra applicazioni.

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
xmlns:soap="http://schemas.xmlsoap.org/
soap/envelope/">
  <soap:Header>
    <!- Blocchi Header -->
  </soap:Header>
  <soap:Body>
    <!- Dati dell'applicazione -->
  </soap:Body>
</soap:Envelope>
```

The Structure of a SOAP Message

- Every XML document must have a root element, and in SOAP it's the `Envelope` element. `Envelope` may contain an optional `Header` element, and must contain a `Body` element. If you use a `Header` element, it must be the immediate child of the `Envelope` element, and precede the `Body` element. The `Body` element contains, in XML format, the actual application data being exchanged between applications. The `Body` element delimits the application-specific data.

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Header>
    <!-- Header blocks go here -->
  </soap:Header>
  <soap:Body>
    <!-- Application data goes here -->
  </soap:Body>
</soap:Envelope>
```

The Header Element

SOAP headers are a very powerful way of extending the SOAP protocol. As a construct for meta-data, a SOAP header is far more flexible and easier than similar mechanisms in other protocols.

The Header element contains information about the message, in the form of one or more distinct XML elements, each of which describes some aspect or quality of service associated with the message.

The Header element can contain XML elements that describe security credentials, transaction IDs, routing instructions, debugging information, payment tokens, or any other information about the message that is important in processing the data in the Body element.

For example, we may want to attach a unique identifier to every SOAP message, to be used for debugging and logging.

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:mi="http://www.MySite.com/jwsbook/message-id" >
   <soap:Header>
     <mi:message-id>11d1def534ea:b1c5fa:f3bfb4dcd7:-8000</mi:message-id>
   </soap:Header>
   <soap:Body>
     <!-- Application-specific data goes here -->
   </soap:Body>
</soap:Envelope>
```

The `message-id` element is called a **header block**, and is an arbitrary XML element identified by its own namespace. A header block can be of any size and can be very extensive.

Using XML Namespaces in a SOAP Message

A SOAP message may include several different XML elements in the `Header` and `Body` elements, and to avoid name collisions each of these elements should be identified by a unique namespace. Each header block in the Header element should have its own namespace.

Example: A SOAP message that contains the `purchaseOrder` element as well as `message-id` and XML digital-signature header blocks would include no fewer than six different namespaces.

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:sec="http://schemas.xmlsoap.org/soap/security/2000-12"
  xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
  xmlns:mi="http://www.MySide.com/message-id">
  <soap:Header>
    <mi:message-id>11d1def534ea:b1c5fa:f3bfb4dcd7:-8000</mi:message-id>
    <sec:Signature>
      <ds:Signature>
        ...
      </ds:Signature>
    </sec:Signature>
  </soap:Header>
  <soap:Body sec:id="Body">
    <po:purchaseOrder orderDate="2003-09-22"
     xmlns:po="http://www.MySite.com/PO"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
      ...
    </po:purchaseOrder>
  </soap:Body>
</soap:Envelope>
```

namespace of the standard SOAP 1.1 elements: Envelope, Header e Body. It is compulsory!

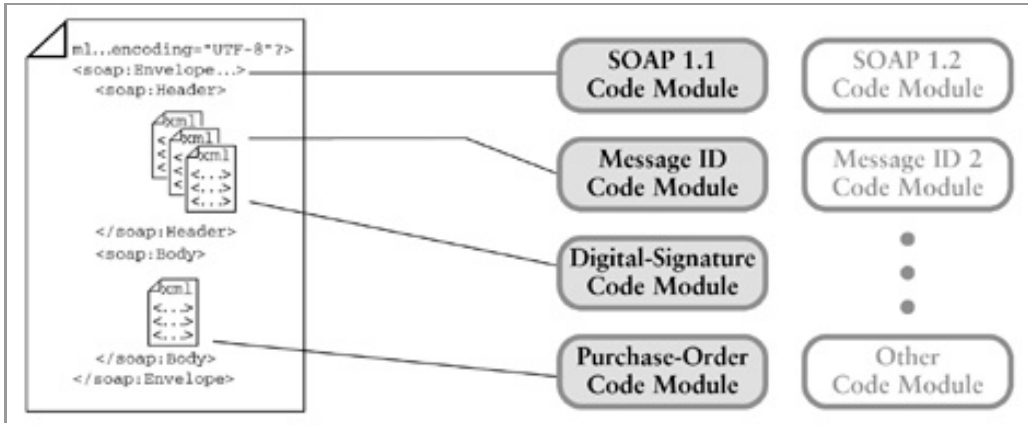
namespaces associated with XML elements in the header blocks

Namespace declaration for the Purchase Order markup language

1. All of the local elements of a SOAP message must be namespace-qualified because the XML schema for SOAP 1.1 specifies the `elementFormDefault` attribute as "`qualified`".
2. In addition, the Basic Profile 1.0 requires that all the application-specific elements contained by the `Body` element must be qualified

Namespaces and Code Modularity

- Namespaces enable a SOAP receiver to handle different versions of a SOAP message, without impairing backward compatibility or requiring different Web service endpoints for each version of a particular SOAP message.
- A SOAP message may contain many different namespaces, which makes SOAP messaging very modular. This modularity enables:
- different parts of a SOAP message to be processed independently of other parts and to evolve separately (the version of the SOAP Envelope or header blocks may change over time, while the structure of the application-specific contents in the Body element remains the same);
- the code that processes the SOAP messages to be modular as well (the code that processes the element Envelope is independent of the code that processes the header blocks, which is independent of the code that processes application-specific data in the SOAP Body element. Modularity enables you to use different code libraries to process different parts of a SOAP message.



Declaring Namespaces Locally

- We could declare namespaces in the Header element or the header blocks. Namespaces are always locally scoped and can be declared at any level as long as the elements in question are within that scope (the element the namespace is declared in, and its subelements)

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/" >
  <soap:Header>
    <mi:message-id xmlns:mi="http://www.MySite.com/jwsbook/message-id" >
      11d1def534ea:b1c5fa:f3bfb4dcd7:-8000
    </mi:message-id>
    <sec:Signature
      xmlns:sec=http://schemas.xmlsoap.org/soap/security/2000-12
      xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
    ...
    </sec:Signature>
  </soap:Header>
<soap:Body>
   . . .
</soap:Body>
</soap:Envelope>
```

W3C Namespace Prefixes

Prefix	Namespace
soap	"http://schemas.xmlsoap.org/soap/envelope/"
xsi	"http://www.w3.org/2001/XMLSchema-instance"
xsd	"http://www.w3.org/2001/XMLSchema"
soapenc	"http://schemas.xmlsoap.org/soap/encoding/"
wsdl	"http://schemas.xmlsoap.org/wsdl/"
soapbind	"http://schemas.xmlsoap.org/wsdl/soap/"
xsi	"http://ws-i.org/schemas/conformanceClaim/"

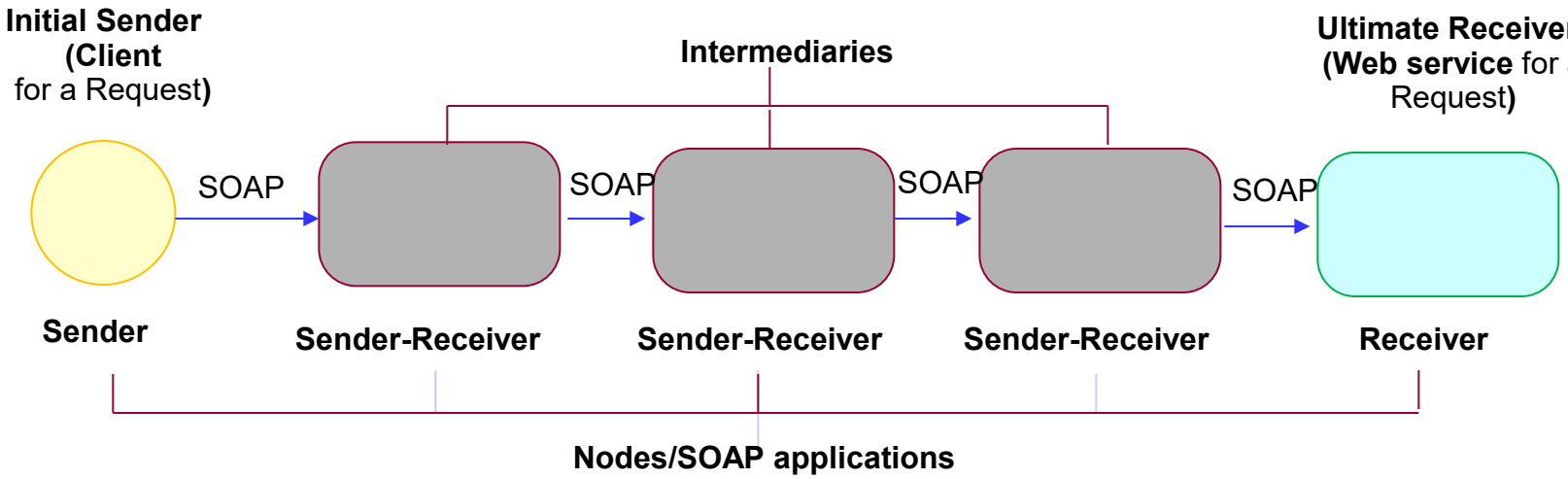
SOAP Message Path

In order to describe all the parties that participate in SOAP messaging, new terminology was invented to avoid restrictive and preconceived notions associated with traditional "client" and "server" terms.

The SOAP specification defines rules by which header blocks must be processed in the message path.

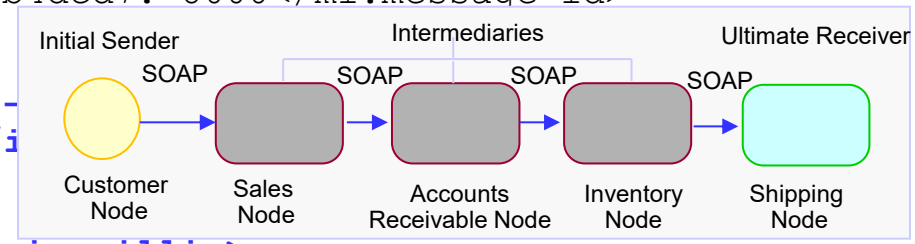
The message path is simply the route that a SOAP message takes from the initial sender to the ultimate receiver. It includes processing by any intermediaries. Intermediaries in a SOAP message path must not modify the application-specific contents of the SOAP Body element, but they may, and often do, manipulate the SOAP header blocks.

The SOAP rules specify which nodes must process particular header blocks and what should be done with header blocks after they've been processed.



Example of a Message Path

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:mi="http://www.MySite.com/message-id"
 xmlns:proc="http://www.MySite.com/processed-by">
  <soap:Header>
    <mi:message-id>11d1def534ea:b1c5fa:f3bfb4dcd7:-8000</mi:message-id>
    <proc:processed-by>
      <node>
        <time-in-millis>1013694680000</time-
        <identity>http://www.customer.com</i
      </node>
      <node>
        <time-in-millis>1013694680010</time-in-millis>
        <identity>http://www.MySite.com/sales</identity>
      </node>
      <node>
        <time-in-millis>1013694680020</time-in-millis
        <identity>http://www.MySite.com/AR</identity>
      </node>
      <node>
        <time-in-millis>1013694680030</time-in-millis
        <identity>http://www.MySite.com/inventory</identity>
      </node>
      <node>
        <time-in-millis>1013694680040</time-in-millis>
        <identity>http://www.MySite.com/shipping</identity>
      </node>
    </proc:processed-by>
  </soap:Header>
  <soap:Body>
     <!-- Application-specific data goes here -->
  </soap:Body>
</soap:Envelope>
```



In the example, each SOAP intermediary is required to add a node element to the processed-by header block, identifying itself and the time it processed the message.

The `actor` attribute

- When processing a header block, each node reads, acts on, and removes the header block from the SOAP message before sending it along to the next receiver. Any node in a message path may also add a new header block to a SOAP message .
- You use an `actor` (SOAP 1.1) or `role` (SOAP 1.2) attribute to identify a function to be performed by a particular node. The attribute is part of the `http://schemas.xmlsoap.org/soap/envelope/` namespace

1. The `actor` attribute uses a URI (Uniform Resource Identifier) to identify the role that a node must perform in order to process that header block. When a node receives a SOAP message, it examines each of the header blocks to determine which ones are targeted to roles supported by that node.
2. A node may have many modules that operate on a message, and therefore many roles, so it may identify itself with several different roles.
3. The `actor` attribute is used in combination with the XML namespaces to determine which code module will process a particular header block.
4. Conceptually, the receiving node: a) will first determine whether it plays the role designated by the actor attribute, and b) then choose the correct code module to process the header block, based on the XML namespace of the header block.

Therefore, the receiving node must recognize the role designated by the actor attribute assigned to a header block, as well as the XML namespace associated with the header block.

Using the `actor` attribute

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:mi="http://www.MySite.com/message-id"
 xmlns:proc="http://www.MySite.com/processed-by">
  <soap:Header>
    <mi:message-id  soap:actor="http://www.MySite.com/logger" >
      11d1def534ea:b1c5fa:f3bfb4dcd7:-8000
    </mi:message-id>
    <proc:processed-by>
      <node>
        <time-in-millis>1013694680000</time-in-millis>
        <identity>http://www.customer.com</identity>
      </node>
    </proc:processed-by>
  </soap:Header>
  <soap:Body>
   <!-- Application-specific data goes here -->
  </soap:Body>
</soap:Envelope>
```

The `actor` attribute identifies the `logger` role with the URL `http://www.MySite.com/logger`. A node that's intended to perform the `logger` role will look for header blocks where that URL is the value of the `actor` attribute.

The `message-id` header block in the purchase-order SOAP message is assigned the `actor` attribute value `http://www.MySite.com/logger`.

Only those nodes in the message path that identify themselves with the `http://www.MySite.com/logger` will process the `message-id` header block; all other nodes will ignore it.

Two standard `actor` Roles

- SOAP identifies two standard roles for the `actor` attribute: `next` and `ultimate receiver`.
- The `next` role indicates that the next node in the message path must process the header. The next role has a designated URI, which must be used as the value of the actor attribute: "`http://schemas.xmlsoap.org/soap/actor/next`".
- The `ultimate receiver` role indicates that only the ultimate receiver of the message should process the header block. The protocol doesn't specify an explicit URI for this purpose; it's the absence of an actor attribute in the header block that signals that the role is ultimate receiver.

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:mi="http://www.MySite.com/message-id"
   xmlns:proc="http://www.MySite.com/processed-by">
  <soap:Header>
    <mi:message-id  soap:actor="http://www.MySite.com/logger" >
        11d1def534ea:b1c5fa:f3bfb4dcd7:-8000
    </mi:message-id>
    <proc:processed-by soap:actor=" http://schemas.xmlsoap.org/soap/actor/next">
       <node>
           <time-in-millis>1013694680000</time-in-millis>
           <identity>http://www.customer.com</identity>
       </node>
    </proc:processed-by>
  </soap:Header>
  <soap:Body>
 <!-- Application-specific data goes here -->
  </soap:Body>
</soap:Envelope>
```

In this scenario, the intermediary node fulfills two roles: it is both a `logger` and the `next` receiver.

The SOAP Message After the Header Blocks Are Processed

When a node processes a header block, it must remove it from the SOAP message. The node may also add new header blocks to the SOAP message. SOAP nodes frequently feign removal of a header block by simply modifying it, which is logically the same as removing it, modifying it, and then adding it back to the SOAP message – a trick that allows a node to adhere to the SOAP specifications while propagating header blocks without losing any data.

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
   xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:mi=http://www.MySite.com/message-id
   xmlns:proc="http://www.MySite.com/processed-by">
  <soap:Header>
    <proc:processed-by soap:actor= "http://schemas.xmlsoap.org/soap/actor/next">
       <node>
           <time-in-millis>1013694680000</time-in-millis>
           <identity>http://www.customer.com</identity>
       </node>
       <node>
          <time-in-millis>1013694680010</time-in-millis>
          <identity>http://www.MySite.com/sales</identity>
       </node>
    </proc:processed-by>
  </soap:Header>
  <soap:Body>
     <!-- Application-specific data goes here -->
  </soap:Body>
</soap:Envelope>
```

The SOAP Body

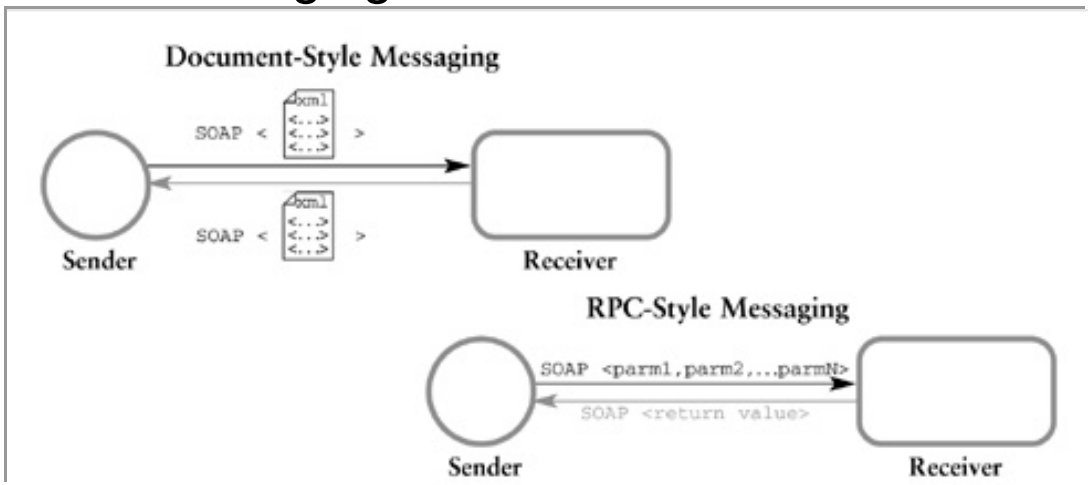
- All SOAP messages must contain exactly one `Body` element.
- `Body` contains either the application-specific data or a fault message.
- Application-specific data is the information that we want to exchange with a Web service. It can be arbitrary XML data or parameters to a procedure call. Either way, the `Body` element contains the application data being exchanged.
- A fault message is used only when an error occurs. The receiving node that discovers a problem, such as a processing error or a message that's improperly structured, sends it back to the sender just before it in the message path.
- Intermediary nodes in the message path may view the `Body` element, but they should not alter its contents in any way. This is very different from header blocks, which may be processed by any number of intermediaries along the message path. This is a critical point: Only the ultimate receiver should alter the contents of the `Body` element.
- SOAP supports mainly two modes of messaging: *RPC/ Literal* and *Document/Literal*.
- A messaging mode is defined by:
  - its messaging style (RPC or Document) and
  - its encoding style.
- Two common types of encoding used in SOAP messaging:
  - SOAP encoding (not supported by WS-I-conformant Web services) and
  - Literal encoding (i.e. the document can be validated against its schema).

Document/Literal and RPC/Literal vs. One-Way and Request/Response Messaging

1. "Document" or "RPC" is about the SOAP message payload:
   - an XML document fragment (for "Document") or
   - A representation of the value of the return parameter of a remote procedure call (for "RPC").
2. One-Way or Request/Response is about the message flow:
   - One-Way is a uni-directional flow,
   - Request/Response is a bi-directional flow.

Combinations

1. Document/Literal can be used with either One-Way or Request/Response.
2. RPC/Literal can be used with both MEP modes, but the most used is the Request/Response messaging.



Document/Literal

In the Document/Literal mode of messaging, a SOAP Body element contains an XML document fragment, a well-formed XML element that contains arbitrary application data (text and other elements) that belongs to an XML schema and namespace separate from the SOAP message's.

A set of XML elements that describes a purchase order, embedded within a SOAP message, is considered an XML document fragment.

The purchase-order SOAP message is a Document/Literal message:

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:mi="http://www.MySite.com/message-id"
 xmlns:proc="http://www.MySite.com/processed-by">
  <soap:Header>
    <!-- Header blocks go here -->
  </soap:Header>
  <soap:Body>
    <po:purchaseOrder orderDate="2003-09-22"
     xmlns:po="http://www.MySite.com/PO">
      <po:accountName>Amazon.com</po:accountName>
      <po:accountNumber>923</po:accountNumber>
      ...
      <po:book>
        <po:title>J2EE Web Services</po:title>
        <po:quantity>300</po:quantity>
        <po:wholesale-price>24.99</po:wholesale-price>
      </po:book>
    </po:purchaseOrder>
  </soap:Body>
</soap:Envelope>
```

RPC/Literal

- The RPC/Literal mode of messaging enables SOAP messages to model calls to procedures or method calls with parameters and return values. In RPC/Literal messaging, the contents of the Body are formatted as a `struct`.
- An RPC request message contains the method name and the input parameters of the call.
- An RPC response message contains the return value and any output parameters (or a fault). In many cases, RPC/Literal messaging is used to expose traditional components as Web services. A traditional component might be a servlet, stateless session bean, Java RMI object, CORBA object, or DCOM component. These components do not explicitly exchange XML data; rather, they have methods with parameters and return values.

Exemple of RPC/Literal Messaging

A remote interface of the `BookQuote` web service endpoint:

```
package com.jwsbook.soap;
import java.rmi.RemoteException;
public interface BookQuote extends java.rmi.Remote {
  // Get the wholesale price of a book public
  float getBookPrice(String ISBN) throws RemoteException,InvalidISBNException;
}
```

SOAP RPC/Literal Request Message

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:mh="http://www.MySite.com/BookQuote">
  <soap:Body>
    <mh:getBookPrice>
      <isbn>0321146182</isbn>
    </mh:getBookPrice>
  </soap:Body>
</soap:Envelope>
```

SOAP RPC/Literal Response Message

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:mh="http://www.MySite.com/BookQuote" >
  <soap:Body>
     <mh:getBookPriceResponse>
        <result>24.99</result>
     </mh:getBookPriceResponse>
  </soap:Body>
</soap:Envelope>
```

SOAP over HTTP

Sending a SOAP message as the payload of an HTTP POST message is very simple.

The BookQuote SOAP message embedded in an HTTP POST message:

```
POST /BookQuote HTTP/1.1
Host: www.MySite.com
Content-Type: text/xml; charset="utf-8"
Content-Length: 295
SOAPAction=""

<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:mh="http://www.MySite.com/BookQuote">
  <soap:Body>
    <mh:getBookPrice>
    <isbn>0321146182</isbn>
    </mh:getBookPrice>
  </soap:Body>
</soap:Envelope>
```

SOAP 1.2 will replace the SOAPAction header with the protocol-independent action media type (a parameter to the "application/soap+xml" MIME type)

The HTTP POST message must contain a `SOAPAction` header field, but the value of this header field is not specified. The `SOAPAction` header field can improve throughput by providing routing information outside the SOAP payload. A node can then do some of the routing work.

The reply to the SOAP message is placed in an HTTP reply message that is similar in structure to the request message, but contains no `SOAPAction` header.

WSDL

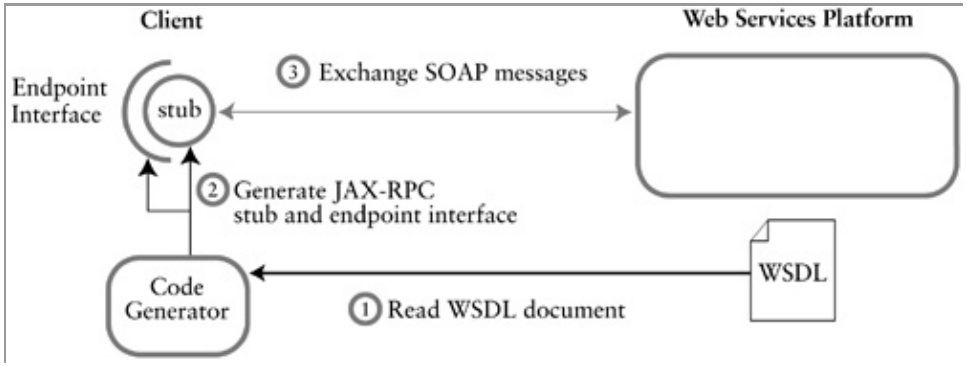
- WSDL (routinely pronounced "whiz-dul") is used to specify the exact message format, Internet protocol, and address that a client must use to communicate with a particular Web service. In other words, a WSDL document tells us how to use a Web service.

WSDL

- WSDL (Web Services Description Language) = a standard for specification of data structures exchanged between two systems that use the SOAP protocol.
- When a new web service is created, a WSDL document that describes the data types to be exchanged is also created.
- Web Service Description Language (WSDL) describes a web servizio by specifying:
  - which messages it accepts
  - with which messages it responds.
- The messages are defined :
  - first, in abstract terms,
  - then binding the abstract terms to transport protocols.
- In java environments a service WSDL definition is used to generate with a tool (for instance the J2EE *wscompile* tool) a Java interface that offers the same operations with the service .
- The Java interface:
  - is easier to be understood than the corresponding WSDL,
  - is used to write the own client application as well as for implement the service itself.

A WSDL Code Generator

- WSDL is well suited for code generators, which can read a WSDL document and generate a programmatic interface for accessing a Web service.
- A JAX-RPC provider (a vendor implementation of the JAX-RPC API) uses WSDL 1.1 to generate Java RMI interfaces and network stubs, which can be used to exchange SOAP messages with a Web service.
- The JAX-RPC toolkit uses a WSDL document to generate a Java RMI interface (an endpoint interface) and a networking stub that implements that interface.



- Many Web service clients use SOAP APIs instead of generated call interfaces and stubs. These APIs usually model the structure of the SOAP message using objects like `Envelope`, `Header`, `Body`, and `Fault`. Examples of SOAP APIs include SAAJ (SOAP with Attachments API for Java), Perl::Lite, Apache SOAP, and others.
- When you use a SOAP API, you can use a Web service's WSDL as a guide for exchanging SOAP messages with that Web service.

WSDL Glossary

1. WSDL describes services beginning from messages exchanged between the service provider and consumer. The messages are abstractly described.
2. The messages are associated to a network protocol and to a concrete message format.
3. Operation is an data exchange between the service provider and consumer.
4. An interface (portType) is a collection of operations. It is associated by bindings to a concrete protocol (SOAP, HTTP, MIME), a concrete message format (RPC or Document) and encoding style (Literal or SOAP Encoding).
5. A binding (thus its interface) is accessible via one or more endpoints. Each endpoint has its URI.
6. A service is a collection of endpoints, each endpoint is bound to the same interface.

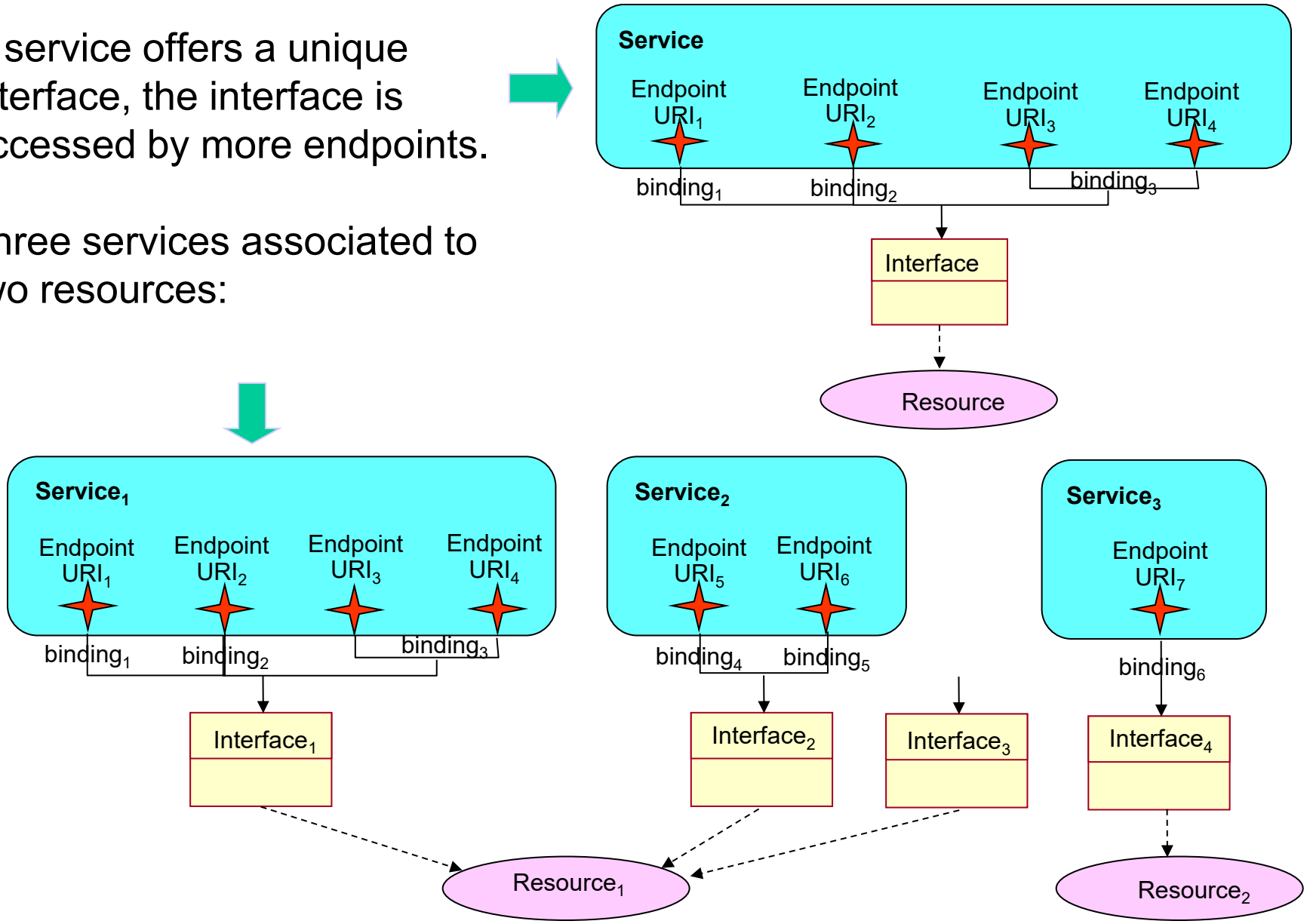
WSDL considers an interface as a behavior of a web resource. As a consequence:

A service is a collection of endpoints bound to the same resource.

Service, interface, endpoint

A service offers a unique interface, the interface is accessed by more endpoints.

Three services associated to two resources:



WSDL Definitions

```
<definitions
    targetNamespace="xs:anyURI" >
    <documentation />?
    [ <import /> | <include /> ]*
    <types />?
    [ <message /> | <interface /> | <binding /> |
    <service /> ]*
</definitions>
```

Abstractly, the `definitions` component is a container for two categories of components:

1. WSDL components : messages, interfaces, bindings and services.
2. type system components, element declarations and type definitions extracted from any type system.

Element declarations define *element information items* with the following properties: [local name], [namespace name], [children] and [attributes]

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions . . . >
  <import . . . >
    . . .
  </import>
  <types>
    . . .
  </types>
  <message . . . >
    . . .
  </message>
  <message . . . >
    . . .
  </message>
  <portType . . . >
  . . .
  </portType>
  <binding . . .>
  . . .
  </binding>
  <service . . . >
  . . .
  </service>
  <documentation >
  . . .
  </documentation>
</definitions>
```

WSDL Document Structure

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions . . . >
  <import . . . >
    . . .
  </import>
  <types>
    . . .
  </types>
  <message . . . >
    . . .
  </message>
  <message . . . >
    . . .
  </message>
  <porType . . . >
  . . .
  </portType>
  <binding . . .>
  . . .
  </binding>
  <service . . . >
  . . .
  </service>
  <documentation >
  . . .
  </documentation>
</definitions>
```

definitions = root **element**

The import element is similar to an import element in an XML schema document; it's used to import WSDL definitions from other WSDL documents.

The types element uses the XML schema language to declare complex data types and elements that are used elsewhere in the WSDL document.

The message element describes the message's payload using XML schema built-in types, complex types, or elements that are defined in the WSDL document's types element, or defined in an external WSDL document the import element refers to.

The portType and operation elements describe a Web service's interface and define its methods. An operation element uses one or more message types to define its input and output payloads.

The binding element assigns a portType and its operation elements to a particular protocol and encoding style.

The service element is responsible for assigning an Internet address to a specific binding.

The documentation element explains some aspect of the WSDL document to human readers.

Example of a WSDL Document

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="BookQuoteWS"
 targetNamespace="http://www.MySite.com/BookQuote"
 xmlns:mh="http://www.MySite.com/BookQuote"
 xmlns:soapbind="http://schemas.xmlsoap.org/wsdl/soap/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns="http://schemas.xmlsoap.org/wsdl/">
  <message name="GetBookPriceRequest">
    <part name="isbn" type="xsd:string" />
  </message>
  <message name="GetBookPriceResponse">
    <part name="price" type="xsd:float" />
  </message>
  <portType name="BookQuote">
    <operation name="getBookPrice">
      <input name="isbn" message="mh:GetBookPriceRequest"/>
      <output name="price" message="mh:GetBookPriceResponse"/>
    </operation>
  </portType>
  <binding name="BookPrice_Binding" type="mh:BookQuote">
    <soapbind:binding style="rpc"
      transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="getBookPrice">
      <soapbind:operation style="rpc"
       soapAction= "http://www.MySite.com/BookQuote/GetBookPrice"/>
        <input>
          <soapbind:body use="literal"
           namespace="http://www.MySite.com/BookQuote" />
        </input>
        <output>
          <soapbind:body use="literal"
           namespace="http://www.MySite.com/BookQuote" />
        </output>
     </operation>
  </binding>
  <service name="BookPriceService">
    <port name="BookPrice_Port" binding="mh:BookPrice_Binding">
      <soapbind:address location= "http://www.MySite.com/BookQuote" />
    </port>
  </service>
</definitions>
```

The `definitions` Element

```
<definitions name="BookQuoteWS"
 targetNamespace="http://www.MySite.com/jwsbook/BookQuote"
 xmlns:mh="http://www.MySite.com/jwsbook/BookQuote"
 xmlns:soapbind="http://schemas.xmlsoap.org/wsdl/soap/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns="http://schemas.xmlsoap.org/wsdl/">
```

- The `definitions` element usually contains several XML namespace declarations, which is normal for a root element. Among these declarations is one for the namespace of the WSDL 1.1 XML schema `http://schemas.xmlsoap.org/wsdl/`. The first attribute is `name` which is optional and used to name the entire WSDL document.
- The `definitions` element also declares a `targetNamespace` attribute which identifies the namespace of elements defined in the WSDL document.
- The `message`, `portType` and `binding` elements are assigned labels using their `name` attributes; these labels automatically take on the namespace specified by the `targetNamespace` attribute. These elements are called **definitions**.

```
<message name="GetBookPriceRequest">
  <part name="isbn" type="xsd:string" />
</message>
<message name="GetBookPriceResponse">
  <part name="price" type="xsd:float" />
</message>
<portType name="BookQuote">
  <operation name="getBookPrice">
    <input name="isbn" message="mh:GetBookPriceRequest"/>
    <output name="price" message="mh:GetBookPriceResponse"/>
  </operation>
</portType>
```

The types element

The types element serves as a container for defining any data types that are not described by the XML schema built-in types: complex types and custom simple types.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="BookQuoteWS"
 targetNamespace="http://www.MySite.com/jwsbook/BookQuote"
 xmlns:mh="http://www.MySite.com/jwsbook/BookQuote"
 xmlns:soapbind="http://schemas.xmlsoap.org/wsdl/soap/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns="http://schemas.xmlsoap.org/wsdl/">
  <types>
    <xsd:schema
     targetNamespace="http://www.MySite.com/jwsbook/BookQuote">
      <!-- The ISBN simple type -->
      <xsd:simpleType name="ISBN">
        <xsd:restriction base="xsd:string">
          <xsd:pattern value="[0-9]{9}[0-9Xx]"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:schema>
  </types>
  <message name="GetBookPriceRequest">
    <part name="isbn" type="mh:ISBN" />
  </message>
  <message name="GetBookPriceResponse">
    <part name="price" type="xsd:float" />
  </message>
   ...
</definitions>
```

The custom simple type, labeled ISBN is defined and assigned to the mh namespace by the XML schema targetNamespace attribute. The mh:ISBN type is then used to define the isbn part of the GetBookPriceRequest message definition.

The `import` Element

- The `import` element makes available in the present WSDL document the definitions of a specified namespace in another WSDL document. This feature can be useful if you want to modularize WSDL documents - for example, to separate the abstract definitions (the `types`, `message`, and `portType` elements) from the concrete definitions (the `binding`, `service`, and `port` elements).
- Another reason to use `import` is to consolidate into one WSDL document several definitions you want to maintain separately.

```
<definitions name="AllMhWebServices"
 xmlns="http://schemas.xmlsoap.org/wsdl/">
  <import namespace="http://www.MySite.com/jwsbook/BookQuote"
   location="http://www.MySite.com/jwsbook/BookPrice.wsdl"/>
  <import namespace="http://www.MySite.com/jwsbook/po"
   location="http://www.MySite.com/jwsbook/wsdl/PurchaseOrder.wsdl"/>
  <import namespace="http://www.MySite.com/jwsbook/Shipping"
   location="http://www.MySite.com/jwsbook/wsdl/Shipping.wsdl"/>
</definitions >
```

- The WSDL `import` element must declare two attributes: `namespace` and `location`. The value of the `namespace` attribute must match the `targetNamespace` declared by the WSDL document being imported. The `location` attribute must point to an actual WSDL document; it cannot be empty or null.

The `message` Element for RPC-Style Web Services

- A `message` element can describe the payloads of outgoing or incoming messages - that is, messages that are directly sent to or received from a Web service. In addition, the `message` element can describe the contents of SOAP header blocks and fault `detail` elements.
- The way to define a `message` element depends on whether you use RPC-style or document-style messaging.

```
<!-- message elements describe the input and output parameters -->
<message name="GetBookPriceRequest">
  <part name="isbn" type="xsd:string" />
</message>
<message name="GetBookPriceResponse">
  <part name="price" type="xsd:float" />
</message>
```

- The `GetBookPriceRequest` message definition describes the payload of the message transmitted from a SOAP client to the `BookQuote` Web service, while the `GetBookPriceResponse` message definition describes the payload of the message transmitted by the `BookQuote` Web service back to the client.
- In RPC-style messaging `message` have more than one part :

```
<definitions name="BookPrice" ...>
  ...
  <message name="GetBulkBookPriceRequest">
    <part name="isbn" type="xsd:string"/>
    <part name="quantity" type="xsd:int"/>
  </message>
  <message name="GetBulkBookPriceResponse">
    <part name="price" type="mh:prices" />
  </message>
  ...
</definitions>
```

The `message` Element for Document-Style Web Services

When you use document-style messaging, the message definition refers to a top-level `element` in the types definition. The following Web service uses One-Way messaging, so there is no reply message, which is common in document-style messaging.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="PurchaseOrderWS"
 targetNamespace="http://www.MySite.com/jwsbook/PO"
 xmlns:mh="http://www.MySite.com/jwsbook/PO"
 xmlns:soapbind="http://schemas.xmlsoap.org/wsdl/soap/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns="http://schemas.xmlsoap.org/wsdl/">
  <types>
    <xsd:schema targetNamespace="http://www.MySite.com/jwsbook/PO">
      <!-- Import the PurchaseOrder XML schema document -->
      <xsd:import namespace="http://www.MySite.com/jwsbook/PO"
       schemaLocation="http://www.MySite.com/jwsbook/po.xsd" />
    </xsd:schema>
  </types>
  <!-- message elements describe the input and output parameters -->
  <message name="SubmitPurchaseOrderMessage">
    <part name="order" element="mh:purchaseOrder" />
  </message>
   ...
</definitions>
```

- A message part may declare either a `type` attribute or an `element` attribute, but not both.
- RPC-style messaging uses types to define procedure calls, where each element represents a type of parameter.
- Document-style messaging, on the other hand, exchanges XML document fragments and refers to their top-level (global) elements.

The portType Element

A portType defines the abstract interface of a Web service. In WSDL the portType is implemented by the binding and service elements, which associate the Internet protocols, encoding schemes, and an Internet address used by a Web service implementation.

The "methods" of the portType are its operation elements.

WSDL portType

```
<portType name="BookQuote">
  <operation name="GetBookPrice">
    <input
     name="isbn"
     message="mh:GetBookPriceRequest"/>
    <output
     name="price"
     message="mh:GetBookPriceResponse"/>
  </operation>
<!-- use the parameterOrder attribute to
   specify the order of input parameters-->
  <operation name="GetBulkBookPrice"
   parameterOrder="isbn quantity">
    <input
     name="request"
     message="mh:GetBulkBookPriceRequest"/>
    <output name="prices"
     message="mh:GetBulkBookPriceResponse"/>
  </operation>
  <operation name="GetBookIsbn">
    <input name="title"
     message="mh:GetBookIsbnRequest"/>
    <output
     name="isbn"
     message="mh:GetBookIsbnResponse"/>
  </operation>
</portType>
```

Conceptually, portType is a lot like a Java interface because it defines an abstract type and its methods, but not an implementation.

Java Interface

```
public interface BookQuote {
  public float getBookPrice (
         String isbn);

  public float getBulkBookPrice (
         String isbn, int quantity);

  public String getBookIsbn (
         String bookTitle);
}
```

The `operation` Element

```
<portType name="BookQuote">
  <operation name="getBookPrice">
    <input name="isbn" message="mh:GetBookPriceRequest"/>
    <output name="price" message="mh:GetBookPriceResponse"/>
    <fault name="InvalidArgumentFault" message="mh:InvalidArgumentFault"/>
  </operation>
</portType>
```

Each operation element declared by a portType uses one or more message definitions to define its input, output, and faults.

An input message represents the payload sent to the Web service, and the output message represents the payload sent to the client.

- In RPC-style the input is the request, and the output is the response.
- In document-style the input is the XML document fragment sent to the Web service, and the output is the XML document fragment sent back to the client.

```
<message name="GetBulkBookPriceRequest">
    <part name="isbn" type="xsd:string"/>
    <part name="quantity" type="xsd:int"/>
  </message>
  <message name="GetBulkBookPriceResponse">
    <part name="prices" type="mh:prices" />
  </message>
  <portType name="GetBulkBookPrice" >
    <operation name="getBulkBookPrice" parameterOrder="isbn quantity">
      <input name="request" message="mh:GetBulkBookPriceRequest"/>
      <output name="prices" message="mh:GetBulkBookPriceResponse"/>
    </operation>
  </portType>
```

Overloading: Every operation defined by a particular `portType` must have a unique name. But it's perfectly acceptable for two or more `portType` elements to declare `operation` elements with the same name.

Pattern per lo scambio di messaggi

There are 4 basic message exchange patterns (MEPs) used in Web services :

- Request/Response,
- One-Way.

- A WSDL document can dictate the MEP for a specific operation by the way it declares its `input` and `output` elements.

Request/Response Pattern

The `operation` element is declared with a single `input` element followed by a single `output` element and/or `fault` elements:

```
<portType name="BookQuote">
  <operation name="getBookPrice">
    <input name="isbn" message="mh:GetBookPriceRequest"/>
    <output name="price" message="mh:GetBookPriceResponse"/>
    <fault name="InvalidArgumentFault" message="mh:InvalidArgumentFault"/>
    <fault name="SecurityFault" message="mh:SecurityFault"/>
  </operation>
</portType>
```

One-Way Pattern

The `operation` element declares only `input` elements. This MEP is typically thought of as asynchronous messaging :

```
<portType name="SubmitPurchaseOrder_PortType">
  <operation name="SubmitPurchaseOrder">
    <input name="order" message="mh:SubmitPurchaseOrderMessage"/>
  </operation>
</portType>
```

One-Way operations may not specify fault elements and do not generate fault messages.

WSDL Implementation: the `binding` Element

The `binding` element maps an abstract `portType` to a set of concrete protocols such as SOAP and HTTP, messaging styles (RPC or document), and encoding styles (Literal or SOAP Encoding). The `binding` element, and its subelements, are used in combination with protocol-specific elements. The `binding` elements identify which `operation` and `portType` elements are being bound, while the protocol-specific elements declare the protocol and encoding style to be associated with the `portType`. Each type of protocol (SOAP, MIME, and HTTP) has its own set of protocol-specific elements and its own namespace.

The binding element is actually composed of two different namespaces. The elements without a namespace prefix (shown in bold) are members of the WSDL 1.1 namespace "`http://schemas.xmlsoap.org/wsdl/`", which is the default namespace of the WSDL document. The WSDL 1.1-generic binding elements are `binding`, `operation`, `input`, and `output`. The `soapbind:binding`, `soapbind:operation`, and `soapbind:body` elements, on the other hand, are protocol-specific, that is members of the namespace for the SOAP-WSDL binding, `http://schemas.xmlsoap.org/wsdl/soap/`.

The binding element

```
<?xml version="1.0" encoding="UTF-8"?>
    <definitions name="BookQuoteWS"
     targetNamespace="http://www.MySite.com/BookQuote"
     xmlns:mh=http://www.MySite.com/BookQuote
     xmlns:soapbind="http://schemas.xmlsoap.org/wsdl/soap/"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
     xmlns="http://schemas.xmlsoap.org/wsdl/">  ...
      <!-- binding tells us which protocols and encoding styles are used -->
      <binding name="BookPrice_Binding" type="mh:BookQuote">
        <soapbind:binding style="rpc"
         transport="http://schemas.xmlsoap.org/soap/http"/>
        <operation name="getBookPrice">
          <soapbind:operation style="rpc"
           soapAction="http://www.MySite.com/BookQuote/GetBookPrice"/>
            <input>
              <soapbind:body use="literal"
               namespace="http://www.MySite.com/BookQuote" />
            </input>
            <output>
              <soapbind:body use="literal"
               namespace="http://www.MySite.com/BookQuote" />
            </output>
          </operation>
       </binding> ...
    </definitions>
```

Binding to `portType` mapping

- The `soapbind:body` and `soapbind:binding` elements are responsible for expressing the SOAP-specific details of the Web service. For example, `soapbind:binding` tells us that the messaging style is RPC and that the network application protocol is HTTP. The `soapbind:body` element tells us that both the input and output messages use literal encoding.
- The children of the (`operation`, `input` and `output`) map directly to the corresponding children of the `portType` element.

```
<portType name="BookQuote">
    <operation name="getBookPrice">
      <input name="isbn" message="mh:GetBookPriceRequest"/>
      <output name="price" message="mh:GetBookPriceResponse"/>
    </operation>
</portType> . . .
<binding name="BookPrice_Binding" type="mh:BookQuote">
    <soapbind:binding style="rpc"
      transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="getBookPrice">
      <soapbind:operation style="rpc"
       soapAction= "http://www.MySite.com/jwsbook/BookQuote/GetBookPrice"/>
        <input>
          <soapbind:body use="literal"
           namespace="http://www.MySite.com/jwsbook/BookQuote" />
        </input>
        <output>
          <soapbind:body use="literal"
           namespace="http://www.MySite.com/jwsbook/BookQuote" />
        </output>
    </operation>
</binding>
```

SOAP Binding

Several SOAP 1.1-specific binding elements are used in combination with the WSDL binding elements. These include `soapbind:binding`, `soapbind:operation`, `soapbind:body`, `soapbind:fault`, `soapbind:header`, and `soapbind:headerfault`. The `soapbind:binding` and `soapbind:body` elements are required, but the other elements are optional. Using the `soapbind` prefix is not required; you can use any prefix, just as with any XML namespace. The namespace assigned to that prefix must, however, be associated with the namespace defined by WSDL 1.1 for SOAP 1.1 bindings, "`http://schemas.xmlsoap.org/wsdl/soap/`".

The `soapbind:binding` Element

It identifies the Internet protocol used to transport SOAP messages and the default messaging style (RPC or document) of its operations:

```
<binding name="BookPrice_Binding" type="mh:BookQuote">
  <soapbind:binding style="rpc"
  transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="getBookPrice">
      <soapbind:operation style="rpc" soapAction=
        "http://www.MySite.com/jwsbook/BookQuote/GetBookPrice"/>
          <input>
            <soapbind:body use="literal"
             namespace="http://www.MySite.com/jwsbook/BookQuote" />
          </input> . . .
```

The soapbind:operation Element

- The soapbind:operation element is required. It specifies the messaging style (RPC or document) for a specific operation and the value of the SOAPAction header field.

```
<!-- binding tells us which protocols and encoding styles are used -->
<binding name="BookPrice_Binding" type="mh:BookQuote">
  <soapbind:binding style="rpc"
   transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="getBookPrice">
    <soapbind:operation style="rpc"
     soapAction="http://www.MySite.com/jwsbook/BookQuote/GetBookPrice"/>
      <input>
        <soapbind:body use="literal"
         namespace="http://www.MySite.com/jwsbook/BookQuote" />
      </input>
      <output>
        <soapbind:body use="literal"
         namespace="http://www.MySite.com/jwsbook/BookQuote" />
      </output>
  </operation>
</binding>
```

- The soapAction attribute dictates the value that must be placed in the SOAPAction header field of the HTTP request message. It can be omitted or empty. In this case the SOAPAction HTTP header field must be present and must contain an empty string. deve essere presente e contenere””. The value of the SOAPAction HTTP header field must match, exactly, the value of the corresponding SOAPAction attribute in the soapbind:operation element.

An HTTP SOAP Message Declaring a `SOAPAction` Header Field

The style of messaging has a direct impact on how the body of the SOAP message is constructed, so declaring the correct style, "rpc" or "document", is important. When you use RPC-style messaging, the Body of the SOAP message will contain an element that represents the operation to be performed. This element gets its name from the operation defined in the portType.

```
POST 1ed/BookQuote HTTP/1.1
Host: www.Monson-Haefel.com
Content-Type: text/xml;
charset="utf-8" Content-Length: nnnn
SOAPAction="http://www.MySite.com/jwsbook/BookQuote/GetBookPrice"

<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:mh="http://www.Monson-Haefel.com/jwsbook/BookQuote">
  <soap:Body>
    <mh:getBookPrice>
       <isbn>0321146182</isbn>
    </mh:getBookPrice>
  </soap:Body>
</soap:Envelope>
```

The soapbind:body Element

The attributes of the soapbind:body element change depending on whether you use RPC- or document-style messaging. The soapbind:body element has four kinds of attributes: use, namespace, part, and encodingStyle:

- use, the value must be "literal"
- the namespace attribute must be specified with a valid URI for "rpc"-style messages.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="BookQuoteWS"
  targetNamespace="http://www.MySite.com/jwsbook/BookQuote"
  xmlns:mh="http://www.Monson-Haefel.com/jwsbook/BookQuote"
  xmlns:soapbind="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns="http://schemas.xmlsoap.org/wsdl/">
 ...
  <!-- binding tells us which protocols and encoding styles are used -->
  <binding name="BookPrice_Binding" type="mh:BookQuote">
    <soapbind:binding style="rpc"
      transport="http://schemas.xmlsoap.org/soap/http"/>
       <operation name="getBookPrice">
         <soapbind:operation style="rpc"
          soapAction=
            "http://www.MySite.com/jwsbook/BookQuote/GetBookPrice"/>
            <input> <soapbind:body use="literal"
                namespace="http://www.MySite.com/jwsbook/BookQuote" />
            </input>
            <output> <soapbind:body use="literal"
                namespace="http://www.MySite.com/jwsbook/BookQuote" />
            </output> . . .
```

Document-Style Binding Doesn't Declare the `namespace` Attribute

Document-style messages must not specify the namespace attribute in the `soapbind:body` element. The namespace of the XML document fragment is derived from its XML schema.

```
<!-- binding tells us which protocols and encoding styles are used -->
<binding name="SubmitPurchaseOrder_Binding" type="mh:SubmitPurchaseOrder">
  <soapbind:binding style="document"
   transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="submit">
    <soapbind:operation style="document"/>
      <input>
        <soapbind:body use="literal" />
      </input>
      <output>
        <soapbind:body use="literal" />
      </output>
  </operation>
</binding>
```

The soapbind:fault Element

In addition to the soapbind:body element, a binding operation may also declare fault elements. The fault message elements are on a par with the input and output message elements.

```
<binding name="BookPrice_Binding" type="mh:BookQuote">
  <soapbind:binding style="rpc"
   transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="getBookPrice">
      <soapbind:operation style="rpc"
       soapAction=
       "http://www.MySite.com/jwsbook/BookQuote/GetBookPrice"/>
      <input>
        <soapbind:body use="literal"
         namespace="http://www.MySite.com/jwsbook/BookQuote" />
      </input>
      <output>
        <soapbind:body use="literal"
         namespace="http://www.MySite.com/jwsbook/BookQuote" />
      </output>
     <fault name="InvalidArgumentFault">
       <soapbind:fault name="InvalidArgumentFault" use="literal" />
     </fault>
  </operation>
</binding>
```

The WSDL fault and soapbind:fault elements include a mandatory name attribute, which refers to a specific fault message declared in the associated port Type.

The soapbind:header Element

- WSDL explicitly identifies a SOAP header block by using the soapbind:header element in the binding's input element, its output element, or both. Example: a binding that describes the message-id header block:

```
<types>
  <xsd:schema
    targetNamespace="http://www.MySite.com/jwsbook/BookQuote"
    xmlns="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="message-id" type="string" />
  </xsd:schema>
</types>
<!-- message elements describe the input and output parameters -->
<message name="Headers">
 <part name="message-id" element="mh:message-id" />
</message>
<message name="GetBookPriceRequest">
  <part name="isbn" type="xsd:string" />
</message>
<message name="GetBookPriceResponse">
  <part name="price" type="xsd:float" />
</message>
<!--portType element describes the abstract interface of a Web service-->
<portType name="BookQuote">
  <operation name="getBookPrice">
    <input name="isbn" message="mh:GetBookPriceRequest"/>
    <output name="price" message="mh:GetBookPriceResponse"/>
  </operation>
</portType> . . .
```

WSDL Implementation: The `service` and `port` Elements

- The `service` element contains one or more `port` elements, each of which represents a different Web service. The `port` element assigns the URL to a specific binding. A `service` may have a single `port` element:

```
<service name="BookPriceService">
  <port name="BookPrice_Port" binding="mh:BookPrice_Binding">
    <soapbind:address location="http://www.MySite.com/jwsbook/BookQuote"/>
  </port>
</service>
```

- A `service` may have many `port` elements:

```
<service name="BookPriceService">
  <port name="BookPrice_Port" binding="mh:BookPrice_Binding">
    <soapbind:address location="http://www.MySite.com/jwsbook/BookQuote"/>
  </port>
  <port name="BookPrice_Failover_Port" binding="mh:BookPrice_Binding">
    <soapbind:address location="http://www.MySite.org/jwsbook/BookPrice"/>
  </port>
  <port name="SubmitPurchaseOrder_Port" binding="mh:SubmitPurchaseOrder_Binding">
    <soapbind:address location= "https://www.MySite.org/jwsbook/po" />
  </port>
</service>
```

- The `soapbind:address` element assigns an Internet address to a SOAP binding via its `location` attribute (its only attribute). Although WSDL allows any type of address (HTTP, FTP, SMTP, and so on), the Basic Profile allows only those URLs that use the HTTP or HTTPS schema.

UDDI

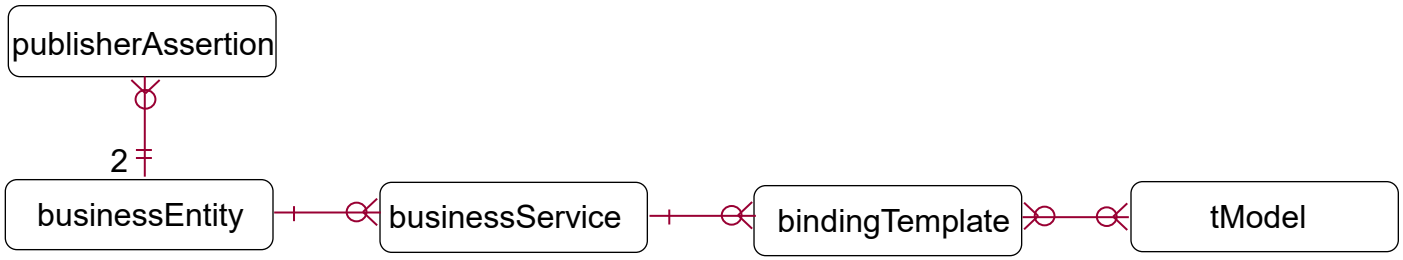
The basic goal of UDDI is to provide a standard data model for storing information about organizations and their Web services. UDDI defines a SOAP-based API that allows remote clients to access, update, and search the information in a UDDI registry.

UDDI

- UDDI (Universal Description, Discovery, and Integration) is a specification for creating a registry service that catalogs organizations and their Web services. An implementation of the UDDI specification is called a UDDI registry.
- A UDDI registry is a database that supports a set of standard data structures defined by the UDDI specification. The data structures model information about organizations (corporations, business units, government agencies, etc.) and the technical requirements for access to Web services hosted by those organizations.
- You can search a UDDI registry for specific kinds of companies or Web services. You can also register your own business and Web services in a UDDI registry.
- A UDDI registry can be compared with “yellow pages” for web services. When information on a web service is searched in a UDDI registry the searching domain can be restrained according to various categories (technology used, business type, etc.). Each entry in a UDDI registry provides information on where the web service is located and how one should communicate with it.
- A UDDI registry must provide access to its data by way of a set of SOAP-based Web services. The UDDI specification describes about thirty different SOAP operations that allow you to add, update, delete, and find information contained in a UDDI registry. Most UDDI products also provide a Web interface for access to information in the registry.

UDDI Primary Data Structures

UDDI defines five primary data structures, which are used to represent an organization, its services, implementation technologies, and relationships to other businesses :



1. A businessEntity represents the business or organization that provides the Web service.
2. A businessService represents a Web service or some other electronic service.
3. A bindingTemplate represents the technical binding of a Web service to its access point (its URL), and to tModels.
4. A tModel represents a specific kind of technology, such as SOAP or WSDL, or a type of categorization system, such as U.S. Federal Tax ID numbers or D-U-N-S numbers. The tModels that a bindingTemplate refers to reveal the type of technologies used by a Web service. Most of the data structures also use tModels to indicate the categorization system of an identifier assigned to the structure.
5. A publisherAssertion represents a relationship between two business entities.

The businessEntity Structure

```
<schema targetNamespace="urn:uddi-org:api_v2"
 xmlns="http://www.w3.org/2001/XMLSchema"
 xmlns:uddi="urn:uddi-org:api_v2"
 version="2.03" id="uddi">
 ...
<element name="businessEntity" type="uddi:businessEntity"/>
<complexType name="businessEntity">
  <sequence>
    <element ref="uddi:discoveryURLs" minOccurs="0"/>
    <element ref="uddi:name" maxOccurs="unbounded"/>
    <element ref="uddi:description" minOccurs="0" maxOccurs="unbounded"/>
    <element ref="uddi:contacts" minOccurs="0"/>
    <element ref="uddi:businessServices" minOccurs="0"/>
    <element ref="uddi:identifierBag" minOccurs="0"/>
    <element ref="uddi:categoryBag" minOccurs="0"/>
  </sequence>
  <attribute name="businessKey" type="uddi:businessKey" use="required"/>
  <attribute name="operator" type="string" use="optional"/>
  <attribute name="authorizedName" type="string" use="optional"/>
</complexType>
 ...
</schema>
```

A businessEntity Structure

```
<?xml version='1.0' encoding='UTF-8'?>
<businessEntity businessKey="01B1FA80-2A15-11D6-9B59-000629DC0A53" xmlns="urn:uddi-org:api_v2">
  <discoveryURLs>
    <discoveryURL useType="buinessEntity">
      http://uddi.ibm.com/registry/uddiget?businessKey=01B1FA80...000629DC0A53
    </discoveryURL>
  </discoveryURLs>
  <name>MySite Books, Inc.</name>
  <description xml:lang="en">Technical Book Wholesaler</description>
  <contacts>
    <contact>
      <description xml:lang="en">Web Service Tech. Support</description>
      <personName>Stanley Kubrick</personName>
      <phone useType="Voice">01-555-222-4000</phone>
    </contact>
  </contacts>
  <businessServices>
  <!-- businessService data goes here -->
  </businessServices>
  <identifierBag>
  <!-- D-U-N-S Number Identifier System -->
    <keyedReference keyName="D-U-N-S"
      keyValue="03-892-4499" tModelKey="uuid:8609C81E-EE1F-4D5A-B202-3EB13AD01823"/>
  </identifierBag>
  <categoryBag>
    <!-- North American Industry Classification System (NAICS) 1997 -->
    <keyedReference keyName="Book, Periodical, and Newspaper Wholesalers"
     keyValue="42292" tModelKey="uuid:C0B9FE13-179F-413D-8A5B-5004DB8E5BB2"/>
    <!-- Universal Standard Products and Services Classification (UNSPSC) Version 6.3 -->
    <keyedReference keyName="Educational or vocational textbooks"
     keyValue="55.10.15.09.00" tModelKey="uuid:CD153256-086A-4236-B336-6BDCBDCC6634"/>
     <!-- ISO 3166 Geographic Taxonomy -->
    <keyedReference keyName="Minnesota, USA"
     keyValue="US-MN" tModelKey="uuid:4E49A8D6-D5A2-4FC2-93A0-0411D8D19E88"/>
  </categoryBag>
</businessEntity>
```

The businessService Data Structure Definition

businessService

```
<schema targetNamespace="urn:uddi-org:api_v2"
  xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:uddi="urn:uddi-org:api_v2" version="2.03" id="uddi">
  ...
  <element name="businessService" type="uddi:businessService"/>
  <complexType name="businessService">
    <sequence>
      <element ref="uddi:name" minOccurs="0" maxOccurs="unbounded"/>
      <element ref="uddi:description" minOccurs="0" maxOccurs="unbounded"/>
      <element ref="uddi:bindingTemplates" minOccurs="0"/>
      <element ref="uddi:categoryBag" minOccurs="0"/>
    </sequence>
    <attribute name="serviceKey" type="uddi:serviceKey" use="required"/>
    <attribute name="businessKey" type="uddi:businessKey" use="optional"/>
  </complexType>
   ...
</schema>
```

The businessService Data Structure Definition

```
<?xml version='1.0' encoding='UTF-8' ?>
<businessEntity businessKey="01B1FA80-2A15-11D6-9B59-000629DC0A53"
      operator="uddi:ibm" xmlns="urn:uddi-org:api_v2">
  ...
  <businessServices>
    <businessService serviceKey="3902AEE0-3301-11D6-9F18-000629DC0A53">
      <name>BookQuoteService</name>
      <description xml:lang="en">
        Given an ISBN number, this service returns the wholesale price
      </description>
      <bindingTemplates>
        <bindingTemplate bindingKey="391E2620-3301-11D6-9F18-000629DC0A53">
          <description xml:lang="en">
            This service uses a SOAP RPC/Literal Endpoint
          </description>
          <accessPoint URLType="http">
             http://www.MySite.com/jwsed1/BookQuote
          </accessPoint>
          <tModelInstanceDetails>
            <tModelInstanceInfo tModelKey="uddi:4C9D3FE0-2A16-11D6-9B59-000629DC0A53"/>
          </tModelInstanceDetails>
        </bindingTemplate>
      </bindingTemplates>
    </businessService>
  </businessServices>
```

JAX-RPC
API Java for web services

The fundamental purpose of JAX-RPC is to make communications between Java and non-Java platforms easier, first by using BP-conformant Web service technologies like XML, SOAP, and WSDL, and then by providing a simple object-oriented API that Java developers can use to communicate using those technologies.

JAX-RPC Advantages

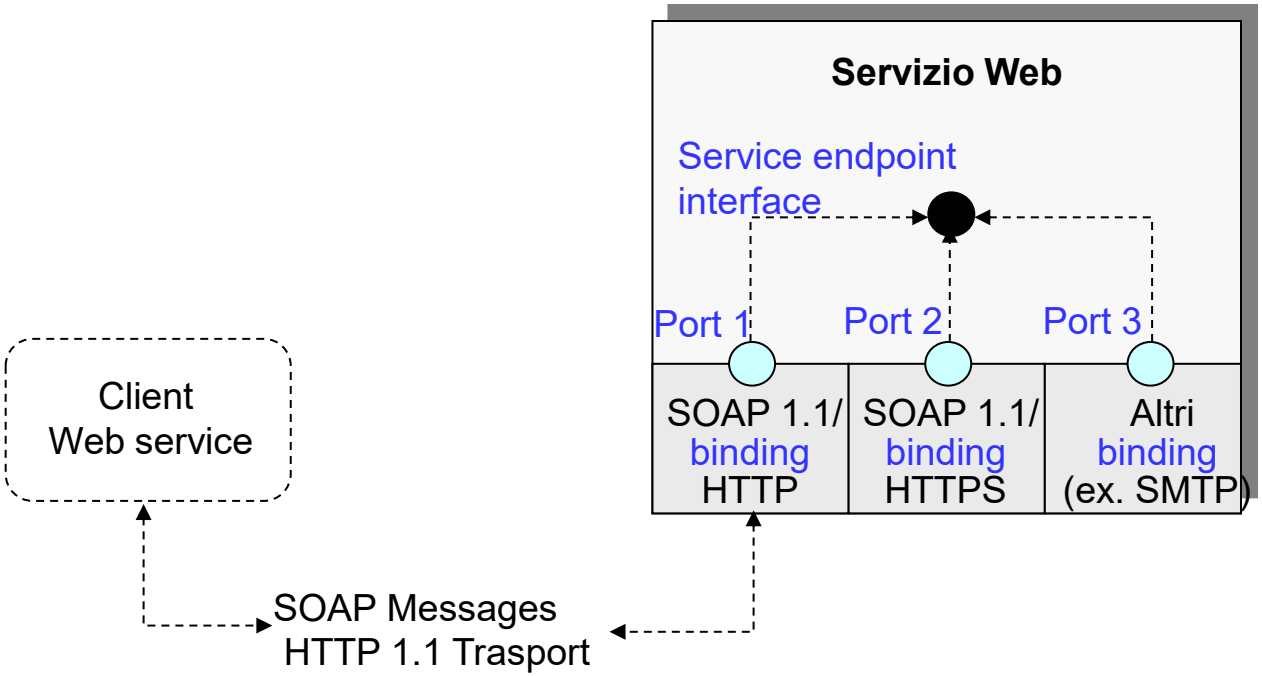
- JAX-RPC is designed as a Java API for Web services, so that J2EE applications can interoperate with non-Java applications.
- It defines the standard programming model for both Web service clients and endpoints in J2EE. There are essentially two sides to the JAX-RPC model: client-side and server-side. The client-side programming model allows you to access a remote Web service as if it were a local object, using methods that represent SOAP operations. The server-side programming model allows you to develop Web service endpoints as Java objects or Enterprise JavaBeans, which run on the J2EE platform.
- JAX-RPC defines two server-side programming models for creating J2EE Web service endpoints: JAX-RPC service endpoints and EJB endpoints. A JAX-RPC service endpoint is the simplest type of endpoint to develop and will be the one you use the most. EJB endpoints allow stateless EJBs to act as endpoints. EJB endpoints are valuable if you want to expose existing EJBs to Web service clients, or need to exploit the transactional qualities of Enterprise JavaBeans.

JAX-RPC Service Endpoint

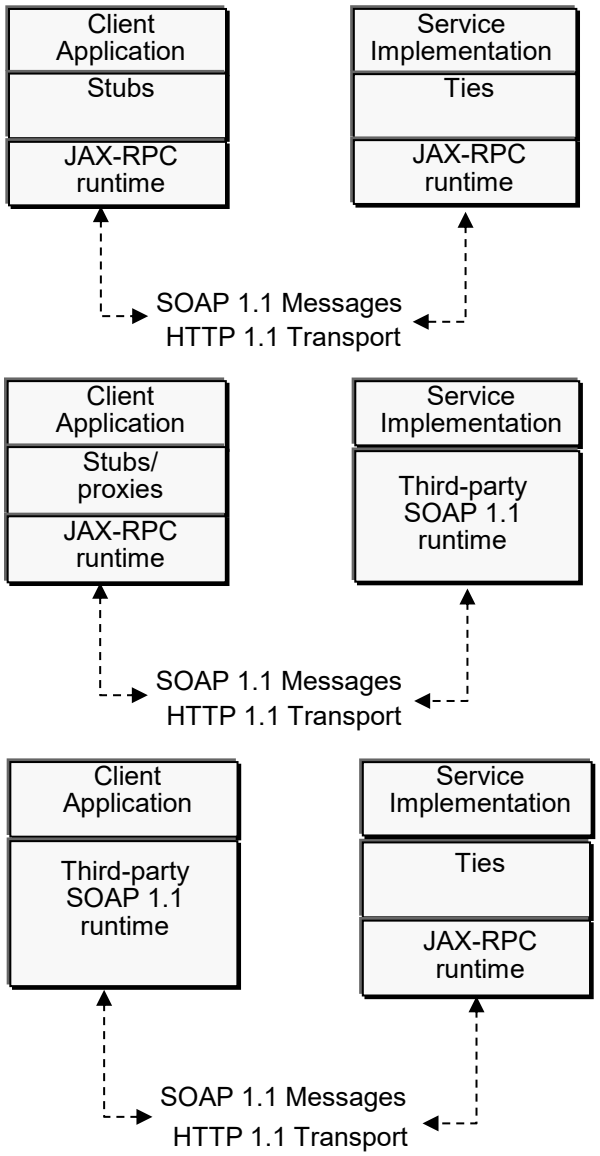
- A JAX-RPC service endpoint (JSE) is easy to develop because it's just a plain old Java object—sort of. In fact, it's a lot more than that, but on the surface it seems that simple. In reality, a JSE is a full-fledged J2EE component, which runs inside the J2EE Servlets container system. It has access to the same resources and contexts as a standard servlet, but instead of exposing HTTP streams, it marshals SOAP messages into method invocations defined by a remote interface.

Web Service Glossary

- Un servizio web consiste in un'interfaccia di endpoint del servizio (spesso chiamata semplicemente endpoint) la quale definisce una o più operazioni che il servizio web offre.
- Per promuovere la portabilità e l'indipendenza dai sottostanti meccanismi di comunicazioni, si può pensare ai servizi web come ad entità a se stanti, a parte dai protocolli usati per accedere a loro. L'accesso ad un endpoint è fornito associandolo (binding) ad uno stack di protocolli attraverso un port, che ha un indirizzo utilizzabile da un cliente può usare per comunicare con esso e per le relative operazioni.



JAX-RPC – Based Implementations



The following list introduces all software combinations actually supported by JAX-RPC :

- A JAX-RPC client implementation that connects to a JAX-RPC service implementation.
- A JAX-RPC client implementation that connects to RPC-based product provided by third parties.
- Implementation of a RPC- based SOAP 1.1 that connects to a JAX-RPC service implementation.

JAX-RPC Service Endpoint

- A JAX-RPC service endpoint (JSE) is easy to develop because it seems to be a plain old Java object. In fact, it's a lot more than that, but on the surface it seems that simple.
- In reality, a JSE is a full-fledged J2EE component, which runs inside the J2EE Servlets container system. It has access to the same resources and contexts as a standard servlet, but instead of exposing HTTP streams, it marshals SOAP messages into method invocations defined by a remote interface.

A JSE is composed of two parts:

1. an implementation class that does all the work, and
2. a java.rmi.Remote interface, called the endpoint interface, that declares the Web service's publicly accessible methods.

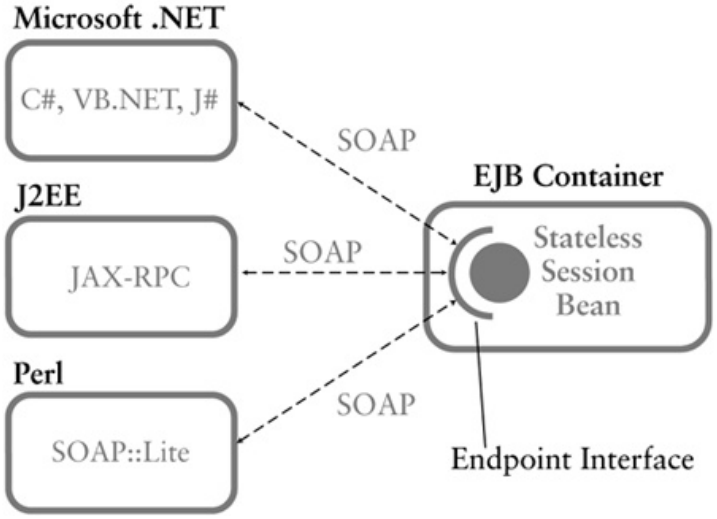
```
public interface BookQuote extends java.rmi.Remote {
  public float getBookPrice(String isbn) throws java.rmi.RemoteException;
}
public class BookQuote_Impl_1 implements BookQuote {
  public float getBookPrice(String isbn){
    return 24.99f;
  }
}
```

Once you have defined the endpoint interface and implementation objects, you can use them, along with other deployment files, to generate a WSDL document that provides a platform-independent description of the JSE.

Enterprise JavaBeans Endpoints

- JAX-RPC is also used to invoke methods on remote services via Java RMI-IIOP and proprietary flavors of Java RMI. Using JAX-RPC to talk to EJBs implies that EJBs can act as Web service endpoints, but because SOAP is a stateless messaging protocol, only stateless session beans can act as EJB endpoints. SOAP has no notion of object identity, so it can't be used with stateful session beans or entity beans.
- Unlike the remote and local interfaces already familiar to EJB developers, though, the endpoint interface does not extend an EJB object type—that is, an `EJBObject` or `EJBLocalObject`. Instead, the endpoint interface extends the `javax.ejb.Remote` interface directly. The EJB endpoint interface follows basically the same rules as the JSE endpoint interface

When developing a Web service as a stateless session bean, you'll define the endpoint interface first, then use it along with other deployment descriptors to generate the WSDL document. The WSDL document can be read and used by other Web service platforms to access the EJB endpoint



Example of EJB Endpoint

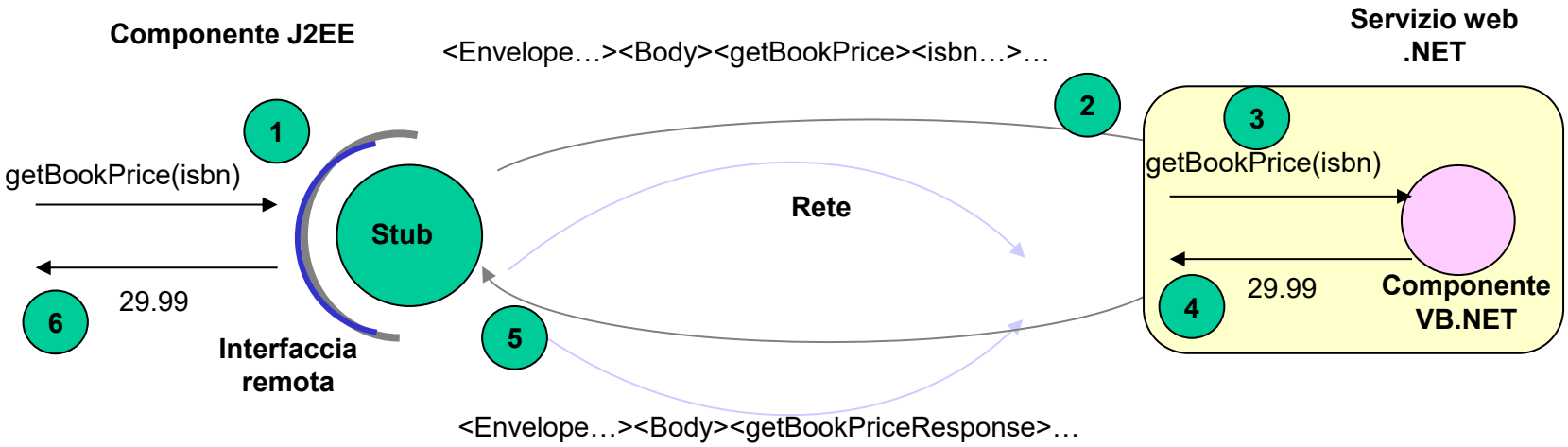
```
public interface BookQuote extends java.rmi.Remote {
 public float getBookPrice(String isbn)throws java.rmi.RemoteException;
}
public class BookPriceWS implements javax.ejb.SessionBean, BookQuote {
  public void setSessionContext(javax.ejb.SessionContext ctx){}
  public void ejbCreate(){}
  public float getBookPrice(String isbn){
    return 24.99f;
  }
  public void ejbRemove(){}
  public void ejbActivate(){}
  public void ejbPassivate(){}
}
```

The Client-Side Programming Models

The JAX-RPC client-side programming models can be used by Java applications, servlets, JSPs, JAX-RPC service endpoints (JSEs), and EJBs to exchange SOAP messages with remote Web service endpoints on any platform. JAX-RPC defines three client-programming models:

1. generated stub,
2. dynamic proxy, and
3. DII (Dynamic Invocation Interface).

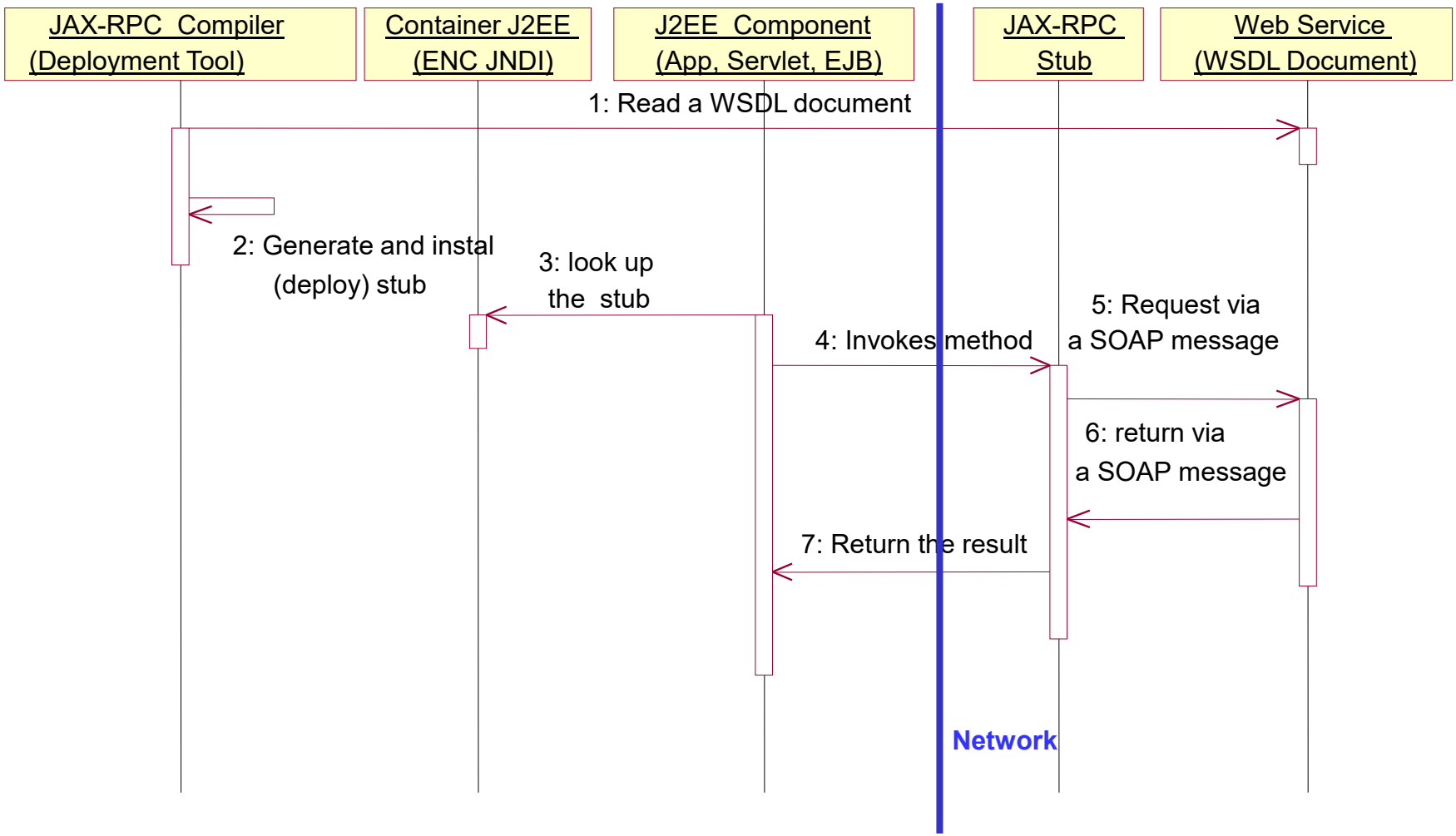
Nowadays the “generated stub” is the most used model.



Generated Stub Programming Model

- The purpose of the generated stub programming model is to provide object façades for accessing Web service endpoints. In other words, generated stubs make Web service endpoints look like local Java objects.
- **At compile time** a generated stub is created by a utility called a *JAX-RPC compiler*.
- The JAX-RPC compiler reads the WSDL document and generates from it:
  - an endpoint interface that extends java.rmi.Remote and defines the methods that can be invoked on the Web service endpoint, and
  - a generated stub class that implements the endpoint interface that is the endpoint operations.
- **At runtime** an instance of the generated stub class converts each method invocation made on the remote interface into a SOAP 1.1 message, and sends it to the Web service endpoint. The generated stub also converts each SOAP reply message into a method return value, or throws an exception if the reply message is a SOAP fault.
- Once the stubs and interfaces are generated, you can bind them to the JNDI ENC (Environment Naming Context) of any J2EE component and use them to communicate with the endpoint.

JAX-RPC Stub Generation Sequence Diagram



JAX-RPC: Mapping WSDL to Java

WSDL Document

```
<message name="getBookPrice">
  <part name="string" type="xsd:string"/>
</message>
<message name="getBookPriceResponse">
  <part name="result" type="xsd:float"/>
</message>

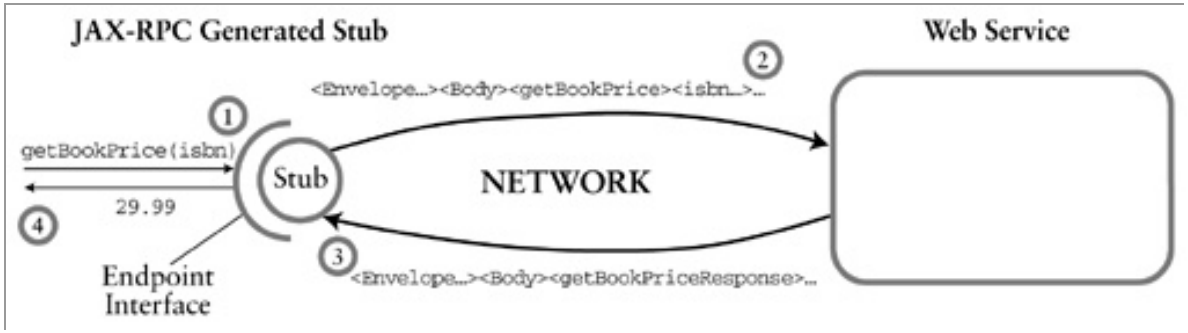
<portType name="BookQuote">
  <operation name="getBookPrice">
   <input name="isbn" message="mh:getBookPrice"/>
   <output message="mh:getBookPriceResponse"/>
  </operation>
</portType>
```

Endpoint interface generated by the JAX-RPC compiler

```
public interface BookQuote extends
   java.rmi.Remote {
  public float getBookPrice(String isbn)
      throws java.rmi.RemoteException;
}
```

Service interface generated by the JAX-RPC compiler

```
public interface BookQuoteService extends
   javax.xml.rpc.Service {
  public BookQuote getBookQuotePort( )
      throws java.rmi.RemoteException;
}
```



Invocation JAX-RPC

```
InitialContext jndiContext = new InitialContext( );
BookQuoteService service = (BookQuoteService) jndiContext.lookup(
"java:comp/env/service/BookQuoteService");
BookQuote bookQuote = service.getBookQuotePort();
float price = bookQuote.getBookPrice( isbn );
```

The Generated Stub

- An instance of the generated stub class is responsible for translating method calls into SOAP communications with the Web service endpoint. It will perform this translation according to the WSDL binding associated with the `portType`, which describes the style of messaging (`rpc` or `document`) and the type of encoding (always Literal in BP-conformant Web services). The stub will use the `soap:address` element specified by the WSDL port definition as the URL of the Web service.
- In addition to implementing the endpoint interface, the stub class will also implement the `javax.xml.rpc.Stub` interface, which defines methods for reading and setting properties relating to network communication and authentication. Using these, an application can set the endpoint address, user name, password, and so on at runtime.

Dynamic Proxy

- A dynamic proxy is used in the same way as a generated stub, except the remote interface's stub implementation is generated dynamically, at runtime.

Example: a J2EE component such as a servlet, application client, or EJB uses a JAX-RPC dynamic proxy to get the wholesale price of a book from some Web service.

```
InitialContext jndiContext = new InitialContext ( );
BookQuoteService service = (BookQuoteService) jndiContext.lookup(
            "java:comp/env/service/DynamicService");
BookQuote BookQuote_port = (BookQuote)service.getPort(BookQuote.class);
float price = BookQuote_port.getBookPrice( isbn );
```

At runtime the getPort() method automatically maps the BookQuote interface to a corresponding port definition in the WSDL document, then generates a stub for the interface that implements the protocol defined by the associated binding.

DII - Dynamic Invocation Interface

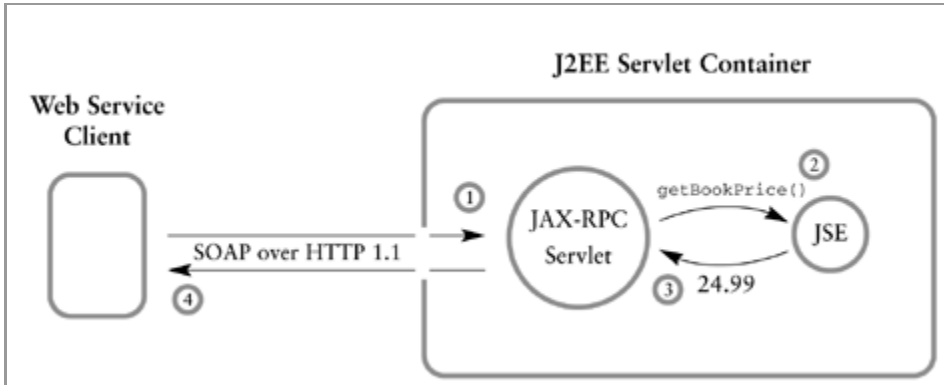
- DII allows the developer to assemble SOAP method calls dynamically at runtime. If you have used the CORBA Dynamic Invocation Interface, then JAX-RPC DII will be a familiar concept.
- JAX-RPC DII is kind of like Java Reflection. It enables you to get a reference to an object that represents a Web service operation, in the form of a method, and to invoke that method without having to use a stub or a remote interface.

```
InitialContext jndiContext = new InitialContext ( );
javax.xml.rpc.Service service = (javax.xml.rpc.Service) jndiContext.lookup(
        "java:comp/env/service/DynamicService");
QName port = new QName("http://www.xyz.com/BookQuote ","BookQuote");
QName operation = new QName("http://www.xyz.com/BookQuote","getBookPrice");
Call callObject = service.createCall(port, operation);
Object [] parameters = new Object[1];
parameters[0] = isbn;
Float price = (Float) callObject.invoke( parameters );
```

- J2EE developers will use generated stubs for access to Web services. In most cases, they will know in advance the Web service endpoints and specific operations you will be accessing, so the DII will not be necessary.

The JSE Runtime Environment

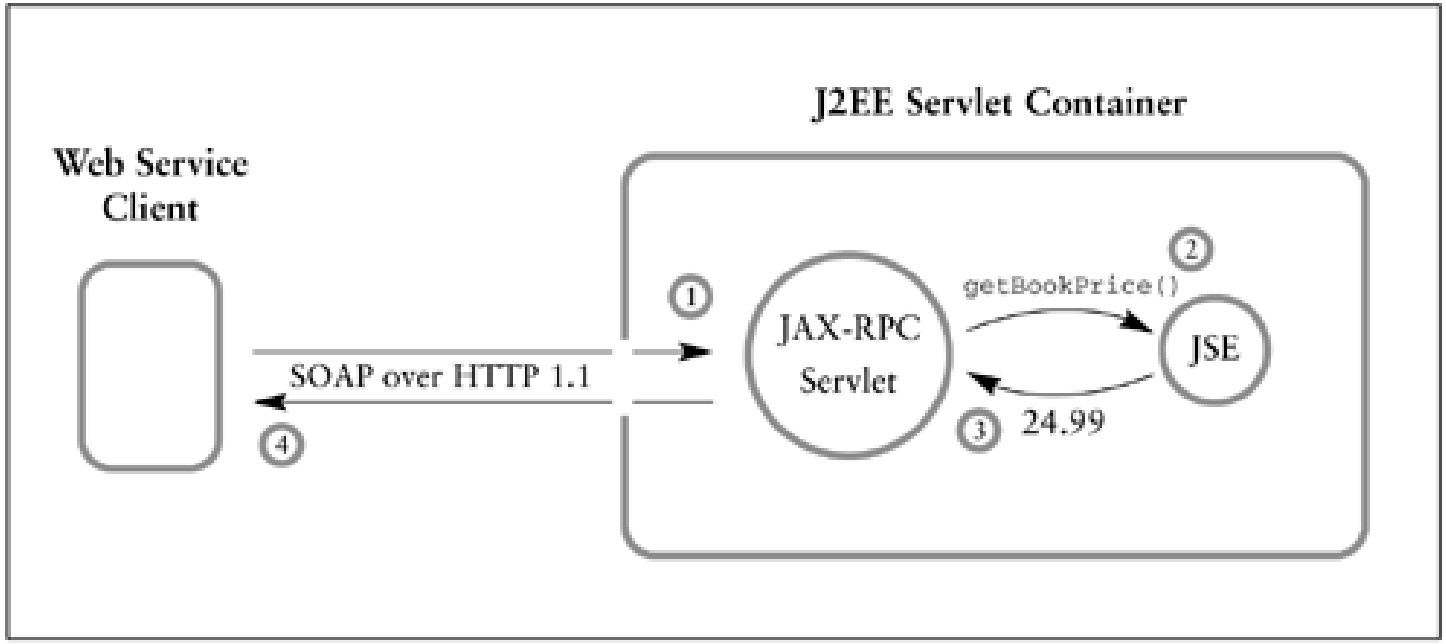
- JSEs are deployed into a J2EE servlet container, and have access to the same resources and context information a servlet has. When a JSE is deployed, it is embedded in a special JAX-RPC servlet provided by the vendor, which is responsible for responding to HTTP-based SOAP requests, parsing the SOAP messages, and invoking the corresponding methods of the JSE implementation object.



1. A Web services client sends a SOAP 1.1 request message over HTTP 1.1 to an endpoint URL that is hosted by the J2EE application server.
2. A JAX-RPC servlet receives the SOAP message and marshals it into a method call on the JSE object.
3. The JSE processes the request and returns the result to the servlet.
4. The JAX-RPC servlet converts the return value (or exception) into a SOAP message and sends it back to the client in the HTTP reply.

Vendors may generate special JAX-RPC servlets for each JSE, or multiplex all requests over a few general-purpose JAX-RPC servlets.

Servizio Web Invocation



Developing Web Services with JAX-WS

Example of a Web Service

```
package hello;
public class CircleFunctions {
 public double getArea(double radius) {
   return java.lang.Math.PI * (r * r);
 }
 public double getCircumference(double radius) {
   return 2 * java.lang.Math.PI * r;
 }
 }
```

To export these methods, you must add two things: an import statement for the javax.jws.WebService package and a @WebService annotation at the beginning that tells the Java interpreter that you intend to publish the methods of this class as a web service.

```
package hello;
import javax.jws.WebService;
@WebService
 public class CircleFunctions {
 public double getArea(double r) {
   return java.lang.Math.PI * (r * r);
 }
 public double getCircumference(double r) {
   return 2 * java.lang.Math.PI * r;
 }
```

Publishing and Execution of WS

- You can use the static publish() method of the javax.xml.ws.Endpoint class to publish the class as a web service in the specified context root:

```
import javax.xml.ws.Endpoint;
 public static void main(String[] args) {
 Endpoint.publish(
  "http://localhost:8080/WebServiceExample/circlefunctions", new
  CircleFunctions());
 }
```

- Now, compile the source code normally using javac. However, you must perform one more step: Call the Wsgen tool, as follows:

```
> wsgen -cp . hello.CircleFunctions
```

- The Wsgen tool will generate a number of source files in a subdirectory called wsgen, which it then compiles. You can browse these source code files to get an idea of how JAX-WS 2.0 creates the appropriate stub files for use while publishing the web service.
- Note that the original source files must be located in a package when you call the Wsgen tool. Otherwise, you may get an error that dictates that classes annotated with @WebService, such as CircleFunctions, must declare a separate javax.jws.Webservice.targetNamespace element because the source files are not part of a package.

Web Service WSDL Document

- When you run the application, the Java SE 6 platform has a small web application server that will publish the web service at the address http://localhost:8080/WebServiceExample/circlefunctions while the JVM is running.* You can verify that the web service is running by displaying the Web Services Definition Language (WSDL) file of the circlefunctions web service.While the JVM is still running, open a browser and go to the following location:

```
http://localhost:8080/WebServiceExample/circlefunctions?WSDL
```

JAX-WS Example

- JAX-WS is a high level API that simplifies development of web services. Developing a web service via JAX-WS consists of writing a class with public methods to be exposed as web services. Both the class and the methods need to be decorated with annotations specifying that the methods are to be exposed as web services. The following example illustrates this process:

```
import javax.jws.WebMethod;
import javax.jws.WebService;
@WebService
public class Calculator{
  @WebMethod
  public int add(int first, int second) {
    return first + second;
  }
  @WebMethod
  public int subtract(int first, int second){
    return first - second;
  }
}
```

- We indicate that the class implements a web service by decorating it with the @WebService annotation.
- Any methods that we would like exposed as web services need to be decorated with the @WebMethod annotation.
- Only public methods can be exposed as web services.

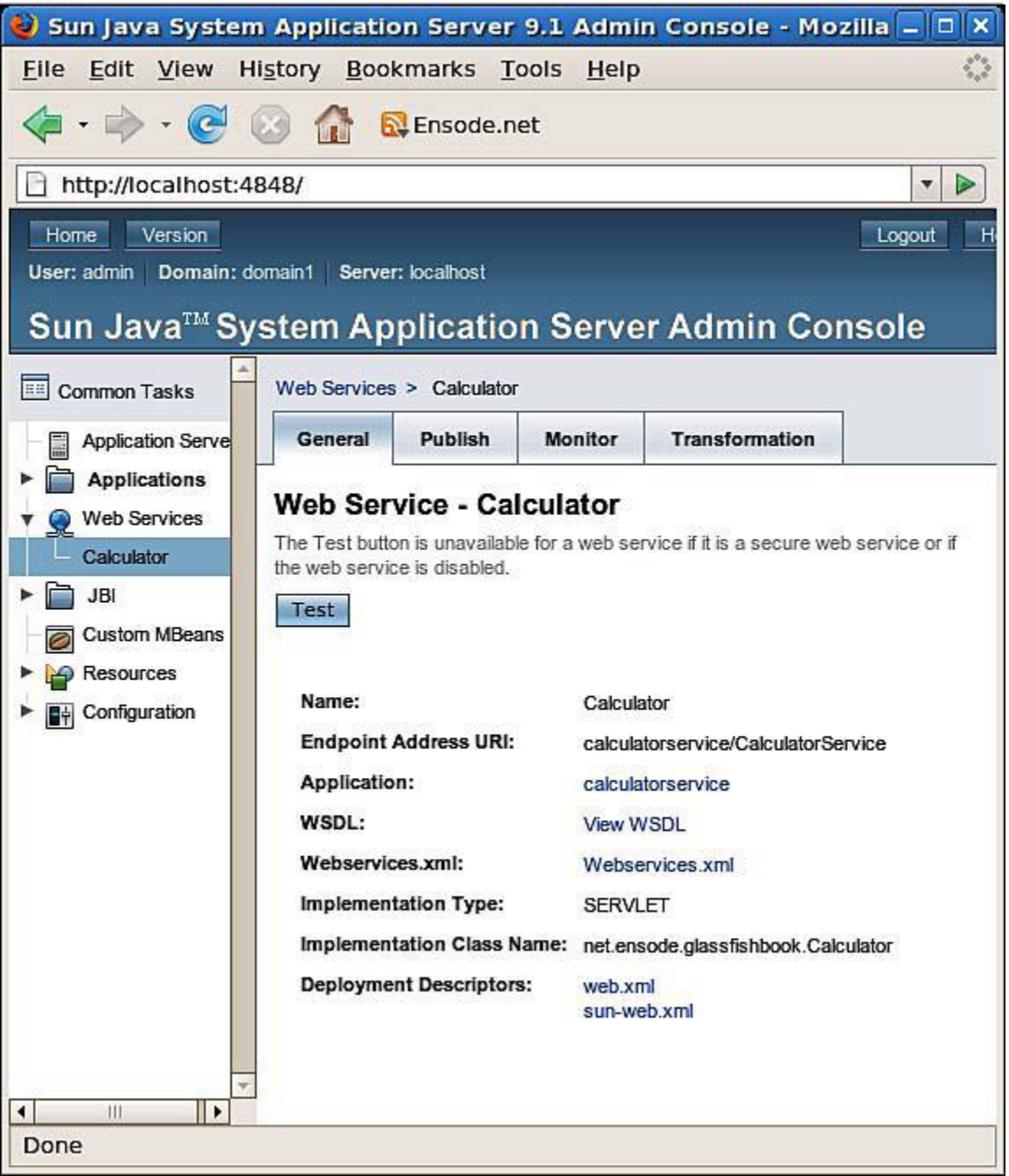
Deploying a Web Service

- To deploy our web service, we need to package it in a WAR file.
- Like any valid WAR file, our WAR file must contain a web.xml deployment descriptor in its META-INF directory.
- If no additional servlets are deployed in the WAR file, then simply having an empty <web-app> element in the deployment descriptor will be enough to successfully deploy our WAR file.
- Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee" version="2.5"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema"
   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
  http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
</web-app>
```

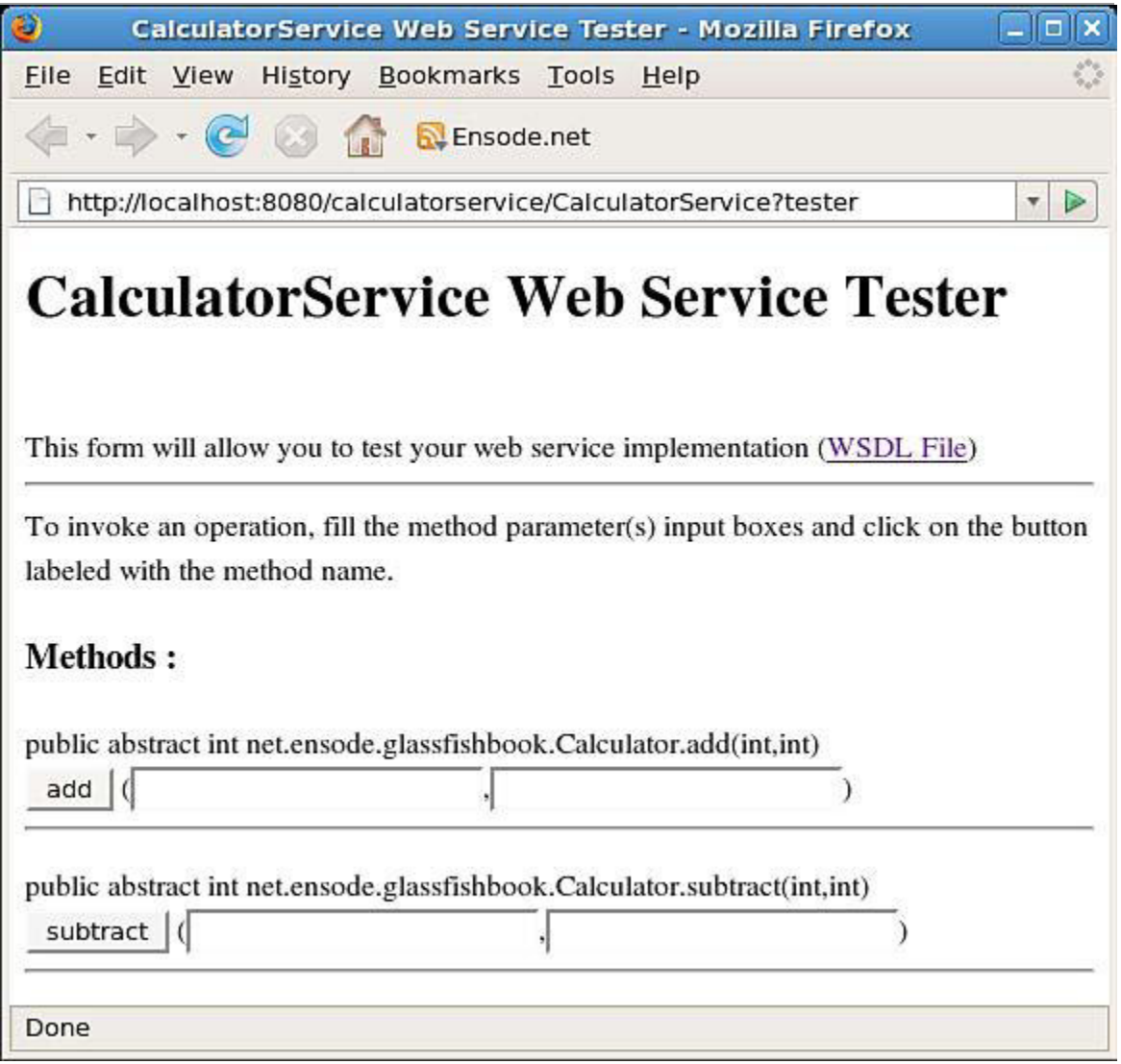
- After compiling and packaging the code and deployment descriptor in a WAR file, deploy it .

Deployment Verification



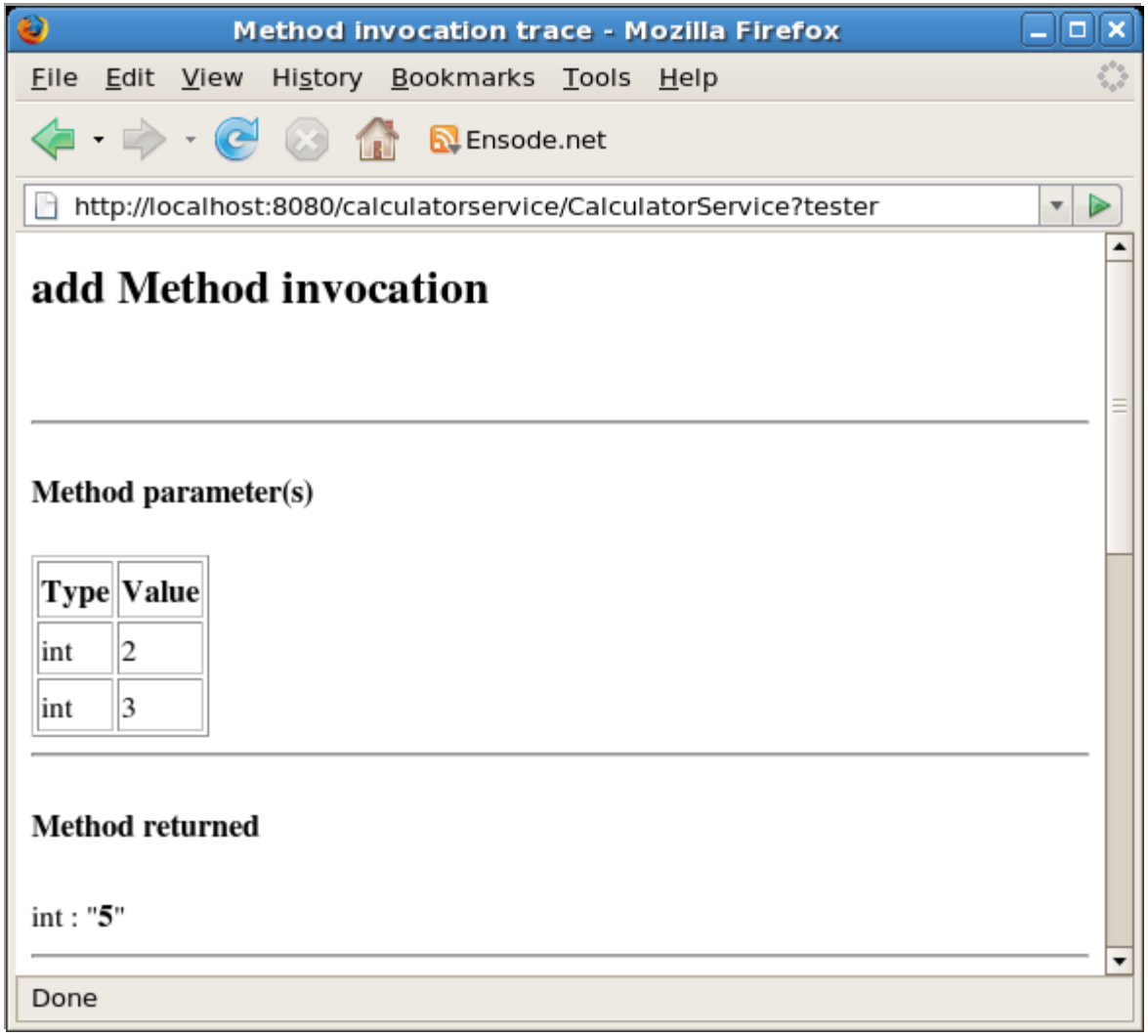
- We can verify that the service was successfully deployed by logging into the Sun Application Server Admin Web Console and expanding the Web Services node at the left-hand side.
- We can easily test the web service by clicking on the **Test button.**

Testing the Web Service



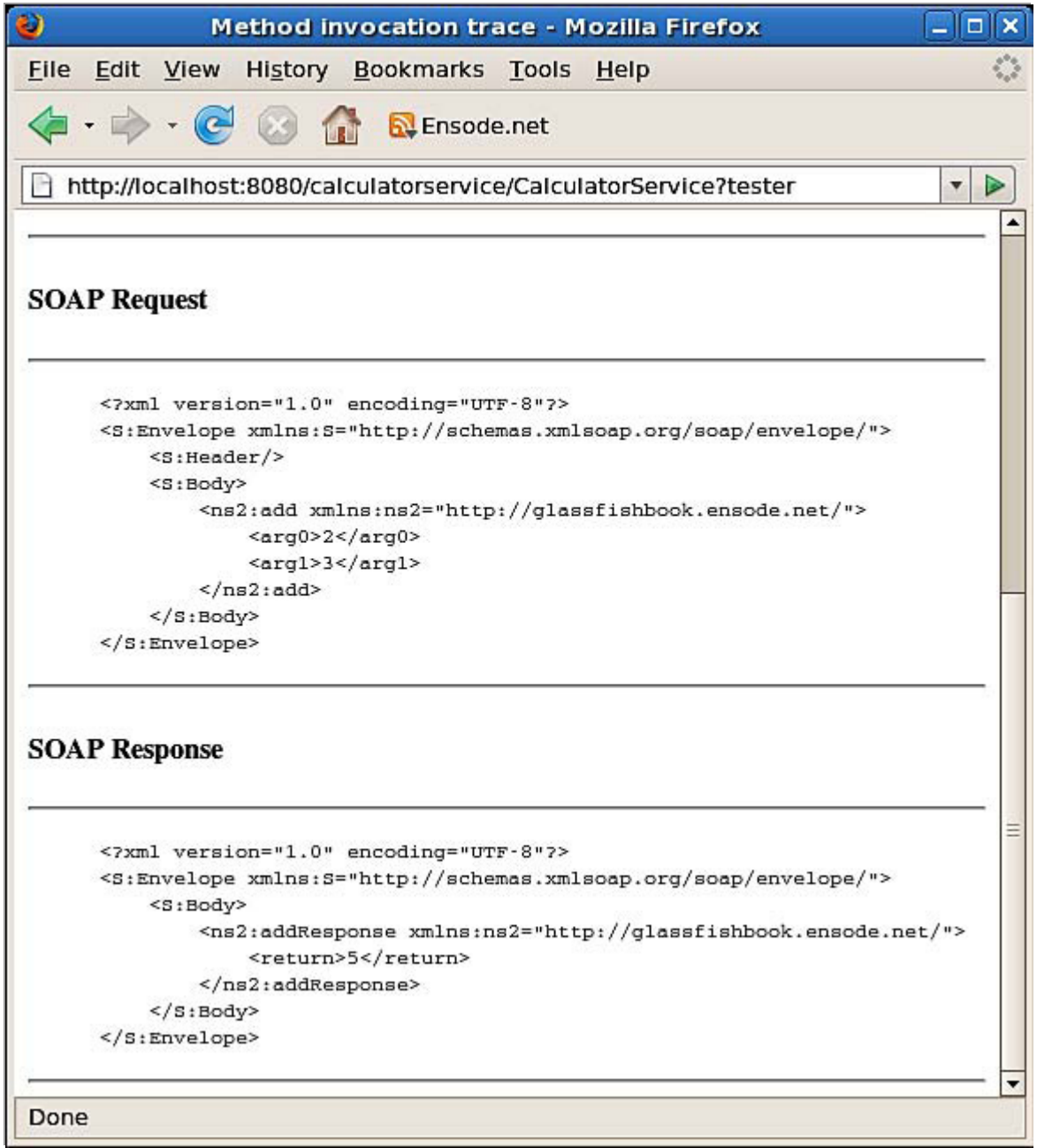
- To test the methods, we can simply enter some parameters in the text fields and click on the appropriate button. For example, entering the values 2 and 3 in the text fields corresponding to the add method, and clicking on the add button would result in a new page.

Result Displaying

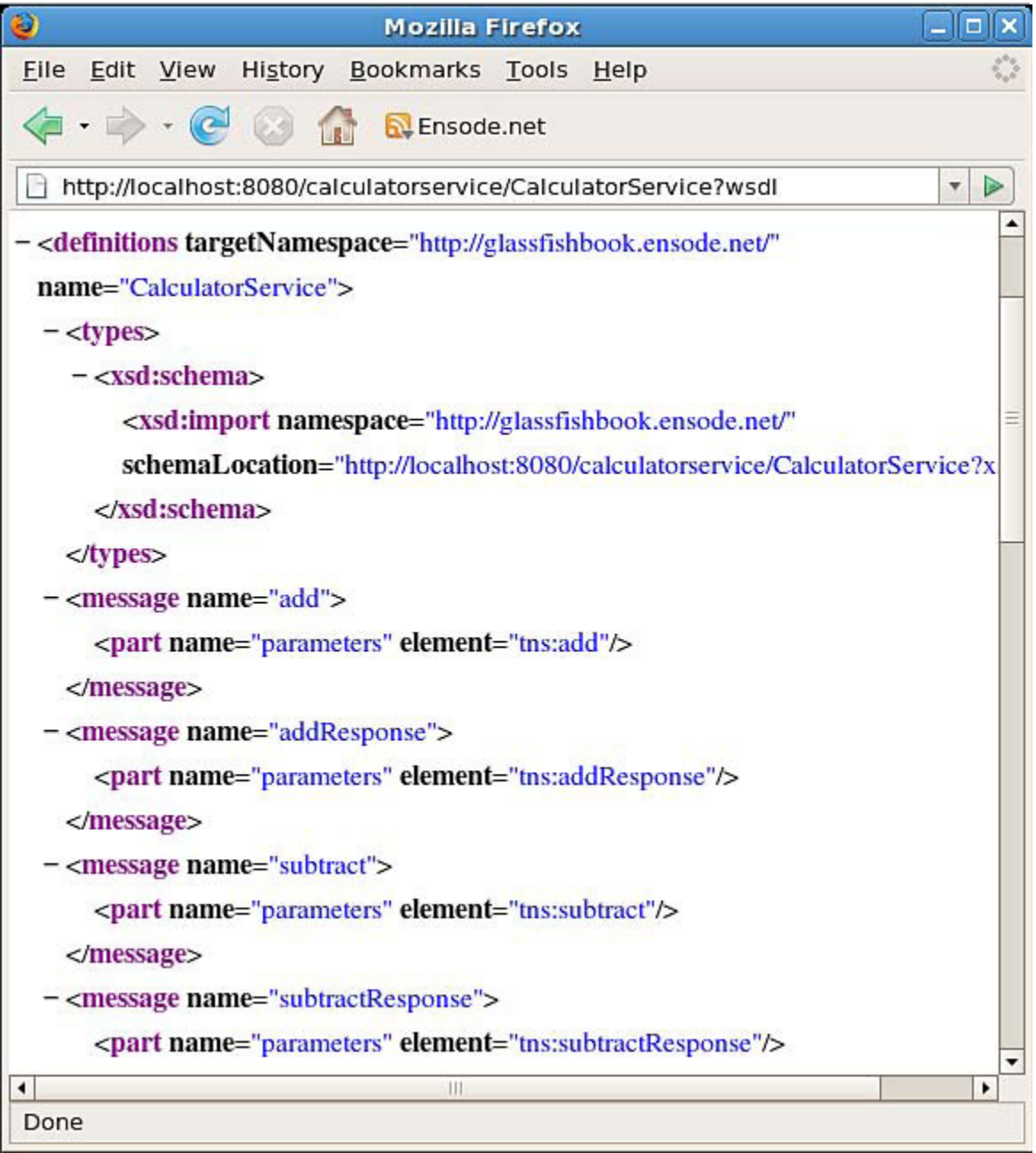


Behind the Scenes

- To exchange information between web service clients and servers, JAX-WS uses the SOAP protocol.



Viewing the WSDL File



- Web service clients need a WSDL (Web Services Definition Language) file in order to generate executable code that they can use to invoke the web service.
- When deploying web services developed using JAX-WS, a WSDL is automatically generated. We can see it, along with its URL, by clicking on the `View WSDL` link in the Application Server web console.

Developing a Web Service Client with GlassFish

- GlassFish includes a utility to generate Java code from a WSDL. The name of the utility is `wsimport`. It can be found under `[glassfish installation directory]/glassfish/bin/`. The only required argument for `wsimport` is the URL of the WSDL corresponding to the web service.

```
wsimport http://localhost:8080/calculatorservice/CalculatorService?wsdl
```

- The command will generate a number of compiled Java classes that allow client applications to access our web service:

```
Add.class
AddResponse.class
Calculator.class
CalculatorService.class
ObjectFactory.class
package-info.class
Subtract.class
SubtractResponse.class
```

These classes need to be added to the client's CLASSPATH in order for them to be accessible to the client's code.

ANT File

In addition to the command-line tool, Glassfish includes a custom ANT task to generate code from a WSDL.

Minimal example:

```
<project name="calculatorserviceclient" default="wsimport" basedir=".">
  <target name="wsimport">
    <taskdef name="wsimport" classname="com.sun.tools.ws.ant.WsImport">
     <classpath path="/opt/glassfish/lib/webservices-tools.jar"/>
     <classpath path="/opt/glassfish/lib/webservices-rt.jar"/>
     <classpath path="/opt/glassfish/lib/javaee.jar"/>
    </taskdef>
    <wsimport
  wsdl="http://localhost:8080/calculatorservice/CalculatorService?wsdl"
  />
  </target>
</project>
```

`<wsimport>` is a custom, not standard ANT target. We need to add a `<taskdef>` element to our ANT build script. We need to set its name and `classname` attributes. Additionally, we need to add the following jar files (found under the `[glassfish installation directory]/glassfish/lib` directory) to the `CLASSPATH` via nested `<classpath>` elements:

- `webservices-tools.jar`
- `webservices-rt.jar`
- `javaee.jar`

We indicate the WSDL location via its `wsdl` attribute. Once this task executes, the Java code needed to access the WS defined by the WSDL is generated.

Client Application

The `getNamePort()` method of the service class returns an instance of a class implementing an interface generated by `wsimport`. This interface is given the name of the web service class and declares all of the methods we declared to be web services.

```
import javax.xml.ws.WebServiceRef;
public class CalculatorServiceClient
 @WebServiceRef(wsdlLocation ="http://localhost:8080/calculatorservice/"
       + "CalculatorService?wsdl")
  private static CalculatorService calculatorService;
  public void calculate() {
   Calculator calculator = calculatorService.getCalculatorPort();
   System.out.println("1 + 2 = " + calculator.add(1, 2));
   System.out.println("1 - 2 = " + calculator.subtract(1, 2));
  }
  public static void main(String[] args){
    new CalculatorServiceClient().calculate();
  }
}
```

- The `@WebServiceRef` annotation injects an instance of the web service into our client application. Its `wsdlLocation` attribute contains the URL of the WSDL corresponding to the web service we are invoking.
- The web service class is an instance of a class called `CalculatorService`. This class was created when we invoked the `wsimport` utility that generates a class with the "Service" suffix.
- Use this service class to obtain an instance of the WS class. In general, the method to invoke to get an instance of our WS class follows the pattern `getXXXXXPort()`.
- Once we get an instance of our web service class, we can simply invoke its methods as with any regular Java object.

Running the client application

In order for resource injection to work in a stand-alone client, we need to execute it through the `appclient` utility. Assuming we packaged our client in a file called `calculatorserviceclient.jar`, the command to execute would be:

```
appclient -client calculatorserviceclient.jar
```

Except the primitive types, valid types that can be used in JAX-WS web service calls are listed below:

```
java.awt.Image
java.lang.Object
Java.lang.String
java.math.BigDecimal
java.math.BigInteger
java.net.URI
java.util.Calendar
java.util.Date
java.util.UUID
javax.activation.DataHandler
javax.xml.datatype.Duration
javax.xml.datatype.XMLGregorianCalendar
javax.xml.namespace.QName
javax.xml.transform.Source
```

We can also use our own custom classes as method parameters and/or return values for web service method, but member variables of our classes must be one of the listed types.

Sending Attachments to Web Services

Web service methods can send and accept file attachments.

```
import java.io.FileOutputStream;
import java.io.IOException;
import javax.activation.DataHandler;
import javax.jws.WebMethod;
import javax.jws.WebService;
@WebService
public class FileAttachment{
  @WebMethod
  public void attachFile(DataHandler dataHandler) {
    FileOutputStream fileOutputStream;
    try {
     //substitute "/tmp/attachment.gif" with a valid path, if necessary.
     fileOutputStream = new FileOutputStream("/tmp/attachment.gif");
     dataHandler.writeTo(fileOutputStream);
     fileOutputStream.flush();
     fileOutputStream.close();
    } catch (IOException e) { e.printStackTrace();
    }
  }
}
```

In order to write a web service method that receives one or more attachments, all we need to do is to add a parameter of type `javax.activation.DataHandler` for each attachment the method will receive.

Packaging and Deployment

- Just as with any standard web service, the code needs to be packaged in a WAR file and deployed.
- Once deployed, a WSDL will automatically be generated.
- We then need to execute the `wsimport` utility to generate code that our web service client can use to access the web service. As previously discussed, `wsimport` can be invoked directly from the command line, via a custom ANT target, or via a Maven 2 plugin. Once we have executed wsimport to generate code to access the web service, we can write and compile our client code package.
- A web service client that needs to send one or more attachments to the web service first obtains an instance of the web service as usual. It then creates an instance of `java.io.File` passing the location of the file to attach as its constructor's parameter.
- Once we have an instance of the `java.io.File`, containing the file we wish to attach, we then need to convert the file to a byte array, and pass this byte array to the web service method that expects an attachment.

Client Application

```
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;
import javax.xml.ws.WebServiceRef;
public class FileAttachmentServiceClient {
  @WebServiceRef(wsdlLocation = "http://localhost:8080/fileattachmentservice/" +
  "FileAttachmentService?wsdl")
  private static FileAttachmentService fileAttachmentService;
  public static void main(String[] args) {
    FileAttachment fileAttachment = fileAttachmentService.getFileAttachmentPort();
    File fileToAttach = new File("src/main/resources/logo.gif");
    byte[] fileBytes = fileToByteArray(fileToAttach);
    fileAttachment.attachFile(fileBytes);
    System.out.println("Successfully sent attachment.");
  }
  static byte[] fileToByteArray(File file) {
    byte[] fileBytes = null;
    try {
      FileInputStream fileInputStream;
      fileInputStream = new FileInputStream(file);
      FileChannel fileChannel = fileInputStream.getChannel();
      fileBytes = new byte[(int) fileChannel.size()];
      ByteBuffer byteBuffer = ByteBuffer.wrap(fileBytes);
      fileChannel.read(byteBuffer);
    } catch (IOException e) { e.printStackTrace();
    }
    return fileBytes;
  }
}
```

Bibliografie

1. Adatia R., et a., Professional EJB, Wrox Press Ltd., 2001.
2. Alur D., et a., Core J2EE Patterns, Sun Microsystems, 2001.
3. E. Armostrong, et al., The J2EE 1.4 Tutorial, novembre 2003, www.java.sun.com
4. D, R. Heffelfinger, Java EE 5 Development using GlassFish Application Server, Packt Publishing, Birmingham-Mumbai, 2007
5. Q. H. Mahmoud, Developing Web Services with Java 2 Platform, Enterprise Edition (J2EE) 1.4 Platform, www.developers.sun.com, 2004
6. Marinescu F., EJB Design Patterns, John Wiley & Sons, 2002.
7. R. Monson-Haefel, Enterprise JavaBeans, O'Reilly, ed. 2, 2001.
8. R. Monson-Haefel, J2EE™ Web Services, Addison WesleyPub, ottobre 2003
9. R. Nagappan, et al., Developing Java Web Services, Wiley Publishing, 2003.
10. P. Perrone, et al., J2EE. Developer's Handbook, Sams Publishing, 2003.
11. E. Roman, et a., Mastering Enterprise JavaBeans, 2nd ed., John Wiley, 2002.
12. B. Tate, M. Clark, B. Lee, P. Linskey, Bitter EJB, Manning Publications, 2003.
13. www.java.sun.com, The J2EE Tutorial v. 1.3, v. 1.4, on-line documentation, Sun Microsystems, 2003.

Annotations (I)

- Annotation type declarations are similar to normal interface declarations. An at-sign (@) precedes the interface keyword. Each method declaration defines an *element* of the annotation type. Method declarations must not have any parameters or a throws clause. Return types are restricted to primitives, String, Class, enums, annotations, and arrays of the preceding types. Methods can have *default values*. Here is an example annotation type declaration:

```
/** Describes the Request-For-Enhancement(RFE) that led to the presence
   of the annotated API element. */
public @interface RequestForEnhancement {
   int id();
   String synopsis();
   String engineer() default "[unassigned]";
   String date(); default "[unimplemented]";
}
```

Once an annotation type is defined, you can use it to annotate declarations. An annotation is a special kind of modifier, and can be used anywhere that other modifiers (such as `public`, `static`, or `final`) can be used. By convention, annotations precede other modifiers. Annotations consist of an at-sign (`@`) followed by an annotation type and a parenthesized list of element-value pairs. The values must be compile-time constants.

Annotations (II)

Here is a method declaration with an annotation corresponding to the annotation type declared above:

```
@RequestForEnhancement(
 id = 2868724,
 synopsis = "Enable time-travel",
 engineer = "Mr. Peabody",
 date = "4/1/3007"
)
public static void travelThroughTime(Date destination) { ... }
```

An annotation type with no elements is termed a *marker* annotation type, for example:

```
/** Indicates that the specification of the annotated API element is
  preliminary and subject to change. */
 public @interface Preliminary { }
```

It is permissible to omit the parentheses in marker annotations, as shown below:

```
@Preliminary public class TimeTravel { ... }
```

In annotations with a single element, the element should be named value, as shown below：

```
/** Associates a copyright notice with the annotated API element. */
  public @interface Copyright {
  String value();
}
```

Annotations (III)

- Many annotations replace what would otherwise have been comments in code.
- The more advanced uses of annotations include writing an *annotation processor* that can read a Java program and take actions based on its annotations. It might, for example, generate auxiliary source code, relieving the programmer of having to create boilerplate code that always follows predictable patterns. To facilitate this task, release 5.0 of the JDK includes an annotation processing tool, called apt. In release 6 of the JDK, the functionality of apt is a standard part of the Java compiler.
- To make annotation information available at runtime, the annotation type itself must be annotated with `@Retention(RetentionPolicy.RUNTIME)`, as follows:

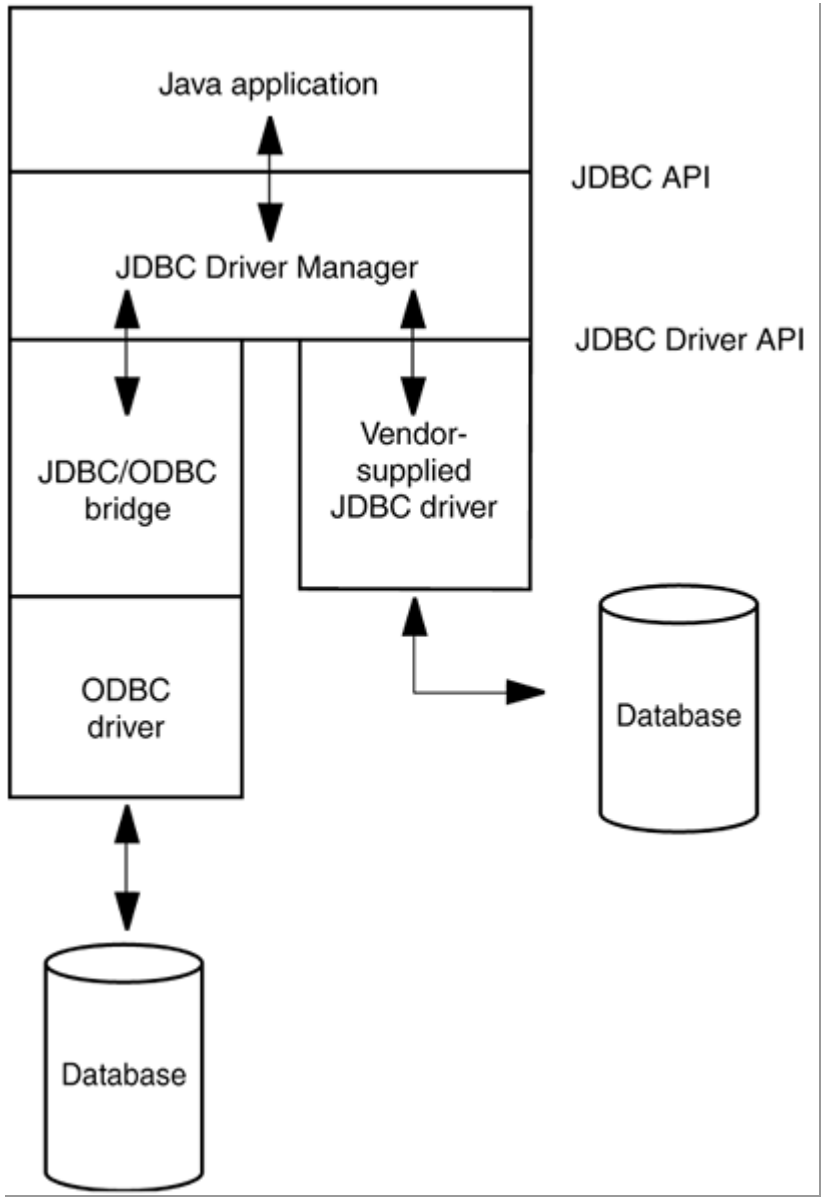
```
import java.lang.annotation.*;
@Retention(RetentionPolicy.RUNTIME)
@interface AnnotationForRuntime {
// Elements that give information
// for runtime processing
 }
```

Appendix A. Working with Relational Databases in Java

java.sql

JDBC

- JDBC = API for access to relational databases.



Four driver types

- JDBC drivers are classified into the following types:
- A type 1 driver translates JDBC to ODBC and relies on an ODBC driver to communicate with the database. Sun includes one such driver, the JDBC/ODBC bridge, with the JDK. At this point, plenty of better drivers are available, and we advise against using the JDBC/ODBC bridge.
- A type 2 driver is written partly in Java and partly in native code; it communicates with the client API of a database. When you use such a driver, you must install some platform-specific code in addition to a Java library.
- A type 3 driver is a pure Java client library that uses a database-independent protocol to communicate database requests to a server component, which then translates the requests into a database-specific protocol. This can simplify deployment since the database-dependent code is located only on the server.
- A type 4 driver is a pure Java library that translates JDBC requests directly to a database-specific protocol.

The package java.sql

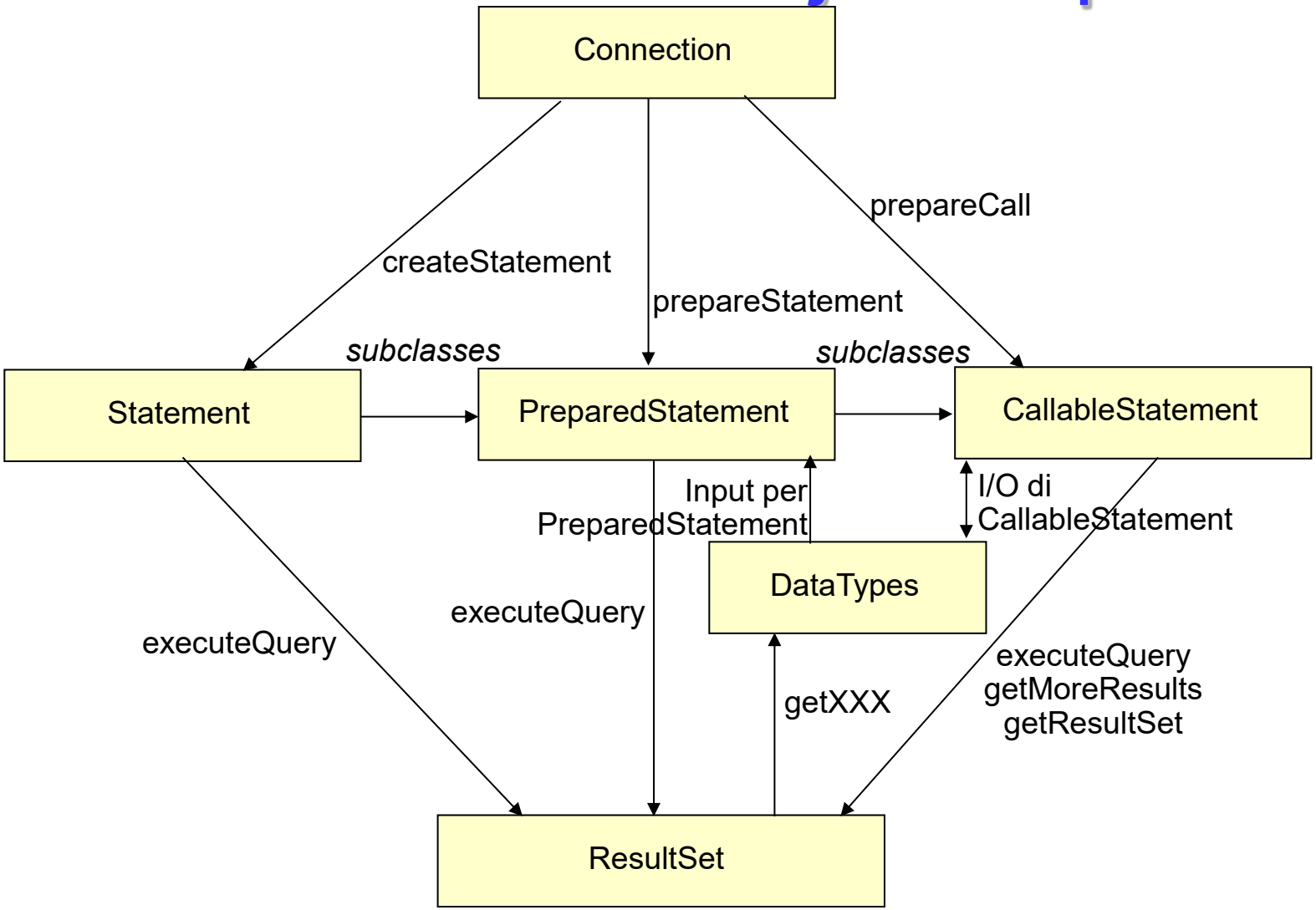
Interfaces

CallableStatement	Used for execution of the stored procedures.
Connection	An object Connection represents a communication session wit the database.
DatabaseMetaData	Provides information on the whole database.
Driver	A Java framework that enables to wiork with drivers of various databases.
PreparedStatement	A PreparedStatement object represents a pre-compiled SQL statement.
ResultSet	An object ResultSet provides access to a data table.
ResultSetMetaData	An object ResultSetMetaData can be used per find information on the properties and column types in a Resultset.
Statement	An object Statement is used to execute a SQL static statement and obtain results from this execution.

Classi

DriverManager	Provides the service of the management of drivers JDBC.
Time	Wrapper in java.util.
Timestamp	Wrapper in java.util.
Types	Defines the constants used for the types SQL.

Relationships between classes and interfaces in java.sql



Il package javax.sql

javax.sql.ConnectionEvent
javax.sql.ConnectionEventListener
javax.sql.ConnectionPoolDataSource
javax.sql.DataSource
javax.sql.PooledConnection
javax.sql.RowSet

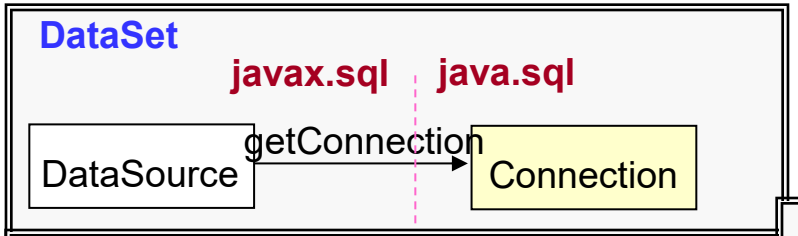
javax.sql.RowSetEvent
javax.sql.RowSetInternal
javax.sql.RowSetListener
javax.sql.RowSetMetaData
javax.sql.RowSetReader
javax.sql.RowSetWriter
javax.sql.XAConnection
javax.sql.XADataSource

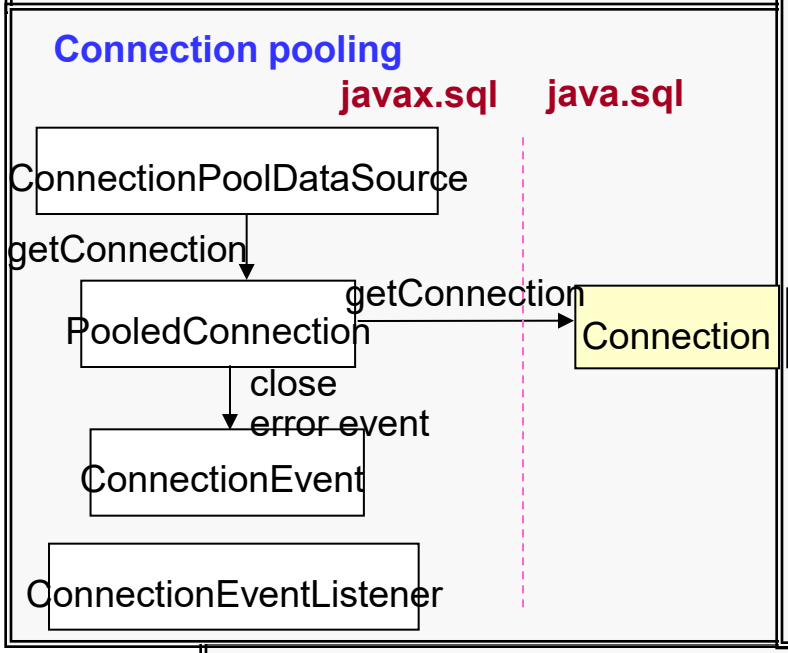
Il pacchetto javax.sql definisce due estensioni importanti dell'interfaccia `DataSource` per poter affrontare applicazioni complesse:

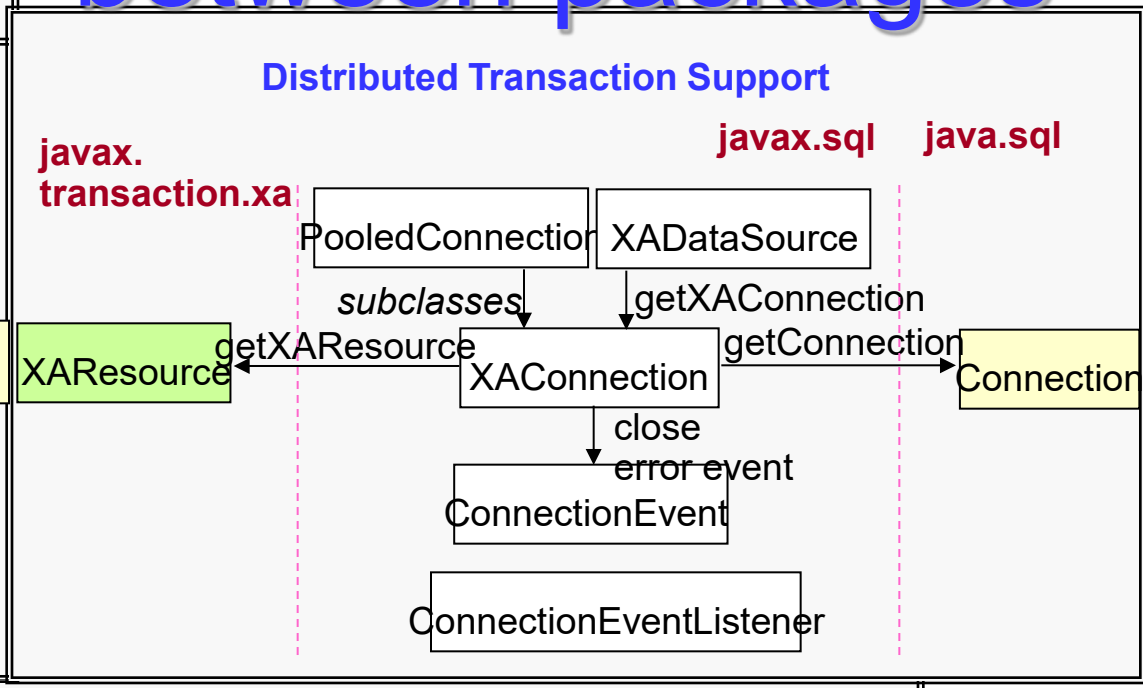
`ConnectionPoolDataSource` – supporta il caching e il riutilizzo delle connessioni fisiche mogliorando così la performance e la scalabilità.

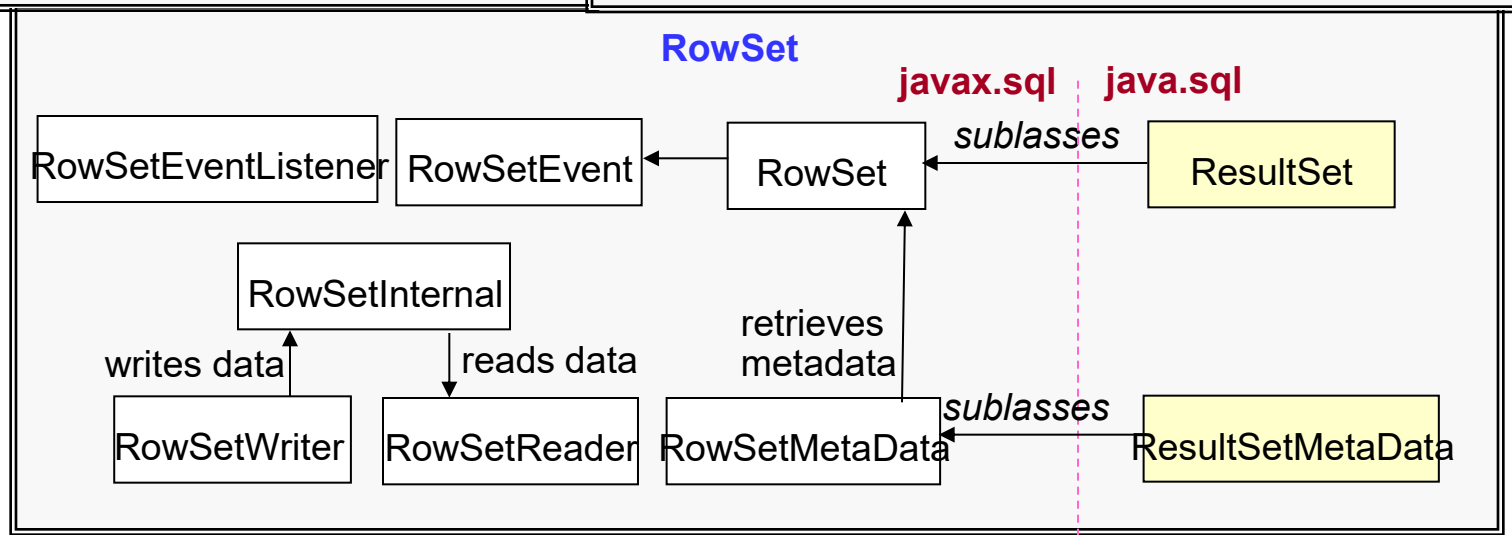
XADataSource (nel pacchetto `javax.transaction.xa`) - fornisce connessioni che possono partecipare ad una transazione distribuita.

Relationships between packages









SQL-Structured Query Language

SQL has two command languages

- DML (Data Manipulation Lang.)
- DDL (Data Definition Lang.)

Commands DML

- SELECT columns
  - FROM tables
  - [WHERE condition]
- INSERT INTO table
  - VALUES (expression, …)
- DELETE
- UPDATE
  - SET column = expression
  - [WHERE condition]

Commands DDL

- CREATE TABLE
  - (column with type)
- DROP TABLE tabel
- ALTER TABLE

```
SELECT COF_NAME, PRICE
   FROM COFFEES
   WHERE PRICE > 60.50

INSERT INTO COFFEES
   VALUES ('Colombian', 101, 7.99, 0, 0)


UPDATE COFFEES
   SET TOTAL = TOTAL + 100
   WHERE COF_NAME = 'Colombian'

CREATE TABLE COFFEES
   (COF_NAME VARCHAR(32), SUP_ID INTEGER,
   PRICE FLOAT, SALES INTEGER, TOTAL INTEGER)
```

Data Types

SQL Type	JDBC Type	Description
INTEGER o INT	int	integer 32-bit
SMALLINT	short	integer 16-bit
NUMERIC(m,n)	java.sql.Numeric	decimal number, fixed point,
DECIMAL(m,n)		m digits and
DEC(m,n)		n digits after the point
FLOAT(n)	double	float. p. number, n digits accuracy
REAL	float	float. p. number, 32-bit
DOUBLE	double	float. p. number, 64-bit
CHARACTER(n)	String	n-length character string
CHAR(n)		
VARCHAR(n)	String	variable length strings, max. n
DATE	java.sql.Date	date, implementation dependent
TIME	java.sql.Time	time, implementation dependent
TIMESTAMP	java.sql.Timestamp	date/time, implem. dependent

URL

Database URL = Specifica la collocazione in internet della fonte di dati che è il database.
Sintax:

```
jdbc:subprotocol:other stuff
jdbc:mckoi://localhost/
jdbc:postgresql:COREJAVA
```

For the `odbc` protocol:

```
jdbc:odbc://min.gov:7000/catalogue;PWD=Ciampi
```

Drivers Registering

- Find out the names of the JDBC driver classes used by your vendor. Examples:

```
org.postgresql.Driver
com.mckoi.JDBCDriver
```

1. Find the library in which the driver is located, such as `pg74jdbc3.jar` or `mkjdbc.jar`. Use one of the following three mechanisms:
2. Launch your database programs with `-classpath command-line argument`.
3. Modify the CLASSPATH environment variable.
4. Copy the database library into the `jre/lib/ext` directory.

The DriverManager class selects database drivers and creates a new database connection. However, before the driver manager can activate a driver, the driver must be registered.

You can specify the property with a command-line argument, such as

```
java -Djdbc.drivers=org.postgresql.Driver MyProg
```

Or your application can set the system property with a call such as

```
System.setProperty("jdbc.drivers", "org.postgresql.Driver");
```

Or you can manually register a driver by loading its class. For example,

```
Class.forName("org.postgresql.Driver"); // force registration of driver
```

Making a Connection

You can specify the property with a command-line argument, such as

```
String url = "jdbc:postgresql:MYDB";
String username = "dbuser";
String password = "secret";
Connection conn = DriverManager.getConnection(url,username, password);
```

It is convenient to use a properties file to specify the URL, user name, and password in addition to the database driver. A typical properties file has the following contents:

```
jdbc.drivers=org.postgresql.Driver
jdbc.url=jdbc:postgresql:COREJAVA
jdbc.username=dbuser jdbc.password=secret
```

Here is the code for reading a properties file and opening the database connection.

```
Properties props = new Properties();
FileInputStream in = new FileInputStream("database.properties");
props.load(in);
in.close();
String drivers = props.getProperty("jdbc.drivers");
if (drivers != null)
   System.setProperty("jdbc.drivers", drivers);
String url = props.getProperty("jdbc.url");
String username = props.getProperty("jdbc.username");
String password = props.getProperty("jdbc.password");
return DriverManager.getConnection(url, username, password);
```

Interfaccia Connection

`void close()` immediately closes the current connection and the JDBC resources that it created.

`Statement createStatement()` creates a Statement object that can be used to execute SQL queries and updates without parameters.

```
DatabaseMetaData getMetaData()
CallableStatement prepareCall(String sql)
PreparedStatement prepareStatement(String sql)
void commit()
void rollback() throws SQLException
void setAutoCommit(boolean enableAutoCommit) throws SQLException
void setTransactionIsolation(int level) throws SQLException
```

Creation of tables in JDBC



Statement

Statement = Interface implemented by objects that encapsulate a SQL statement.

`ResultSet executeQuery(String sqlQuery)` executes the SQL statement given in the string and returns a ResultSet object to view the query result.

`int executeUpdate(String sqlStatement)` executes the SQL INSERT, UPDATE, or DELETE statement specified by the string. Also executes Data Definition Language (DDL) statements such as CREATE TABLE. Returns the number of records affected, or -1 for a statement without an update count.

`boolean execute(String sqlStatement)` executes the SQL statement specified by the string. Returns true if the statement returns a result set, false otherwise. Use the geTResultSet or getUpdateCount method to obtain the statement outcome.

`int getUpdateCount()` returns the number of records affected by the preceding update statement, or -1 if the preceding statement was a statement without an update count. Call this method only once per executed statement.

`ResultSet getResultSet()` returns the result set of the preceding query statement, or null if the preceding statement did not have a result set. Call this method only once per executed statement.

`void close()` closes this statement object and its associated result set.

CreaDB (I)

```
import java.net.*;
import java.sql.*;
import java.io.*;
import java.util.*;
class CreaDB {
  public static void main (String args[]) {
    try {

  // 1. Si esegue una connessione al database
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      String url = "jdbc:odbc:GestLibri";
      String utente = "Pippo";
      String password = "pizza";
      Connection con = DriverManager.getConnection(url,utente, password);
      Statement stmt = con.createStatement();

  // 2. Se non trova il nome del file di input nella linea di comando, lo richiede.
      String nomeFile = "";
      if (args.length > 0)
        nomeFile = args[0];
      else {
        System.out.println("Digita nome file: ");
        nomeFile = new BufferedReader(new InputStreamReader(System.in)).readLine();
      }
  // 3. Estrae il nome del file e della tabella togliendo l'estensione .txt.
      int i = 0;
      while (i < nomeFile.length() && (Character.isLowerCase(nomeFile.charAt(i))
                  || Character.isUpperCase(nomeFile.charAt(i))))
        i++;
      String nomeTabella = nomeFile.substring(0, i);
```

```
// 4. Legge i nomi delle colonne con readLine()
    BufferedReader in = new BufferedReader(new FileReader(nomeFile));
    String[] nomiColonne = readLine(in);
// 5. Legge i tipi delle colonne.
    String[] nomiTipi = readLine(in);
// 6. Utilizza il metodo creaTabella() per creare una stringa:
//            CREATE TABLE Nome (Col1 Tipo1, Col2 Tipo2, . . . )
    createTable(stmt, nomeTabella, nomiColonne, nomiTipi);
// 7. Per ciascuna riga del file di input si esegue un'istruzione INSERT,
// utilizzando il metodo inserIn().
    boolean pronto = false;
    while (!pronto) {
      String[] valori = readLine(in);
      if (valori.length == 0) pronto = true;
      else inserIn(stmt, nomeTabella, nomiTipi, valori);
    }
// 8. Chiama il metodo mostraTabella() per visualizzare come è stata creata e
// inizializzata la tabella.
    mostraTabella(stmt, nomeTabella, nomiColonne.length);
    stmt.close();
    con.close();
 } catch (SQLException ex) {
      System.out.println ("SQLException:");
      while (ex != null) {
         System.out.println("SQLState:"+ex.getSQLState());
         System.out.println("Messagio:"+ex.getMessage());
         System.out.println("Venditore:"+ex.getErrorCode());
         ex = ex.getNextException();
         System.out.println ("");
      }
    } catch (java.lang.Exception ex) {
      System.out.println("Eccezione:"+ex);
      ex.printStackTrace ();
    }
  }
}
```

CreaDB (II)

CreaDB (III)

```
  private static String[] readLine(BufferedReader in) throws IOException {
   // divide le linee lette in token con StringTokenizer e "|".
     String line = in.readLine();
     Vector risultato = new Vector();
     if (line != null) {
        StringTokenizer t = new StringTokenizer(line, "|");
        while (t.hasMoreTokens())
          risultato.addElement(t.nextToken().trim());
     }
     String[] retval = new String[risultato.size()];
     risultato.copyInto(retval);
     return retval;
  }

  private static void createTable(Statement stmt, String nomeTabella,
        String[] nomiColonne, String[] nomiTipi) throws SQLException {
         String comando = "CREATE TABLE "+nomeTabella+"(\n";
         String primary = "";
         for (int i = 0; i < nomiColonne.length; i++) {
            if (i > 0) comando += ",\n";
            String nomeColonna = nomiColonne[i];
            if (nomeColonna.charAt(0) == '*') {
               if (primary.length() > 0) primary += ", ";
               nomeColonna = nomeColonna.substring(1, nomeColonna.length());
               primary += nomeColonna;
            }
            comando += nomeColonna + " " + nomiTipi[i];
         }
         if (primary.length() > 0)
             comando += "\nPRIMARY KEY (" + primary + ")";
         comando += ")\n";
         stmt.executeUpdate(comando);
  }
}
```

CreaDB (IV)

```
private static void inserIn(Statement stmt, String nomeTabella,
             String[] nomiTipi,String[] valori) throws SQLException {
   String comando="INSERT INTO "+nomeTabella+" VALUES (";
   for (int i = 0; i < nomiTipi.length; i++) {
      if (i > 0) comando += ", ";
      String columnType = nomiTipi[i].toUpperCase();
      String valore = "";
      if (i < valori.length) valore = valori[i];
      if (columnType.startsWith("CHAR") || columnType.startsWith("VARCHAR")) {
         int da = 0;
         int a = 0;
         comando += "'";
         while ((a = valore.indexOf('\'', da)) >= 0) {
             comando += valore.substring(da, a) + "''";
             da = a + 1;
         }
         comando += valore.substring(da) + "'";
      }
      else comando += valore;
   }
   comando += ")";
   stmt.executeUpdate(comando);
}
```

CreaDB (V)

```
  private static void mostraTabella(Statement stmt,
      String nomeTabella, int numCol) throws SQLException {
      String query = "SELECT * FROM " + nomeTabella;
      ResultSet rs = stmt.executeQuery(query);
      while (rs.next()) {
         for (int i = 1; i <= numCol; i++) {
           if (i > 1) System.out.print("|");
             System.out.print(rs.getString(i));
         }
         System.out.println("");
      }
      rs.close();
   }
}
```

Data Processing with JDBC



ResultSet

`boolean next()` makes the current row in the result set move forward by one. Returns false after the last row. Note that you must call this method to advance to the first row.

`Xxx getXxx(int columnNumber)`

`Xxx getXxx(String columnName)` (Xxx is a type such as int, double, String, Date, etc.) return the value of the column with the given column number or name, converted to the specified type. Not all type conversions are legal. See documentation for details.

`int findColumn(String columnName)` gives the column index associated with a column name.

`void close()` immediately closes the current result set.

Connection, Statement, ResultSet

```
Connection conn = . . .;
try
{
   Statement stat = conn.createStatement();
   ResultSet result = stat.executeQuery(queryString);
   process query result
}
finally
{
   conn.close();
}
```

Processing the results

3. Loop for results:

```
while (rs.next()) {
      . . . . analyze a row . . . .
}
```

4. For a row it is useful to know the column contents:

```
String isbn = rs.getString(1);
float prezzo = rs.getDouble("Price");
```

PreparedStatement

PreparedStatement = A SQL statement already compiled, only the parameters must be provided .

Exemple:

```
String editorQuery =
      "SELECT Books.Price, Books.Title " +
      "FROM Books, Publishers " +
      "WHERE Books_Pub_ID=Publishers.Pub_ID " + "AND Publisher.Name=?";


publisherQueryStmt=con.prepareStatement(publisherQuery);

      editorQueryStmt.setString(1, publisher);
```

PreparedStatement Interface

`void clearParameters()`

Elimina tutti i parametri correntemente attivi per l'istruzione. Altrimenti, i parametri rimangono attivi tragli usi del Statement.

`boolean execute()`

Alcune PreparedStatement restituiscono risultati multipli. Il metodo elabora queste istruzioni complesse come le forme più semplici elaborate con executeQuery and executeUpdate.

`ResultSet executeQuery()`

E' eseguita una query SQL pre-compilata ed è rilasciato il suo ResultSet.

`int executeUpdate()`

Esegue un'istruzione SQL INSERT, UPDATE o DELETE.

`ResultSetMetaData getMetaData()`

Fornisce il numero, tipi e le proprietà di una delle colonne del ResultSet.

`void setXxxl(int parameterIndex, xxx x)`

Imposta un parametro ad un valore di tipo xxx. Xxx può essere BigDecimal, boolean, byte, byte[], Date, double, float, int, long, Object, short, String, Time, Timestamp.

PreparedStatement

```
String editorQuery =
      "SELECT Libri.Prezzo, Libri.Titolo " +
      "FROM Libri, Editori " +
      "WHERE Libri_Edi_ID = Editori.Edi_ID " + "AND Editori.Nome = ?";
PreparedStatement  editorQueryStmt= con.prepareStatement(editorQuery);
editorQueryStmt.setString(1, editor);
```

La query viene eseguita con il metodo *executeQuery()*:

```
      ResultSet rs = editorQueryStmt.executeQuery();
```

L'insieme di risultati viene poi elaborato come qualsiasi ResultatSet:

```
TextArea risultato = new TextArea();
risultato.setText("");
while (rs.next())
      risultato.appendText(rs.getString(1)+" | "+rs.getString(2)+"\n");
rs.close();
```

Graphical Interface

```
import java.net.*;
import java.sql.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
public class InterrogaDB extends Frame implements ActionListener {
    public InterrogaDB() {
       setLayout(new GridBagLayout());
       GridBagConstraints gbc = new GridBagConstraints();
       autori = new Choice();
       autori.addItem("Qualsiasi");
       editori = new Choice();
       editori.addItem("Qualsiasi");
       risultato = new TextArea(4, 50);
       risultato.setEditable(false);
      try {
                Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
                String url = "jdbc:odbc:GestLibri";
                String user = "Pippo";
                String password = "pizza";
                con=DriverManager.getConnection(url,user,password);
                stmt = con.createStatement();
                String query = "SELECT Nome FROM Autori";
                ResultSet rs = stmt.executeQuery(query);
                while (rs.next())
                        autori.addItem(rs.getString(1));
                query = "SELECT Nome FROM Editori";
                rs = stmt.executeQuery(query);
                while (rs.next())
                        editori.addItem(rs.getString(1));
          } catch(Exception e){
                risultato.setText("Errore " + e);
          }
```

Graphical Interface

```
  gbc.fill = GridBagConstraints.NONE;
   gbc.weightx = 100;
   gbc.weighty = 100;
   add(autori, gbc, 0, 0, 2, 1);
   add(editori, gbc, 2, 0, 2, 1);
   gbc.fill = GridBagConstraints.NONE;
   Button queryButton = new Button("Query");
   add(queryButton, gbc, 0, 1, 1, 1);
   queryButton.addActionListener(this);
   Button cambiaButton = new Button("Cambia prezzo");
   add(cambiaButton, gbc, 2, 1, 1, 1);
   cambiaButton.addActionListener(this);
   prezzoCambia = new TextField(8);
   prezzoCambia.setText("-5000");
   add(prezzoCambia, gbc, 3, 1, 1, 1);
   gbc.fill = GridBagConstraints.BOTH;
   add(risultato, gbc, 0, 2, 4, 1);
}

private void add(Component c, GridBagConstraints gbc,int x,int y,int w,int h) {
    gbc.gridx = x;
    gbc.gridy = y;
    gbc.gridwidth = w;
    gbc.gridheight = h;
    add(c, gbc);
}
```

“GestLibri”

3. Quando l’utente preme il bottone “Query”, si eseguirà una delle 4 query impegnate a portare i dati nell’interfaccia.
Si utilizza un join dalla tabella Libri alla tabella LibriAutori e poi alla tabella Autori per comparare con il nome scelto dal utente.

```
SELECT Libri.Prezzo, Libri.Titolo
FROM libri, Editori, LibriAutori, Autori
WHERE Libri.Edit_Id = Editori.Edit_Id
    AND Editori.Nome = ?
    AND Libri.ISBN = LibriAutori.ISBN
    AND LibriAutori.Autore = Autori.Autore
    AND Autori.Nome = ?
```

4. I risultati della query sono visualizzati nella casella risultati.

Queries

```
public void actionPerformed(ActionEvent evt) {
   String arg = evt.getActionCommand();
   if (arg.equals("Query")) {
      ResultSet rs = null;
      try {
               String autore = autori.getSelectedItem();
               String editore = editori.getSelectedItem();
               if (!autore.equals("Qualsiasi") && !editore.equals("Qualsiasi")){
                   if (autoreEditoreQueryStmt == null) {
                        String autoreEditoreQuery =
                            "SELECT Libri.Prezzo, Libri.Titolo " +
                            "FROM Libri, LibriAutori, Autori, Editori " +
                            "WHERE Autori.Aut_Id = LibriAutori.Aut_Id AND " +
                                "LibriAutori.ISBN = Libri.ISBN AND " +
                                "Libri.Edit_Id = Editori.Edit_Id AND " +
                                "Autori.Nome = ? AND " +
                                "Editori.Nome = ?";
                        autoreEditoreQueryStmt =
                                 con.prepareStatement(autoreEditoreQuery);
                   }
                   autoreEditoreQueryStmt.setString(1, autore);
                   autoreEditoreQueryStmt.setString(2,editore);
                   rs = autoreEditoreQueryStmt.executeQuery();
               }
```

Queries

```
 else if (!autore.equals("Qualsiasi") && editore.equals("Qualsiasi")) {
     if (autoreQueryStmt == null) {
             String autoreQuery =
                     "SELECT Libri.Prezzo, Libri.Titolo " +
                     "FROM Libri, LibriAutori, Autori " +
                     "WHERE Autori.Aut_Id = LibriAutori.Aut_Id AND " +
                          "LibriAutori.ISBN = Libri.ISBN AND " +
                          "Autori.Nome = ?";
             autoreQueryStmt = con.prepareStatement(autoreQuery);
     }
     autoreQueryStmt.setString(1, autore);
     rs = autoreQueryStmt.executeQuery();
 }
 else if (autore.equals("Qualsiasi") && !editore.equals("Qualsiasi")) {
     if (editoreQueryStmt == null) {
             String editoreQuery =
                     "SELECT Libri.Prezzo, Libri.Titolo " +
                     "FROM Libri, Editori " +
                     "WHERE Libri.Edit_Id = Editori.Edit_Id AND " +
                             "Editori.Nome = ?";
             editoreQueryStmt=con.prepareStatement(editoreQuery);
     }
     editoreQueryStmt.setString(1, editore);
     rs = editoreQueryStmt.executeQuery();
 }
```

Queries

```
        else {
             if (allQueryStmt == null) {
                      String allQuery =
                               "SELECT Libri.Prezzo, Libri.Titolo FROM Libri";
                      allQueryStmt=con.prepareStatement(allQuery);
             }
             rs = allQueryStmt.executeQuery();
        }
        risultato.setText("");
        while (rs.next())
             risultato.append(rs.getString(1) + " | " + rs.getString(2) + "\n");
        rs.close();
    }  catch(Exception e){
             risultato.setText("Error " + e);
    }
}
```

Table Updating

5.

```
UPDATE Libri
SET Prezzo = Prezzo + prezzoCambia
WHERE Libri.Edit_Id =
        (SELECT Edit_Id
        FROM Editors
        WHERE Nome = editore)
```

Price Change: Update

```
    else if (arg.equals("Cambia prezzo")) {
            String editore = editori.getSelectedItem();
            if (editore.equals("Qualsiasi"))
                    risultato.setText("Scusi, non posso!");
         else try {
              String updateStatement =
                 "UPDATE Libri " +
                 "SET Prezzo = Prezzo + " + prezzoCambia.getText() +
                 " WHERE Libri.Edit_Id = " +
                   "(SELECT Edit_Id ” +
                    "FROM Editori ” +
                    ”WHERE Nome = '" +  editore + "')";
              int r = stmt.executeUpdate(updateStatement);
              risultato.setText(r + " record aggiornati.");
            } catch(Exception e) {
                risultato.setText("Errore " + e);
            }
    }
}
```

Resources Releasing

6.

```
public class InterrogaDB extends Frame implements ActionListener {
        public InterrogaDB() {
          try {
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            String url = "jdbc:odbc:GestLibri";
            String user = "Pippo";
            String password = "pizza";
            con=DriverManager.getConnection(url,user,password);
            stmt = con.createStatement();
                 . . . . .
                   } catch(Exception e){
       . . . . . .
       public void dispose() {
         try {
           stmt.close();
           con.close();
         } catch(SQLException e) {}
      }
       . . . .
       Connection con;
       Statement stmt;
}
```

Main Program

```
  public void dispose() {
        try {
                stmt.close();
                con.close();
        } catch(SQLException e) {}
  }
  public static void main (String args[]) {
        Frame f = new QueryDB();
        f.setSize(400, 300);
        f.show();
  }
  private Choice autori;
  private Choice editori;
  private TextField prezzoCambia;
  private TextArea risultato;
  private Connection con;
  private Statement stmt;
  private PreparedStatement autoreQueryStmt;
  private PreparedStatement autoreEditoreQueryStmt;
  private PreparedStatement editoreQueryStmt;
  private PreparedStatement allQueryStmt;
}
```

Advanced JDBC



Advanced Connection and ResultSet (JDBC 2.0)

Connection

```
Statement createStatement(int scorr, int concorrenza);
scorr = TYPE_FORWARD_ONLY (simile con JDBC 1.0)
        TYPE_SCROLL_INSENSITIVE (scorrevoli, ma non si aggiorna dal DB)
        TYPE_SCROLL_SENSITIVE (scorrevoli, si aggiorna dal DB)
concorrenza=   CONCUR_READ_ONLY (simile con JDBC 1.0)
               CONCUR_UPDATABLE (si può modificare)
```

Esempio:

```
Statement stmt = con.createStatement(TYPE_SCROLL_SENSITIVE,
                                     CONCUR_UPDATABLE);
```

ResultSet

```
previous()                      first()                 last()
beforeFirst()                   afterLast()             moveToInsertRow()
moveToCurrentRow()              absolute(int pos)       relative(int pos)
int getRow()                    isFirst()               isLast()
isBeforeFirst()                 isAfterLast()
updateXxx(String col,Xxx val)   cancelRowUpdates()      insertRow()
deleteRow()                     refreshRow()
```

Esempio:

```
ResultSet rs = stmt("SELECT * FROM INDIRIZZI");
rs.last();
rs.updateString("NOME", "MARCO");
```

Working with Advanced ResultSets

- Risultati scorrevoli:

```
Statement stmt = con.createStatement(scorr, concorrenza);
PreparedStatement stmt = con.prepareStatement(comando, tipo, concorrenza);
```

- Per lavorare cu insiemi di risultati scorrevoli (`TYPE_SCROLL_INSENSITIVE` e `TYPE_SCROLL_SENSITIVE`) si utilizza:

if (rs.previous()) . . .	per scorrere in dietro
rs.relative(n);	per spostare il cursore in avanti e indietro con n righe
rs.absolute(n);	per spostare il cursore alla n-a riga
int i = rs.getRow();	per ricavare il numero di riga corrente
first(), last(), sull'ultima	consentono di spostare il cursore sulla prima riga,
beforeFirst(), afterLast()	prima della prima riga e dopo l'ultima riga
isFirst(), isLast(), posizioni	verificano se il cursore si trova in una delle 4
isBeforefirst(), isAfterlast()	

- Per lavorare con insiemi di risultati aggiornabili (`CONCUR_UPDATABLE`) si utilizza:

rs.updateXxx(*col*, *val*)	per modificare la colonna *col*, di tipo Xxx, al valore *val*
rs.updateRow() corrente	per allineare il database ai nuovi valori della riga
rs.cancelRowUpdates()	per annullare gli aggiornamenti apportati alla riga
rs.moveToInsertRow()	per spostare il cursore sulla *riga di inserimento*
rs.insertRow()	per trasmettere la nuova riga al database
rs.deleteRow()	per rimuovere la riga sia dal insieme che dal database

Cursor Moving in ResultSets

Creare un recordset scorrevole:

```
Statement stmt = con.createStatement( ResultSet.TYPE_SCROLL_INSENSITIVE,
   ResultSet.CONCUR_READ_ONLY);
ResultSet srs = stmt.executeQuery("SELECT COF_NAME, PRICE FROM COFFEES");

 if (rs.getType()== ResultSet.TYPE_SCROLL_INSENSITIVE) {}
```

Scorrere tra record in avanti:

```
Statement stmt = con.createStatement( ResultSet.TYPE_SCROLL_INSENSITIVE,
   ResultSet.CONCUR_READ_ONLY);
ResultSet srs = stmt.executeQuery("SELECT COF_NAME, PRICE FROM COFFEES");
while (srs.next()) {
 String name = srs.getString("COF_NAME");
 float price = srs.getFloat("PRICE");  // double price=srs.getDouble("PRICE");
 System.out.println(name + " " + price);
}
```

Scorrere tra record in dietro:

```
Statement stmt = con.createStatement( ResultSet.TYPE_SCROLL_INSENSITIVE,
   ResultSet.CONCUR_READ_ONLY);
ResultSet srs = stmt.executeQuery( "SELECT COF_NAME, PRICE FROM COFFEES");
srs.afterLast();
while (srs.previous()) {
 String name = srs.getString("COF_NAME");
 float price = srs.getFloat("PRICE");
 System.out.println(name + " " + price);
}
```

Spostare il cursore in una posizione data:

```
srs.absolute(4);
```

Updatable ResultSet

Creare un ResulSet aggiornabile

```
Connection con = DriverManager.getConnection( "jdbc:mySubprotocol:mySubName");
Statement stmt = con.createStatement( ResultSet.TYPE_SCROLL_SENSITIVE,
   ResultSet.CONCUR_UPDATABLE);
ResultSet uprs = stmt.executeQuery("SELECT COF_NAME,PRICE FROM COFFEES");
```

Verificare il tipo di ResultSet

```
if (uprs.getConcurrency()== ResultSet.CONCUR_UPDATABLE) {}
```

Lavorare con ResultSet aggiornabili

```
uprs.last();
uprs.updateFloat("PRICE", 10.99f);

uprs.last();
uprs.updateFloat("PRICE", 10.99f);
uprs.updateRow();

uprs.last();
uprs.updateFloat("PRICE", 10.99f);
 . . .
uprs.cancelRowUpdates();
```

Inserimento di un nuovo record

```
uprs.moveToInsertRow();
uprs.updateString("COF_NAME", "Kona");
uprs.updateInt("SUP_ID", 150);
uprs.updateFloat("PRICE", 10.99f);
uprs.updateInt("SALES", 0);
uprs.updateInt("TOTAL", 0);
uprs.insertRow();
```

Bibliografia

1. E. Armstrong, e.a., The J2EE 1.4 Tutorial, Sun Microsystems, 2003.
2. M. Hall, Core Servlets and Java Server Pages, Sun Microsystems Press, 2000.
3. A. Patzer, JSP Examples and Best Practices, Apress, 2002.
4. P. J. Perrone, e.a., J2EE. Developer's Handbook, Sams Publishing, 2003.
5. Documentazione Tomcat Web Server, Jakarta Apache Project, http://jakarta.apache.org/tomcat