

# Introduction to web programming

## Contents

Copyright .....	5
Introduction .....	6
Web Architectures.....	6
HTTP .....	7
Http Request .....	7
Format .....	7
Request Methods .....	8
Http Responses.....	9
Format .....	9
Http Headers .....	10
Http Secure and Cryptography .....	11
Symmetric key mechanism.....	12
Asymmetric key mechanism.....	12
Markup Languages .....	13
Types of Markup Languages .....	14
1. Presentational Markup .....	14
2. Procedural Markup.....	14
3. Semantic/Descriptive markup .....	14
Smgl.....	15
XML .....	15
Xml derivative languages.....	16
XML Grammar .....	17
Parser .....	19
XML Validation .....	20
DTD Definition .....	20
DTD Elements and Attlist.....	21
XML Schema .....	21
HTML.....	25
Structure of an HTML doc.....	26

Head .....	26
Tags .....	28
Basic Formatting: Blocks, Inline .....	29
Image and Figure .....	30
File paths .....	31
Hyperlinks.....	31
Tables .....	31
Forms .....	32
CSS.....	35
How to add style.....	35
Lengths .....	36
Absolute units.....	36
Relative lengths .....	36
Colors .....	37
Fonts.....	38
Font choice .....	38
Font referencing .....	39
Font attributes.....	39
Background .....	40
Box Model .....	40
Formatting boxes.....	41
Lists.....	41
Selectors .....	42
Basic .....	42
Combinator .....	42
Pseudo-Class and Pseudo-elements .....	43
Attribute selectors.....	44
Cascading and positioning .....	44
Origin and importance.....	44
Specificity .....	45
Cascading order .....	46
Positioning.....	46
Server side programming .....	47
First solution: CGI .....	47
Example .....	48
Parameters passing .....	50

Second solution: PHP .....	51
PHP variable scope .....	52
Third Solution: Java servlets .....	53
Servlet lifecycle.....	53
Interaction with IntelliJ Idea .....	55
Code Example n.1: Basic Servlet .....	56
Code Example n.2: Basic Servlet, Date and Time .....	57
HttpServletRequest .....	59
Include vs Forward .....	60
Code Example n.3: Structured code and Parameters, Calculator .....	62
Track the servlet execution .....	66
Serialization .....	67
Code Example n.4: Serialization, Hit Counter .....	68
Cookies .....	71
Code Example n.5: Cookies, Calculator with cookies.....	73
Cookie Regulation.....	80
Session.....	81
Concurrency .....	83
Sharing information between servlets: Servlet Context .....	84
Associating events with session objects .....	85
Jsp.....	87
JSP Elements.....	88
Example 1: Date and Time .....	89
JSP Processing and Lifecycle .....	90
Request, Response and Sessions in JSP .....	91
Best Practices .....	91
Java Beans .....	92
Code Example: JavaBean .....	93
MVC pattern .....	96
Javascript.....	98
Basics.....	99
Events.....	99
Javascript and HTML.....	99
Syntax .....	99
Operators .....	99
Data Types .....	100

DOM .....	100
JavaScript HTML DOM Document* .....	102
Finding HTML elements .....	102
Changing HTML elements .....	102
Adding/Deleting .....	102
Input, Output.....	103
Strings.....	103
“+” Operator.....	104
Functions .....	105
Hoisting .....	106
Special functions properties* .....	106
Array methods.....	107
Variables.....	108
Objects .....	109
Object() method and defineProperty .....	109
Object constructors .....	110
Prototypes .....	110
How to inspect objects .....	111
Object inheritance .....	112
Predefined objects .....	113
Classes .....	114
Arrays .....	116

## Copyright

Copyright © 2023 Matteo Bregola. All rights reserved. This work is based on materials from Introduction to Web Programming taught by G. Varni at Unitn, as well as materials from the website mentioned in the note below<sup>1</sup>, and is protected by copyright laws. It is provided for personal, educational, and non-commercial use only. Any unauthorized reproduction or distribution of this work, whether in whole or in part, is strictly prohibited without the express written consent of the copyright holder. For permission to use or reproduce this work, please contact Matteo Bregola at [matteo.bregola@studenti.unitn.it](mailto:matteo.bregola@studenti.unitn.it)

<sup>1</sup>. <https://www.w3schools.com/> , <https://www.tutorialspoint.com/> , <https://developer.mozilla.org/> , <https://docs.oracle.com/en/>, <https://gdpr.eu/cookies/> .



# UNIVERSITÀ DI TRENTO

## Introduction

Web= usage of Internet through the http protocol

The resource of the WWW are:

- described via the **Hypertext Markup Language (HTML)**
- transferred via the **Hypertext Transfer Protocol (HTTP)**
- accessed by users by a software app called a **web browser**
- published by a software app called a **web server**

A **Uniform Resource Identifier (URI)** is a compact sequence of characters that identifies an abstract or physical resource

URI= scheme: [//authority]path[?query][#fragment]. authority=[userinfo@]host[\_port]

*The elements inside the parenthesis are optional.*

A **Uniform Resource Locator (URL)** refers to the subset of URI that identify resources via a representation of their primary access mechanism. *Can be taught like a postal code.*

**Uniform Resource Name (URN)** refers to the subset of Uri that are required to **remain globally unique and persistent** even when the resource ceases to exist or becomes unavailable.

An URN can be associate to many URL, both URL and URN are URI

**MIME TYPE** = Multipurpose Internet Mail Extension

MEDIA\_TYPE/SUBTYPE

text -> text/plain, text/html, text/richtext ...

image -> image/jpeg, image/png, image/svg+xml...

audio -> audio/basic, audio/ogg, audio/x-wav...

video -> video/mp4, video/ogg...

application -> application/x-apple-diskimage... ..

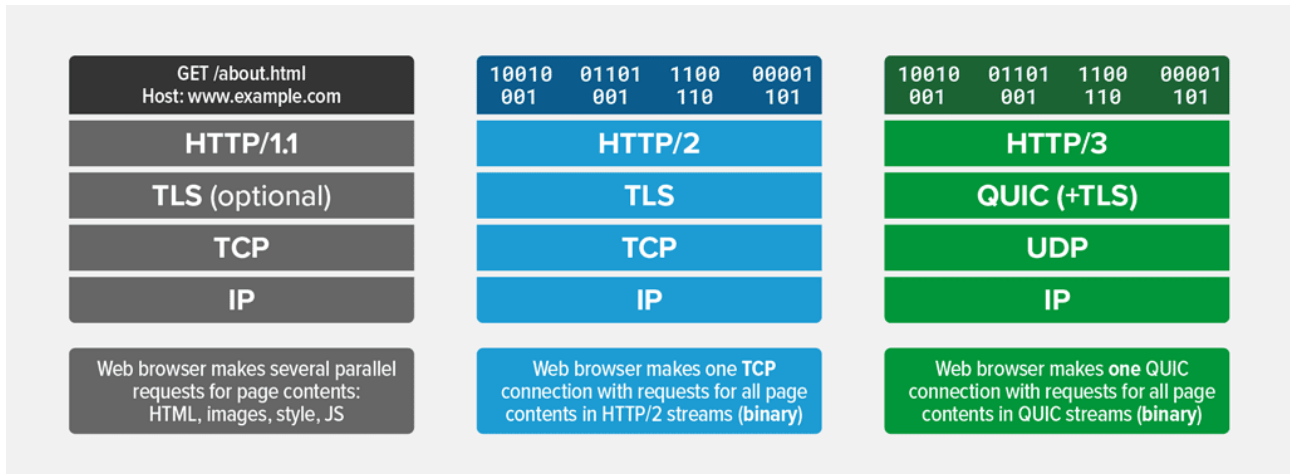
multipart

## Web Architectures

- 1) **Static Pages:** Static Html pages as response to the GET requests
- 2) **Dynamic Pages:** a process or thread is “called” after the request so beyond the static html page there are some dynamic elements
- 3) **Dynamic Pages with DB:** the programs interact to the database
- 4) **Smart Browser:** part of the code executes on the client browser
- 5) **Plug in:** optional code added to the browser (like Flash)
- 6) **Responsive:** there is a reload only on the “new” parts, usage of json, xml...

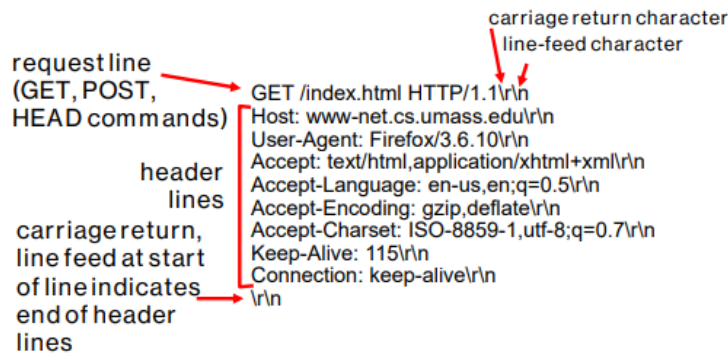
HTTP

HTTP/0.9	HTTP/1.0	HTTP/1.1
telnet-friendly protocol	Browser-friendly protocol	Browser-friendly protocol
No HTTP headers. error codes, versioning.	Header fields / metadata (HTTP version number, status code, content type)	Performance optimizations and feature enhancements persistent and pipelined connections, virtual hosting...)
Methods supported: GET	Methods supported: GET, HEAD, POST	Methods supported: GET. HEAD. POST. PUT. DELETE, TRACE. OPTIONS
Response type; hypertext	Response: not limited to hypertext (Content-Type header enable transmission Of files Other than plain HTML files)	Response: as V1.0
Connection nature: terminated immediately after the response	Connection nature: terminated immediately after the response	Connection nature: long-lived

Http RequestFormat

- **Request line:** Method, URL, version
- **Header lines (optional) :** Header field name, Value. Divided into:
  - General Headers, Request Headers, Entity Headers
- **Blank line**
- **Message body (optional) .** Called message if there is a payload, otherwise entity

- HTTP request message:
  - ASCII (human-readable format)



The diagram shows an HTTP request message in ASCII format. Red arrows point from labels to specific parts of the message:

- request line (GET, POST, HEAD commands)** points to the first line: `GET /index.html HTTP/1.1\r\n`
- header lines** points to the block of lines starting with `Host:` and ending with `Connection: keep-alive\r\n`.
- carriage return, line feed at start of line indicates end of header lines** points to the `\r\n` at the end of the header block.
- carriage return character line-feed character** points to the `\r\n` at the end of the request line.

The full message is:

```
GET /index.html HTTP/1.1\r\n
Host: www-net.cs.umass.edu\r\n
User-Agent: Firefox/3.6.10\r\n
Accept: text/html,application/xhtml+xml\r\n
Accept-Language: en-us,en;q=0.5\r\n
Accept-Encoding: gzip,deflate\r\n
Accept-Charset: ISO-8859-1,utf-8;q=0.7\r\n
Keep-Alive: 115\r\n
Connection: keep-alive\r\n
\r\n
```

## Request Methods

**Idempotent** = A Method is **idempotent** if an identical request can be made once or several times in a row with the same effect while leaving the server in the same state

**Safe** = A method is safe if it doesn't alter the state of the server. A method is safe if it leads to a read-only operation

METHOD	DESCRIPTION	IDEMPOTENT	SAFE
Head	-Retrieves only the Headers associated with a resource but not the entity itself - Usage: Protocol analysis, diagnostic	✓	✓
Get	-Request a resource from the server -Support passing of query string arguments -It <b>doesn't</b> contain a body	✓	✓
Post	-Allow passing or submitting data in entity rather than URL - <b>Append</b> resources to the existing ones -Arguments not displayed in the URL	X	X
Put	-Creates a new resource or replaces a		X



	representation of the target resource with the request payload.	✓	
Trace	-Ask the server to return a diagnostic method for Tunneling other protocols through HTTP	✓	✓
Options	-Ask the server to return the list of request methods it supports	✓	✓
Delete	-Deletes a resource	✓	X
Connect	-Starts two-way communications with the requested resource. It can be used to open a tunnel.	X	X

## Http Responses

### Format

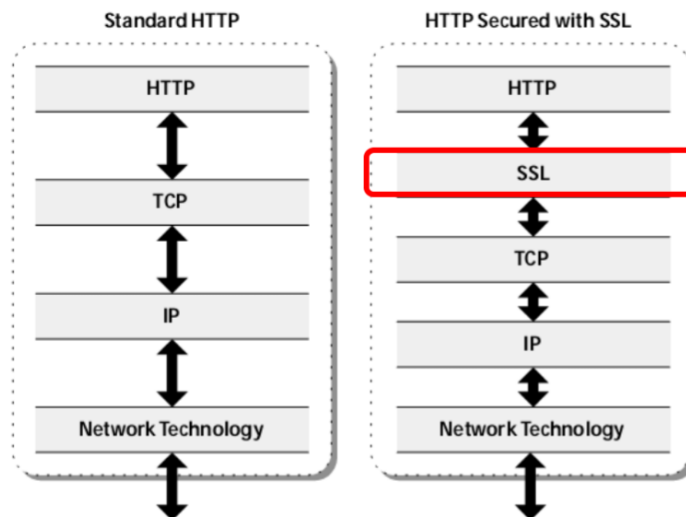
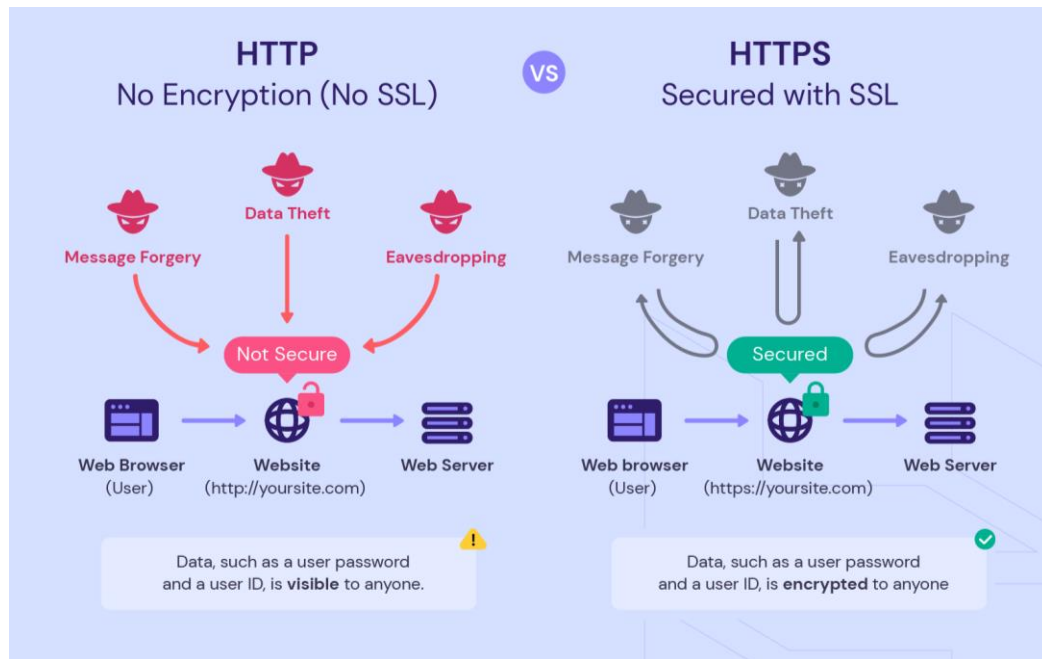
- **Status line:** Protocol, Status code, status message
- **Message Headers (optional):** Date, Server, Last-Modified, Content-Length...
- **Empty line**
- **Message Body (optional)**

HTTP Status Codes				javaconceptoftoday.com	
1xx : Informational Purpose		4xx : Client Errors		5xx : Server Errors	
100	Continue	400	Bad Request	500	Internal Server Error
101	Switching Protocols	401	Unauthorized	501	Not Implemented
102	Processing	402	Payment Required	502	Bad Gateway
103	Early Hints	403	Forbidden	503	Service Unavailable
2xx : Success		404	Not Found	504	Gateway Timeout
200	Ok	405	Method Not Allowed	505	HTTP Version Not Supported
201	Created	406	Not Acceptable	507	Insufficient Storage
202	Accepted	407	Proxy Authentication Is Required	508	Loop Detected
203	Non-Authoritative Information	408	Request Time Out	510	Not Extended
204	No Content	409	Conflict	511	Network Authentication Required
205	Reset Content	410	Gone		
206	Partial Content	411	Length Required		
207	Multi Status	412	Precondition Failed		
208	Already Reported	413	Payload Too Large		
226	IM Used	414	URI Too Long		
3xx : Redirection		415	Unsupported Media Type		
300	Multiple Choices	416	Range Not Satisfiable		
301	Moved Permanently	417	Expectation Failed		
302	Found	421	Misdirect Request		
303	See Other	422	Unprocessable Entity		
304	Not Modified	423	Locked		
305	Use Proxy	424	Failed Dependency		
306	No Longer Used	425	Too Early		
307	Temporary Redirect	426	Upgrade Required		
308	Moved Permanently	428	Precondition Required		
		429	Too Many Requests		
		431	Request Header Fields Too Large		
		451	Unavailable For Legal Reasons		

Http Headers

<b>Connection</b>	Lets clients and servers manage connection state <i>Eg:</i> <i>Connection: Keep -Alive</i> <i>Connection: close</i>	<b><u>General Headers</u></b>
<b>Date</b>	When the message was crated  <i>Eg: Date: Sat, 31-May-03 15:00:00 GMT</i>	
<b>Via</b>	Shows proxies that handled message Proxy= Intermediate server that answer in behalf of the central one, to be used a conditional get is required, if the content in the proxy is updated than send it otherwise it asks to the original server. <i>Eg: Via: 1.1 www.myproxy.com (Squid/1.4)</i>	
<b>Cache Control</b>	Among the most complex headers, enables caching directive <i>Eg: Cache-Control: no cache</i>	
<b>Host</b>	The hostname (and optionally port) of server to which request is being sent	<b><u>Request Headers</u></b>
<b>Refer</b>	The URL of the resource from which the current request Uri Came -> used for business aspects	
<b>User-Agent</b>	The name of the requesting application, used in browser sensing	
<b>Accept</b>	Inform servers of client's capabilities and preferences <i>Eg: Enables content negotiation</i> <i>Accept: image/gif, image/jpeg; q=0.5</i> <i>Accept -&gt; variants for Language, Encoding, Charset</i>	
<b>Cookie</b>	How clients pass cookies back to the servers that set them <i>Eg: Cookie: id=23432;level=3</i>	
<b>Server</b>	The server's name and version. (can be problematic for security reasons) <i>Eg: Server: Microsoft-IIS/5.0</i>	<b><u>Response Headers</u></b>
<b>Set-Cookie</b>	How a server sets a cookie on a client <i>Eg:</i> <i>Set-Cookie: id=234; path=/shop; expires=Sat, 31-May-03 15:00:00 GMT; secure</i>	
<b>Allow</b>	Lists the request methods that can be used on the entity <i>Eg: Allow: GET, HEAD, POST</i>	<b><u>Entity Headers</u></b>
<b>Location</b>	Gives the alternate or new location of the entity, used with 3xx response codes (redirects) <i>Eg: Location: http://www.something.edu/it</i>	
<b>Content-Encoding</b>	Specifies encoding performed on the body of the response, used with HTTP compression. Corresponds to Accept-Encoding request Header <i>Eg: Content-Encoding: gzip</i>	
<b>Content-Length</b>	The size of the entity body in bytes	
<b>Content-Type</b>	Specifies Media (MIME) type of the entity body	

## Http Secure and Cryptography



Disadvantages:

- Slower than http
- Need to setup

**Cryptography** = practice and study of techniques for secure communication in the presence of adversarial behavior

## Symmetric key mechanism

Message: "Good Morning"

Encryption algorithm: *shift left by the key  $n$ ,  $n=w$*

Encoded message: *key=2 "lqqf Oqtpkpi"*

## Asymmetric key mechanism

Receiver → sends an open lock

Sender → sends the message with the lock (lock=public key)

Receiver → receives the public key and use the private key to unlock it

SSL Session:

- Uses **asymmetric encryption** to privately share the **session key**.
- Use **symmetric encryption** to encrypt **data**.

## Markup Languages

**Markup Language** = system for annotating a document in a way that is syntactically distinguishable from the text.

It specifies a code for formatting, both the layout and style within a text file. The code used to specify the formatting are called tags.

In most case is human readable.

Is not a programming language (the instructions do not become a machine instruction but are interpreted).

Usage:

- Storing, managing, delivering, structuring, publishing data.
- Reusage of the same information in different formats, contexts, classes of users

**Tag** = a sequence of characters that begins with "<" and ends with ">".

Three types:



- Start tag: Eg: <book>
- End tag: Eg: </book>
- Empty element tag: Eg: <br/>

**Element** = a sequence of characters that begins with a start-tag and ends with a matching end-tag, or consists only of a empty-element tag. The characters between the start and the end tag, if any, are the element's content and can contain other elements called **children**.

**Attribute** = a name-value pair existing in a start-tag or in a empty element tag.

Eg:

```
<data>
  <NETWORK>
    <IP>172.150.1.101</IP>
  </NETWORK>
  <LECTURE id="27">
    <COURSE_NAME>Web Programming</COURSE_NAME>
    <LECTURE_NAME>Introduction to XML</LECTURE_NAME>
    <TIME>5225.00</TIME>
  </LECTURE>
</data>
```

 = attribute
 = element

## Types of Markup Languages

### 1. Presentational Markup

Used by traditional word-processing systems: **binary codes embedded within document text** that produce the “what you see is what you get” effect.

The markup is usually hidden from the human users.

*Eg: Rich Text Format (RTF), language internally used by word*

```
{\rtf1\ansi\deff0
{\fonttbl {\f0 Times New Roman;}}
{\colortbl ;\red255\green0\blue0;}
This is an example of an RTF document. {\b Bold text.} {\i
Italic text.} {\ul Underlined text.}
{\cf1 This text is red.}
{\pard This is a new paragraph.}
}
```

### 2. Procedural Markup

They **provide instructions for programs to process the text** (images..). The processor runs through the text from beginning to end, following the instruction as encountered. Text with such markup is often edited with the markup visible and directly manipulated by the author.

*Eg: Markdown, Postscript.*

```
/Times-Roman findfont
12 scalefont
setfont
newpath
100 200 moveto
(Example 3) show
```

### 3. Semantic/Descriptive markup

They are used to label parts of the document for what they are (*eg: define the title of a document*) rather than how they should be processed. The **objective is to organize the document with a logical structure**. Descriptive markup encourages author to write in a way that describes the material conceptually, rather than visually.

*Eg: LaTeX, HTML, XML*

## Smgl

**SMGL** is an ISO standard which provides a formal notation for the definition of **generalized** markup languages. It's not a language in itself. Rather it is a metalanguage that is used to define other languages.

A SMGL document is the combination of three parts (generally divided in files):

- The content of the document (words, pictures...). (This is the part that the author wants to expose to the client);
- The grammar that defines the accepted syntax (DTD = Data Type Definition), used to define the elements and organize it;
- The stylesheet that establishes how the content that conforms to the grammar has to be rendered on the output device. (it's the modality in which we visualize it).

## XML

eXtensible Markup Language is a group of technologies created in 1998 from the World Wide Web Consortium for web development.

Xml is used for:

- Having data in an easy format
- Exchanging data between applications
- Accessing to databases
- Changing the format of data
- Writing config files
- Web services

Xml doesn't do anything by itself, it is just information wrapped in tags.

### **Advantages:**

- Ease:
  - Simple rules
  - Human readable documents
- Compatibility:
  - Platform independent
  - Many tools available for writing and validating docs
- Power:
  - Definition of really complex documents
  - Managed as a text doc when It travels on the web

## Xml derivative languages

**DTD** = Document Type Definition. It is a way to **describe the structure and content of an XML** (Extensible Markup Language) document. A DTD **specifies the rules** that define how elements and attributes can be used in an XML document. It also **defines the order of elements, their relationship with each other, and the allowed values for attributes**. In simpler terms, a DTD is like a blueprint or a set of **guidelines** that developers can use to ensure that their **XML documents are valid and structured correctly**. By adhering to a DTD, XML documents can be validated, and errors can be caught before the document is processed.

**XSL** = Extensible Stylesheet Language, which is a language **used for transforming XML** (Extensible Markup Language) documents into other formats, such as HTML, PDF, or plain text. XSL consists of two parts: XSLT (XSL Transformations) and XSL-FO (XSL Formatting Objects). XSLT is a language used for transforming XML documents into other XML or non-XML formats, using templates and rules to manipulate the structure and content of the input XML. **XSLT allows developers to transform XML documents into a wide range of formats, such as HTML, XHTML, WML** (Wireless Markup Language), SVG (Scalable Vector Graphics), and more. **XSL-FO is a language used for formatting XML documents for print or display**, defining layout and design aspects such as page size, margins, fonts, and colors. XSL-FO documents can be transformed into various output formats, such as PDF, PostScript, or RTF (Rich Text Format). XSL plays an important role in web development, as it allows developers to create dynamic and customized web pages, convert data from one format to another, and automate various tasks related to XML processing.

XPath, XLink, and XPointer are three related technologies used in XML (Extensible Markup Language) documents to navigate, link, and identify specific parts of the document.

**XPath** is a language used to **select and navigate specific parts of an XML document**, such as elements and attributes. XPath expressions are used to **identify nodes in an XML tree structure**, allowing developers to extract data or perform transformations on the document.

**XLink** (XML Linking Language) is a language used **to create hyperlinks between XML documents or specific parts of XML documents**. XLink allows developers to link to resources outside of the current document, as well as link to specific parts of a document using XPointer.

**XPointer** is a language used to **identify specific parts of an XML document**, such as elements or attributes, within the context of an XLink. XPointer can be used in conjunction with XLink to create more specific and precise links within and between XML documents.

Together, XPath, XLink, and XPointer provide developers with powerful tools for navigating, linking, and identifying specific parts of XML documents, making it easier to create dynamic and interactive content for the web.

**XQL** = XML Query Language is a **query language** designed for searching and extracting data from XML (Extensible Markup Language) documents. XQL was developed as an alternative to XSLT (XSL Transformations) and other XML processing technologies, providing a more flexible and expressive way to query XML data.

and many others...



## XML Grammar

XML does not use predefined tags but are user-defined.

Provides a grammar to:

- define tags;
- define rules for the tags;
- allowed attributes;
- containment rules.

The grammar is defined in a:

- **DTD file;**
- **XML-Schema file;**
- not defined at all.

All **XML documents must be well formed**, that is they must comply **several requirements**:

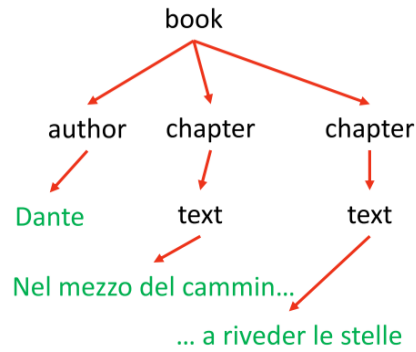
- **case sensitive**
- there **must be a single top-level tag** called the **root tag** that contains all the tags in the document
- **each tag must have a closing tag**, or if empty may provide the abbreviated form
- tag must be **appropriately nested**: the end tags must follow the reverse order of their respective start tags.  
*Eg: <b><i>This text is bold and italic</i></b>.*
- **Attribute values must always be enclosed in a single or double quotes**. You can surround the value with single quotes if the attribute contains a double quote.  
*Eg: <note id="322423"> </note>.*
- **Empty element can contain attributes**  
*Eg: <bool author = "eckel" price="\$43.21" />*
- **No markup character** are allowed, but it can be used different notations  
*Eg: "<" = &lt; or &#60 (ascii) or &#x3C (hexadecimal)*

There are no rules about when to use attributes or when to use elements in XML but attributes cannot contain multiple values and tree structure.

An XML document is an information unit that can be seen in two ways:

- 1) **Linear sequence of characters** that contain characters data and markup
- 2) An abstract data structure that is a **tree of nodes**

```
<book>
  <author> Dante </author>
  <chapter id='1'>
    <text> Nel mezzo del
cammin...</text>
  </chapter>
```



### XML Logical Structure:

- **Prolog** = XML declaration, optional but if present must be the first element

Eg: `<<?xml version='1.0' encoding='utf-8'?>`

- Optional DTD Declaration
- Optional comments and Processing Instructions (= PIs, pass information to parameters, usually for style sheet)
- The root element's start tag
- All other elements, comments and PIs
- The root element closing tag

### Namespaces:

- used to solve conflicts

<pre>&lt;table&gt;   &lt;name&gt; Vangsta &lt;/name&gt;   &lt;width&gt; 80 &lt;/width&gt;   &lt;length&gt;180 &lt;/length&gt;   &lt;height&gt; 120 &lt;/height&gt; &lt;/table&gt;  &lt;table&gt;   &lt;name&gt; Pippo &lt;/name&gt;   &lt;surname&gt; de Pippis &lt;/surname&gt;   &lt;height&gt; 180 &lt;/height&gt; &lt;/table&gt;</pre>	<pre>&lt;root xmlns:h="https://www.ikea.com/furniture" xmlns:f="http://www.gym4all.it/tnt/people"&gt;    &lt;h:table&gt;     &lt;h:name&gt; Vangsta &lt;/h:name&gt;     &lt;h:width&gt; 80 &lt;/h:width&gt;     &lt;h:length&gt;180 &lt;/h:length&gt;     &lt;h:height&gt; 120 &lt;/h:height&gt;   &lt;/h:table&gt;    &lt;f:table&gt;     &lt;f:name&gt; Pippo &lt;/f:name&gt;     &lt;f:surname&gt; de Pippis &lt;/f:surname&gt;     &lt;f:height&gt; 120 &lt;/f:height&gt;   &lt;/f:table&gt;  &lt;/root&gt;</pre>
--	---

## Parser

**Parser** = is a software tool that preprocesses an XML document in some fashion, handing the results over to an application program.

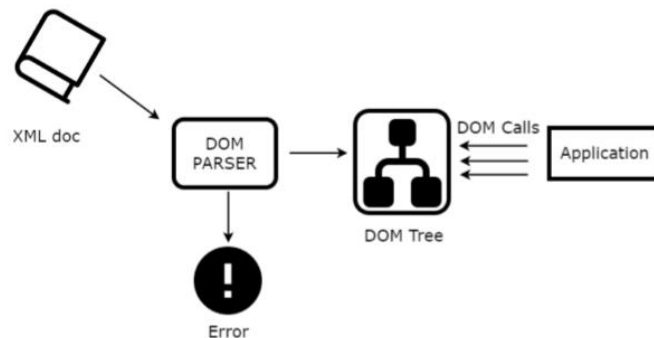
Purpose:

- **Check if the XML file is well-formed**
- **Permit to other files to access to elements**

Types:

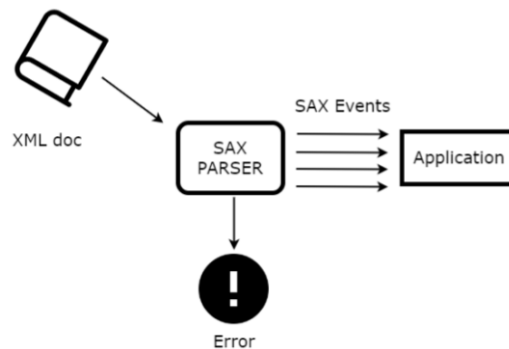
- **Tree based API** =

The element arrives to the parser → Checks if it's well formed, if it is then it creates a **DOM Tree**. The applications use **Dom Calls** to access to the tree. NB: a **disadvantage** is the **space** occupied by the tree



- **Event Driven/Based Parser** =

Uses callbacks. Reads consecutively until it reaches the requested tag. Usually there are two versions push (like Sax) and pull.

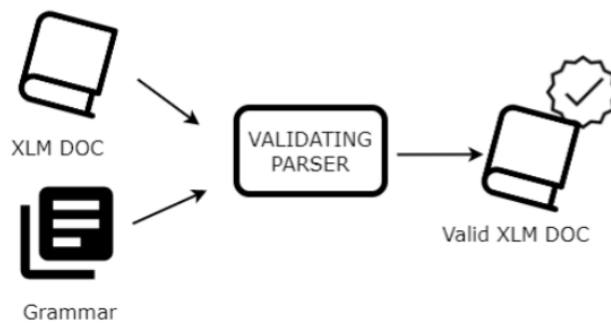


## XML Validation

A well-formed XML document conforms to the rules (the grammar) of either a DTD or a XML-Schema is a valid XML document .

Validity is not a requirement of XML:

- An invalid XML document can be a perfectly good and useful XML document.
- A non well-formed document cannot be valid, and is not an XML document.



If the document is well formed is validated with respect to the grammar.

## DTD Definition

A DTD can be:

- **Internal** to a document:

Eg: `< !DOCTYPE rubrica [Content of DTD] >`

- **External** to a document:

Eg:

`<!DOCTYPE root_tag SYSTEM "FileDTD.dtd" >`

`< !DOCTYPE root_tag PUBLIC "FPI" "URL" >` #usually defined by a public entity

Examples (this case external DTD Definition):

### XML FILE:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE book SYSTEM "Book.dtd">
<book>
<title> 1984 </title>
<author> George Orwell </author>
<editor> de Pippis </editor>
</book>
```



### DTD File:

```
<!DOCTYPE book [
<!ELEMENT book (title, author, editor)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT author(#PCDATA)>
<!ELEMENT editor (#PCDATA)>
]>
```

## DTD Elements and Attlist

### Element:

```
<!ELEMENT element-name category>  
or  
<!ELEMENT element-name (element-content)>
```

### Subtags:

?	Zero or one
+	One or more
*	Zero or more
,	Sequence
	Or

Eg: `<!ELEMENT book (title, author+, editor*)>`

### Attribute:

```
<!ATTLIST element-name attribute-name attribute-type attribute-value>
```

<b>CDATA</b>	Character data
<b>ID</b>	Unique key
<b>IDREF</b>	Key of another element
<b>(... ..)</b>	Set of eligible values

<b>#IMPLIED</b>	optional, no default
<b>#fixed</b>	optional, default supplied if present must match default
<b>#REQUIRED</b>	must be provided

Eg: `<!ATTLIST author nationality (Italian|other) #REQUIRED>`

## XML Schema

XML schemas (also referred as XSD) are born from the need to create a grammar for XML that was written in XML, in fact **DTDs are not written in XML**.

Each XML document defined through a DTD can be also defined through a XML Schema, the opposite is not necessarily true.

Eg, how to reference a XML schema:

```
<note  
xmlns="https://www.w3schools.com"  
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
xsi:schemaLocation="https://www.w3schools.com/xml note.xsd">
```

### XSD Structure :

- **<schema>** element -> **root element** in every XML schema

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

it may contain some attributes:

`<xs:schema`

`xmlns:xs="http://www.w3.org/2001/XMLSchema"` → indicates that the elements and data types used in the schema come from the website namespace and should be prefixed with xs: .

`targetNamespace="https://www.w3schools.com"` → indicates that the elements defined by this schema (note, to, from, heading, body..) come from the "https://www.w3schools.com" namespace.

`xmlns="https://www.w3schools.com"` → indicates that the default namespace is "https://www.w3schools.com".

`elementFormDefault="qualified">` → indicates that any elements used by the XML instance document which were declared in this schema must be namespace qualified.

### XSD Simple Elements:

- A **simple element** is an XML element that contains only text (Boolean, string, date, custom..). It **cannot contain any other elements** or attributes.
- **Syntax:** `<xs:element name="xxx" type="yyy"/>` where xxx is the name of the element and yyy is the data type of the element.
- Built-in data types: xs:string, xs:decimal, xs:integer, xs:Boolean, xs:date, xs:time
- Eg:

#### XML FILE:

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

#### XML SCHEMA :

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

- They can have **default values** automatically assigned when no value is specified.  
Eg: `<xs:element name="color" type="xs:string" default="red"/>`
- they can have **fixed value** that cannot be changed .  
Eg: `<xs:element name="color" type="xs:string" fixed="red"/>`

### XSD Complex Elements:

- A **complex element** is an XML element that contains other elements and/or attributes
- There are four kinds of complex elements (each of them may contain attributes as well):
  - Empty elements
  - Elements that contain only other elements
  - Elements that contain only text
  - Elements that contain both other elements and text

Examples of the four types:

1. `<product pid="1345"/>`
2. `<employee>`  
    `<firstname>John</firstname>`  
    `<lastname>Smith</lastname>`  
    `</employee>`
3. `<food type="dessert">Ice cream</food>`
4. `<description>`  
    It happened on `<date lang="norwegian">03.03.99</date>` ....  
    `</description>`

- How to define them:

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

→ only the "employee" element can use the specified complex type.

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

→ several elements can refer to the same complex type

Indicators:

- Used to control how elements are to be used in documents.
- Different types:
  - Order:
    - All → any order
    - Choice → one element or another
    - Sequence → elements follow the order
  - Occurrence
  - Group

*Exercise:*

Design the DTD for managing an address book

- The address book contains at least one contact
- Each contact consists of :
  - A name
  - Zero or more mail addresses
  - Zero or more phone numbers
  - Zero or more email addresses
- Each name is composed of:
  - A first name
  - Possible other names (second, third)
  - Last name
- Each address is composed of:
  - Possible a street
  - Surely a zip code
  - Surely a town

Solution:

rubrica.dtd

```
<! ELEMENT heading (countct+)>
<! ELEMENT contact (fullname,
address*,
phone*, email*, nationality?) >
<! ELEMENT fullname (name,
secondname*, lastname)>
<! ELEMENT nome (#PCDATA)>
<! ELEMENT secondname (#PCDATA)>
<! ELEMENT lastname (#PCDATA)>
<! ELEMENT address (street?, zip
code, country)>
<! ELEMENT street (#PCDATA)>
<! ELEMENT zip (#PCDATA)>
<! ELEMENT nation (#PCDATA)>
<! ELEMENT telephone (#PCDATA)>
<! ELEMENT email (#PCDATA)>
<! ELEMENT nationality (#PCDATA)>
```

rubrica.xml

```
<?xml version="1.0" encoding="ISO-8859-
1" ?>
<!DOCTYPE Address_list SYSTEM
"address_book.dtd">
<rubrica>
<contact>
<fullname>
<name>Donald</nome>
<secondname>Fauntleroy</secondoname>
<lastname>Duck</lastname>
</fullname>
<address>
<street> Platani 17</street>
<zip>00001</zip>
<nation>USA</nation>
</address>
<email>donald.duck@paperopoli.com</email
>
<nationality>Statunitense</nationality>
</contact>
<contact>
<fullname>
<name>Peter</name>
<lastname>Wooden Leg</lastname>
</fullname>
</contact>
```



## HTML

**HTML** = Hypertext Markup Language → standard markup language for creating web pages

**Hypertext** → Displayed on electronic devices with references (**hyperlinks**) to other text that the reader can immediately access. Hyperlinks permits to interconnect hypertexts.

The Html standards were initially developed by the World Wide Web Consortium W3C.

Version	Date	Notes
HTML 2.0	1995	First official version
HTML 3.2	1997	
HTML 4.0 (4.01)	1999	Add table, CSS, JS
XHTML 1.0	2000	XML version of HTML 4
DHTML		HTML 4 +JS + CSS + DOM
XHTML 2.0		Abandoned in 2009
HTML 5 (5.2)	2014 (2017)	(2008 first draft, 2021 last version)
XHTML 5.1		XML version of HTML 5

Note: HTML5 has no DTD.

### Two main principles:

- 1) **Graceful degradation of presentation** : The HTML documents will be displayed regardless of errors.

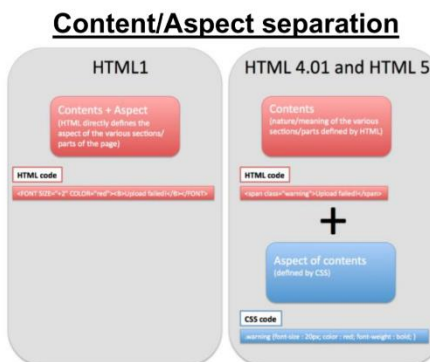
Example:

```
<audio src="teenage_dream.ogg" autoplay controls loop>
<a href="teenage_dream.ogg">listen</a>
</audio>
```

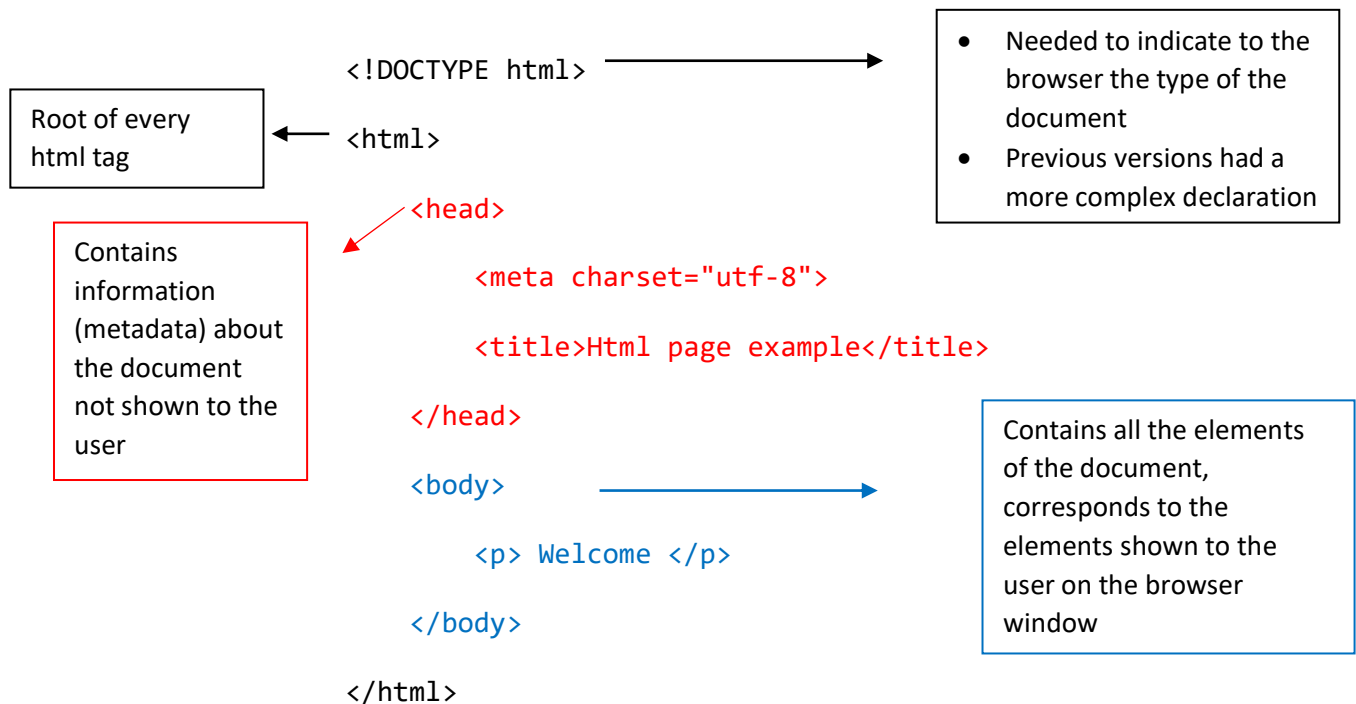
In HTML5: → will create a player control

In HTML5: → will only reproduce the audio (no audio tag)

- 2) **HTML based on the structure not presentation**: This principle of HTML as a **semantic language** has evolved to a **presentation language** since the introduction of CSS



## Structure of an HTML doc



- Indentation is irrelevant
- No case sensitivity

## Head

- **The head can contain these elements:**

- **<title>** → appears at the top of the browser window, required in every document

- **<style>** → used to define style information (CSS) for a document.

Eg:

```
<style>
  h1 {color:red;}
  p {color:blue;}
</style>
```

- **<base>** → specifies the base URL and/or target for all relative URLs in a document.

Eg:

```
<base href="https://www.w3schools.com/" target="_blank">
```

- **<link>** → defines the relationship between the current document and an external resource. is most often used to link to external style sheets or to add a favicon to your

website.is an empty element, it contains attributes only.

Eg: `<link rel="stylesheet" href="styles.css">`

- **<script>** → used to embed a client-side script, either contains scripting statements, or it points to an external script file through the src attribute. It has many attributes (see doc).

Eg:

```
<script>
document.getElementById("demo").innerHTML = "HelloJavaScript";
</script>
```

- **<noscript>** → Defines an alternate content to be displayed to users that have disabled scripts in their browser or have a browser that doesn't support script.

Eg:

```
<script>
document.getElementById("demo").innerHTML = "Hello
JavaScript!";
</script>
```

- **<meta>** → used to define the content, different attributes:

Attribute	Value	Description
<b>charset</b>	character_set	Specifies the character encoding for the HTML document.*
<b>content</b>	text	Specifies the value associated with the http-equiv or name attribute
<b>http-equiv</b>	content-security-policy, content-type, default-style refresh	Provides an HTTP header for the information/value of the content attribute
<b>name</b>	application-name, author, description, generator, keywords, viewport	Specifies a name for the metadata

\* To display an HTML page correctly a web browser must know which character set to use.

It describes the mapping between chars and integer numbers:

- **ASCII: 128** different alphanumeric characters: {numbers (0-9), English letters (A-Z), some special characters e.g. ! \$ + - ( ) @ < >}
- **ISO-8859-1**: default character set for HTML 4. This character set supported 256 different character codes

- **ANSI** (Windows-1252): the original Windows character set. Add 32 extra characters to ISO-8859-1
- **UTF-8** (Unicode): default character set for HTML5.

The **URL support only ASCII format**, it is then possible that some characters are then formatted with the % followed by two hexadecimal characters.

`<meta name="viewport" content="width=device-width, initial-scale=1.0">`  
permits to adapt the content dimension to the client window

## Tags

- From HTML5 the ending slash "/" is optional
- The tags can be nested:

Eg:

`<B>` The winner is: `<I>` Sofia Goggia! `</B>` Congratulations! `</I>` →

The winner is: *Sofia Goggia! Congratulations!*

- **Attributes:**
  - Attributes provide additional information about elements
  - Attributes are always specified in the start tag
  - Attributes usually come in name/value pairs like: `name="value"`
  - You can add multiple attributes within a single tag
  - Some attributes only have name (no associated value)
  - Values are limited to 1024 characters in length
  - Attributes values are within single or double quotes.
- **Basic Tags:**
  - Headings → `<h1>...<h6>`
  - Emphasis → `<em>`, `<strong>`, `<b>`, `<i>`
  - Subscript, superscript → `<sub>`, `<sup>`

**This is H1**

**This is H2**

**This is H6**

**Bold text**

***Strong italic***

normal text

*Emphasis text*

Subscript

Superscript

*citation*

Eg:

<pre>&lt;!DOCTYPE html&gt; &lt;html&gt;   &lt;head&gt;     &lt;meta charset="utf-8"&gt;     &lt;title&gt;Html page example&lt;/title&gt;   &lt;/head&gt;   &lt;body&gt;     &lt;h1&gt;This is H1&lt;/h1&gt;     &lt;h2&gt;This is H2&lt;/h2&gt;     &lt;h6&gt;This is H6&lt;/h6&gt;</pre>	<pre>&lt;b&gt;Bold text&lt;/b&gt; &lt;br&gt; &lt;strong&gt; &lt;i&gt;Strong italic&lt;/i&gt;&lt;/strong&gt;&lt;br&gt; normal text &lt;br&gt; &lt;em&gt;Emphasis text&lt;/em&gt;&lt;br&gt; &lt;sub&gt;Subscript&lt;/sub&gt;&lt;br&gt; &lt;sup&gt;Superscript&lt;/sup&gt;&lt;br&gt; &lt;cite&gt;citation &lt;/cite&gt;&lt;br&gt; &lt;/body&gt; &lt;/html&gt;</pre>
---	--

Type	Tag	Entries
Ordered List	ol	li
Unordered List	ul	li
Description list	dl	dt → defines the term dd → describes the tem

### OL - Ordered List

1. First item
2. Second item
3. Third item



```
<ol>
  <li>First item</li>
  <li>Second item</li>
  <li>Third item </li>
</ol>
```

### UL – Unordered List

- First item
- Second item
- Third item



```
<ul>
  <li>First item</li>
  <li>Second item</li>
  <li>Third item </li>
</ul>
```

### DL – Description List

First item  
Description of 1st item  
Second item  
Description of 2<sup>nd</sup> item



```
<dl>
  <dt>First item</dt>
  <dd>Descrip 1..</dd>
  <dt>Second item</dt>
  <dd>Descrip 2..</dd>
</dl>
```

## Basic Formatting: Blocks, Inline

Each HTML element has a default display value:

- **Block element:** start on a **new line**, the browser automatically add some **space before** and **after** the element. They always take the **full width available**.
  - <p> → paragraph of content, it cannot contain block-level elements
  - <div> → describes a container of data
- **Inline element:** do not start on a new line and take only the necessary width
  - <span> → is an inline container used to mark up a part of a text, or a part of a document.

## Image and Figure

**Image**→ used to embed an image in a web page. Is empty (only attributes).

### Principal attributes:

- src → specifies the path to the image. (Required)
- alt → text shown if cannot find the image. (Required)
- width, height → used to specify the dimensions, can also be inserted in style attribute. It's also used in the search engine optimization techniques on page

Eg: ``

**Figure**→ specifies self-contained content, like illustrations, diagrams, photos, code listings, etc. Is used to **semantically organize** the content of an image in the HTML document. The visual difference between the image usage is difficult to see but is very different from the browser side

The `<figcaption>` element is used to add a caption for the `<figure>` element.

### Comparison:

#### Image

I will attach an immagine of a car below



Here is an image of a car

```
<body>
  <p>
    I will attach an immagine of a car
  below
  </p>
  <div>
    
    <p>
      Here is an image of a car
    </p>
  </div>
</body>
```

#### Figure

I will attach an immagine of a car below



Here is an image of a car

```
<body>
  <p>
    I will attach an immagine of a car
  below
  </p>
  <div>
    <figure>
      
      <figcaption> Here is an image of
a car</figcaption>
    </figure>
  </div>
</body>
```

## File paths

The file path describes the location of a file in a website's folder structure, they are used when we need to link external files like web pages, images, style sheets, javascripts.

<code>&lt;img src="http://machine.domain/images/picture.jpg"&gt;</code>	<b>Absolute Path</b>
<code>&lt;img src="picture.jpg"&gt;</code>	<b>Relative Path:</b> picture.jpg is located in the same server, same folder as the current page
<code>&lt;img src="images/picture.jpg"&gt;</code>	<b>Relative Path:</b> picture.jpg is located in the images folder in the current folder
<code>&lt;img src="/images/picture.jpg"&gt;</code>	<b>Relative Path:</b> picture.jpg is located in the images folder at the root of the current web
<code>&lt;img src="../picture.jpg"&gt;</code>	<b>Relative Path:</b> picture.jpg is located in the folder one level up from the current folder

Is best practice to use **relative file paths**.

## Hyperlinks

Hyperlinks are used to **jump to another document** or another **part** of the current document. Syntax:

`<a href="url"> text </a>`

href attribute can have as value other url schemas, Eg: *tel, mailto, ftp*

**Internal Hyperlinks** → linked within the same web page. Syntax:

`<a href="#url"> text </a>`

→ add "#" corresponding to an id

`<h1 id="url">...</h1>`

## Tables

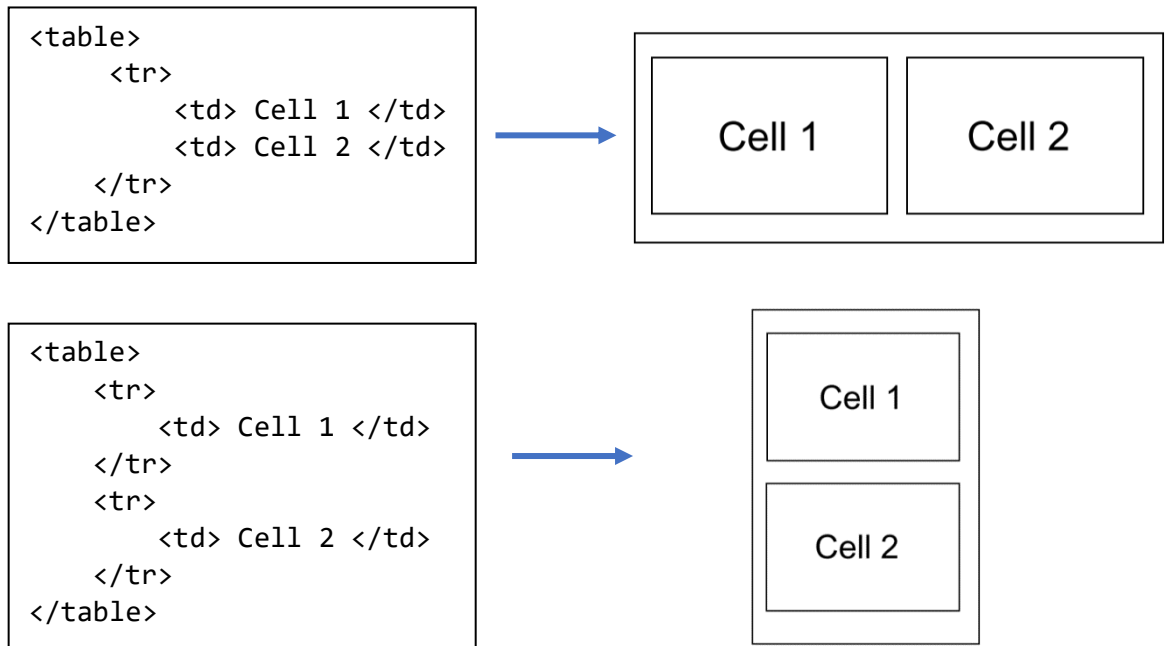
table → used to create the table

tr → row tag+

td → column tag. Attribute colspan → permits to a column to occupy more cells

th → equivalent to td but used for the header

Eg:



## Forms

A form enables to **collect** (in an interactive way) **information from the user**, the information is then passed and processed to the back end. (For an example see *Server Side Programming/ThirdSolution/CodeExn3*)

### Attributes :

- **Action** → defines the **resource** that **process action** to be performed **when** the **form** is **submitted**.  
Eg: `<form action="/action_page.php">`
- **Method** → specifies the **HTTP method** to be used when submitting the form data (GET/POST).  
Note: GET cannot be used with sensitive data and the limit of the URL is 2048 chars.  
usefull when the user wants to bookmark the result  
Eg: `<form action="/action_page.php" method="get">`
- **Autocomplete** → specifies whether a form should have autocomplete on or off
- **Enctype** → specifies the MIME type used to send data to the server, it can be used only with POST
- (Accept-charset, name, novalidate, rel)



- **Target** → specifies where to display the response that is received after submitting the form, possible values:

Value	Description
<b>_blank</b>	The response is displayed in a new window or tab
<b>_self (DEFAULT)</b>	The response is displayed in the current window
<b>_parent</b>	The response is displayed in the parent frame
<b>_top</b>	The response is displayed in the full body of the window
<b>framename</b>	The response is displayed in a named iframe

### **Elements :**

- **Input** → has different types, for example:
  - text
  - password
  - radio
  - email
  - checkbox
  - button
  - submit → defines a button for submitting form data to a form-handler, the form handler is specified in the action attribute,
  - reset
  - file
  - hidden → used to send to a server additional data without showing it to the user

Each input element has different attributes:

- required. Nb: if applied to elements with an ID it will be applied to all the elements with the same ID and it will require the selection of only one of them.
- readonly
- disabled
- size
- maxlength
- min, max
- multiple
- pattern
- placeholder
- step
- autofocus
- height, width
- list
- autocomplete

If the input is big then use text area.

- **Label** → defines a **label** for **several form elements**, it has special feature such as providing help for people with disability (the labels will be read). NB: **Click on element** = click on the **corresponding button/area**. Usually there is a bind between the label for attribute and the id attribute of an input element:

Eg:

```
<form>
  <label for="name_insert"> Enter your name here: </label>
  <input type="text" id="name_insert">
</form>
```



Enter your name here:

- **Button** → defines a **clickable button**, unlike “input” can be an image or other, usually is best practice to use also the attribute type= “button”
- **Select** → defines a **drop down list**, usually binded with “option” element.

The **first** element is **selected by default**, to **change the pre selected option** add **selected** to the **option element**:

```
<form>
  <select>
    <option value="John">Jhon</option>
    <option value="Mark" selected>Mark</option>
    <option value="Alfred">Alfred</option>
  </select>
</form>
```



it has different attributes like :

- **size** → used to **specify the number of visible values** (Eg: if in the previous example I substitute the first select with `<select size="2">` “Alfred” won’t be shown).
  - **multiple** → allows the user to select more than one value.
- **Textarea** → defines a **multi-line input** field. Attributes:
    - **rows** → specifies the visible number of line.
    - **cols** → the visible width of the text area.

## CSS

CSS = Cascading Styles Sheets.

### How to add style

Nb: without using style the browser automatically decides how to render the content.

Possible **way** to customize a tag **adding style**:

1) Inline :

Eg: `<h1 style=color:blue; text-align:center;> Random Heading </h1>`

2) Internal CSS:

Eg:

```
<head>
  <style>
    h1 {color:blue;}
    p {color: red;}
  </style>
</head>
<body>...</body>
```

3) External CSS:

HTML DOCUMENT

```
<head>
  <link rel="stylesheet" type="text/css"
    href="mystyle.css">

</head>
<body>...</body>
```

MYSTYLE.CSS

```
body{
  background color: lightblue;
}

h1 {
  color: navy;
  margin-left: 20px;
}
```

Note:

- `rel` → indicates the **type** of the **relationship between** the **html** document and the other **file** and it's required
- `type` → can be omitted

## Lengths

Two **types** of **lengths** used in CSS: **absolute** and **relative**.

### Absolute units

Absolute units are considered to always be the **same size**. **Not suggested** for **screen** but for print.

Unit	Description
cm	centimeters
mm	millimeters
in	inches (1in = 96px = 2.54cm)
px *	pixels (1px = 1/96th of 1in)
pt	points (1pt = 1/72 of 1in)
pc	picas (1pc = 12 pt)

*\* Pixels (px) are relative to the viewing device. For low-dpi devices, 1px is one device pixel (dot) of the display. For printers and high resolution screens 1px implies multiple device pixels.*

### Relative lengths

Relative length units specify a length relative to another length property.

Unit	Description
em	Relative to the font-size of the element (2em means 2 times the size of the current font)
ex	Relative to the x-height of the current font (rarely used)
ch	Relative to the width of the "0" (zero)
rem	Relative to font-size of the root element
vw	Relative to 1% of the width of the viewport*
vh	Relative to 1% of the height of the viewport*
vmin	Relative to 1% of viewport's* smaller dimension
vmax	Relative to 1% of viewport's* larger dimension
%	Relative to the parent element

*\*Viewport ? the browser window size*

**Em** and **rem** are the suggested ones.

## Colors

Colors can be expressed in two ways:

1) Names

Eg:



DeepPink

2) Values

- a. **RGB** (red, green blue) → additive color type (cmykb is subtractive, used by printers)

Eg: `rgb(255, 20, 147)` = Deep pink

- b. **RGBA** (red, green, blue, alpha) → alpha channel, specifies the opacity

Eg:

```
#p1 {background-color:rgba(255,0,0,0);}
#p2 {background-color:rgba(255,0,0,0.1);}
#p3 {background-color:rgba(255,0,0,0.5);}
#p4 {background-color:rgba(255,0,0,1);}
```

Red

Red

Red

Red

- c. **Hex** (#RRGGBB) → Hexadecimal colors

Eg: `#FF1493` = Deep pink

- d. **Hex with transparency** (#RRGGBB 00) → add two digits between 00 and FF

```
#p1 {background-color:#ff000000;}
#p1a {background-color:#ff00001F;}
#p2 {background-color:#ff00007C;;}
#p2a {background-color:#ff0000FF;;}
```

Red

Red

Red

Red

## Fonts

There are five generic font families:

- **Serif** → decorative, with strokes at the edges of each letter. *Formality and elegance*
- **Sans serif** → clean lines. *Modern and minimalistic*. (Suggested for screens).
- **Monospace** → all the letters have the same fixed width. *Mechanical*
- **Cursive/script/calligraphic** → human handwriting
- **Fantasy** → decorative

Generic Font Family	Examples of Font Names
Serif	Times New Roman Georgia Garamond
Sans-serif	Arial Verdana Helvetica
Monospace	Courier New Lucida Console Monaco
Cursive	<i>Brush Script MT</i> <i>Lucida Handwriting</i>
Fantasy	<b>Copperplate</b> Papyrus

Google fonts are special category, not HTML standard, to use them, add in the head section:  
<link rel="stylesheet" href="https://fonts.googleapis.com/css?family=NAME">

## Font choice

An important feature is the **font family** attribute, is used to ensure to provide a font in case the font chosen is not supported by the browser (or not present), syntax:

```
p{font-family: fontName1,fontName2, ..., fontType;}
```

The **fonts should be gradually more generic**, ending possibly with a font category. In this way the preferred font is the first one in the list, if not present it will be chosen the first present, if no one is present, it will be picked a present font of the type specified. (Note: if the name has a space then "" are required).

Eg: `p{font-family: Helvetica, Verdana, sans-serif;}`

**Web Safe Fonts** (supported by most browser):

- Arial (sans-serif)
- Verdana (sans-serif)
- Tahoma (sans-serif)
- Trebuchet MS (sans-serif)
- Times New Roman (serif)
- Georgia (serif)
- Garamond (serif)
- Courier New (monospace)
- Brush Script MT (cursive)

Note: if more fonts are used is suggested to create a contrast, font superfamilies are created on this principle and often there is one serif and one sans-serif

## Font referencing

@font face is a CSS rule that allows to input your own font or **non-standard ones**, usage:

```
@font-face {  
  font-family: 'fontName';  
  src:  
    local(position),  
    url(url),  
    format(type);  
}
```

Eg:

```
@font-face {  
  font-family: 'xyzfont';  
  src:  
    local('XYZfont'),  
    url('https://fonts/xy.otf'),  
    format('opentype');  
}
```

The **first search** is **local** (user), if not present it searches it (in a folder of the website or online) , there are different formats, Chromium based browser usually have otf or ttf, Explorer use eot.

Then they are referenced like a normal one:

Eg: p{font-family: xyzfont, Verdana, sans-serif;}

## Font attributes

- **font size** → *absoluteSize* | *relativeSize* | *percentage* | *length*  
absoluteSize → xx-small | x-small | small | medium | large | x-large | xx-large | xx-large  
relativeSize: → larger | smaller
- **font-style** → normal | italic
- **font-weight** → bold | bolder | lighter | normal | 100 | 200... | 900
- **text-align** → left | center | right | justify
- **text-decoration** → line-through | overline | underline
- **text-transform** → none | capitalize | uppercase | lowercase

## Background

- **background-color**: *colorname/UTF/RGB/ RGBA*  
usually works with the **opacity** attribute :

```
div {  
  background-color: green;  
  opacity: 0.3;  
}
```

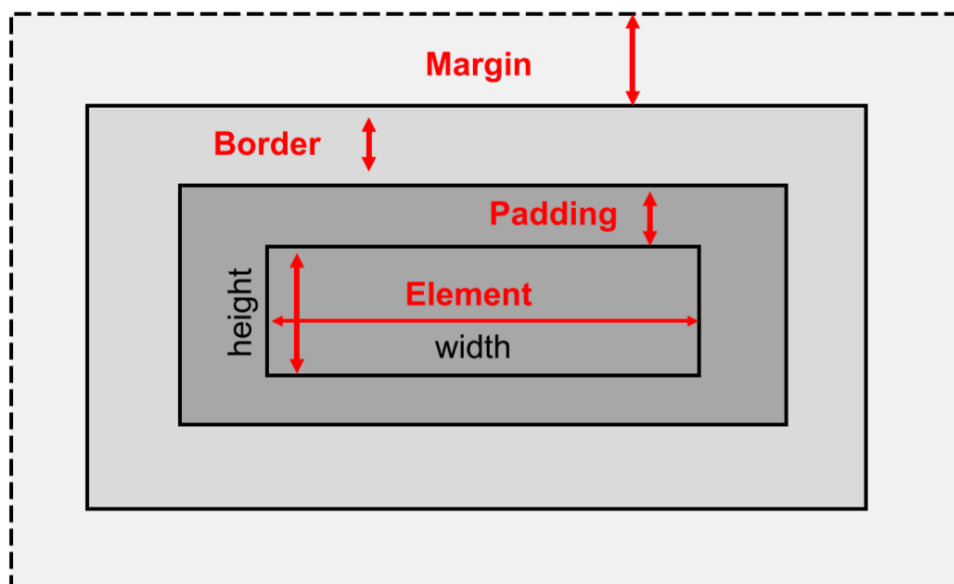


- **background-image**: sets an image as the background of an element, by **default** the image is **repeated** so it covers the entire element. Usually used with the **repeat** attribute that has this values:
  - repeat-x
  - repeat-y
  - no repeat
- **background-position**: used to specify the **position** of the background image.
- **background-attachment**: specifies whether the background should **scroll** or be **fixed**.
- **background**: **shorthand**, it can be used to set the previous attributes in **one declaration**.

Eg:

```
body {  
  background: #ffffff url("img_tree.png") no-repeat right top;  
}
```

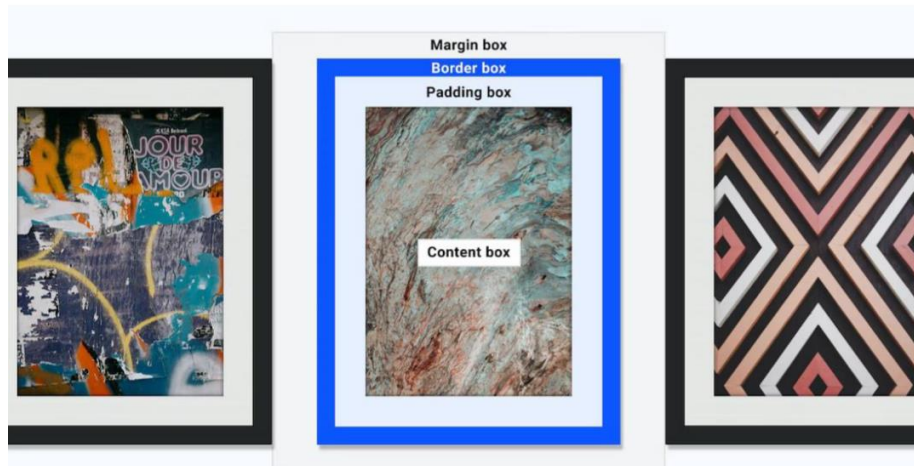
## Box Model





- **Element**/Content → Content of the box, where text and images appear.
- **Padding** → Clears an area around the content. Has the **same color** of the **element** (transparent).
- **Border** → Bordered that goes around the padding and content.
- **Margin** → Clear area outside the border. The margin is **transparent**.

Total element width = width + left padding + right padding + left border + right border + left margin



+ right margin.

## Formatting boxes

- **padding-bottom, padding-top, padding-right, padding-left:** *absolute | relative | percentage*
- **padding:** *absolute | relative | percentage*. → applied for all the 4.
- **border-color:** *color*
- **border-style:** *dotted | dashed | solid | double | groove | none*
- **border-width:** *thin | medium | thick | absolute | relative*
- **margin-bottom, margin-top, margin-right, margin-left:** *auto | absolute | relative | percentage*

## Lists

- **line-style-type:** *decimal | lower-alpha | upper-alpha | lower-roman | upper-roman*. → for **ol**.
- **line-style-type:** *circle | disc | square* → for **ul**. *disc* = empty circle.
- **line-style-image:** *Es: line-style-image: url('triangle.gif')*

## Selectors

Css selectors select the HTML element(s) to style.

There are five families of selectors:

- Basic
- Combinator
- Pseudo-class
- Pseudo-element
- Attribute

### Basic

- **Element selector**. Syntax: **tag {style}**  
*Eg: p {text-align: center; color: red} → all the p elements will have these properties.*
- **Id selector**. Syntax: **#id {style}**  
*Eg: #carpar1 {text-align: center; color: red} → The carpar1 element will have these properties.*
- **Class selector**. Syntax: **.className {style}**  
*Eg: .vehicle {text-align: center; color: red} → The elements belonging to the vehicle class will have these properties.*
- **Universal selector**. Syntax: **\* {style}**  
*Eg: #carpar1 {text-align: center; color: red} → The carpar1 element will have these properties.*
- **Grouping selector**. Syntax: **tag1, tag2 ,..., tagn {style}**  
*Eg: p, h1,h2 {text-align: center; color: red} → The p, h1, h2 elements will have these properties.*

### Combinator

Combinators select elements **exploiting** the specific **relationship** between them.

There are five different combinators in CSS:

- 1) **Simple + Class**. Syntax: **tag.className {style}**  
*Eg: p.red {text-align: center; color: red} → The elements of class red inside a paragraph will have these properties.*
- 2) **Descendant selector**. Syntax: **tag1 tag2 {style}**  
*Eg: div p {text-align: center; color: red} → The descendent of all levels inside a div element and belonging to a paragraph will have these properties.*

3) **Child selector**. Syntax: **tag1 > tag2 {style}**

Eg: `div > p {text-align: center; color: red}` → All the sons of the div element will have these properties.

Nb:

```
<div>
  <p> car1 </p> → son = applied
  <p> car2 </p> → son = applied
  <section>
    <p> car3 </p> → grandson = NOT applied (would work with a descendant selector)
  </section>
  <p> car4 </p> → son = applied
</div>
```

4) **Adjacent sibling selector**. Syntax: **tag1 + tag2 {style}**

Eg: `div + p {text-align: center; color: red}` → The first adjacent sibling will have these properties.

Nb:

```
<div>
  <p> I like Pizza</p> → NOT sibling = NOT applied
</div>
<p> I love Pizza too</p> → adjacent sibling= applied
<p> I think is bad</p> s → NOT adjacent = NOT applied
```

5) **General sibling selector**. Syntax: **tag1 ~ tag2 {style}**

Eg: `div ~ p {text-align: center; color: red}` → All the siblings will have these properties.

In the previous example also the last p would be selected

## Pseudo-Class and Pseudo-elements

**Pseudo-class selectors** are used to define a **special state** of an **element**: mouse over, link is visited or not and so on.

Eg: `button:hover{ color: blue;}`

Documentation:

[https://www.w3schools.com/css/css\\_pseudo\\_classes.asp](https://www.w3schools.com/css/css_pseudo_classes.asp)  
<https://developer.mozilla.org/en-US/docs/Web/CSS/Pseudo-classes>

**Pseudo-element selectors** are used to **style specified parts** of an **element**: the first letter, the first line...

Eg: `p::first-line {color: blue;text-transform: uppercase;}`

Documentation:

[https://www.w3schools.com/css/css\\_pseudo\\_elements.asp](https://www.w3schools.com/css/css_pseudo_elements.asp)  
<https://developer.mozilla.org/en-US/docs/Web/CSS/Pseudo-elements>

## Attribute selectors

**Attribute selectors** are used to style elements that have **specific attributes** or **attribute values**.

Selector	Example	Description
<b>[attribute]</b>	[target]	Selects all elements with a target attribute
<b>[attribute=value]</b>	[target=_blank]	Selects all elements with target="_blank"
<b>[attribute~=value]</b>	[title~=flower]	Selects all elements with a title attribute containing the word 'flower'
<b>[attribute =value]</b>	[lang =en]	Selects all elements with a lang attribute value equal to "en" or starting with "en-"
<b>[attribute^=value]</b>	a[href^="https"]	Selects every <a> element whose href attribute value begins with "https"
<b>[attribute\$=value]</b>	a[href\$=".pdf"]	Selects every <a> element whose href attribute value ends with ".pdf"
<b>[attribute*=value]</b>	a[href*="w3schools"]	Selects every <a> element whose href attribute value contains the substring "w3schools"

## Cascading and positioning

The steps that apply to the cascading order:

- 1) Selection of rules
- 2) Origin and importance
- 3) Specificity
- 4) Order of appearance

## Origin and importance

Order (high to low)	Origin	Importance
1	user-agent (browser)	!important
2	user	!important
3	author (developer)	!important
4	author (developer)	normal
5	user	normal
6	user-agent (browser)	normal

## Specificity

In case of **equality** with an **origin**, the **specificity** of a **rule** is the **choice criterium**. The declaration with the highest specificity wins.

### Order of decreasing specificity:

- 1) **Inline styles** (always overwrite any normal style in author stylesheet, the only way to override is to use the flag `!important` ). E: `<h1 style="color: pink;">`
- 2) **Id selectors**. Eg: `#navbar`
- 3) **Class selectors, pseudo-classes, attribute selectors**. Eg: `.test`, `:hover`, `[href]`
- 4) **Elements and pseudo-elements**. Eg: `h1`, `::before`.

Universal selector DOES NOT impact the specificity

Combinator selectors: `+`, `>`, `~`, DO NOT impact the specificity.

### How to calculate specificity:

- `!important` → wins over everything,
- Inline → wins over everything except important.
- add 100 for each ID value.
- add 10 for each class value, pseudo-class or attribute selector.
- add 1 for each element selector pseudo element.

Eg:

Selector	Specificity Value	Calculation
p	1	1
p.test	11	1 + 10
p#demo	101	1 + 100
<p style="color: pink;">	1000	1000
#demo	100	100
.test	10	10
p.test1.test2	21	1 + 10 + 10
#navbar p#demo	201	100 + 1 + 100
*	0	0 (the universal selector is ignored)

## Cascading order

Order of appearance: In the origin with precedence, if there are competing values for a property that are in style block matching selectors of equal specificity, the **last declaration in the style order is applied**.

Eg:

```
<html>
  <head>
    <style>
      span+p {color:red;}
      span.p {color:green;}
      body {color:orange;}
    </style>
  </head>
  <body>
    <span class="p">A</span>
    <span>B</span><p>C</p>
    <span class="p"> D</span><p>E</p>
  </body>
</html>
```

A B  
C  
D  
E

Note: span doesn't go to the top of the line

A: span.p =1+10 vs body =1

B: body=1

C: span+p=2 vs body =1

D: span.p=1+10 vs body =1

E: span+p=2 vs body =1

## Positioning

The position property is used to set position for an element. Only after the position is set the left right , top and bottom properties can be used. These properties behave differently, depending on the value of position.

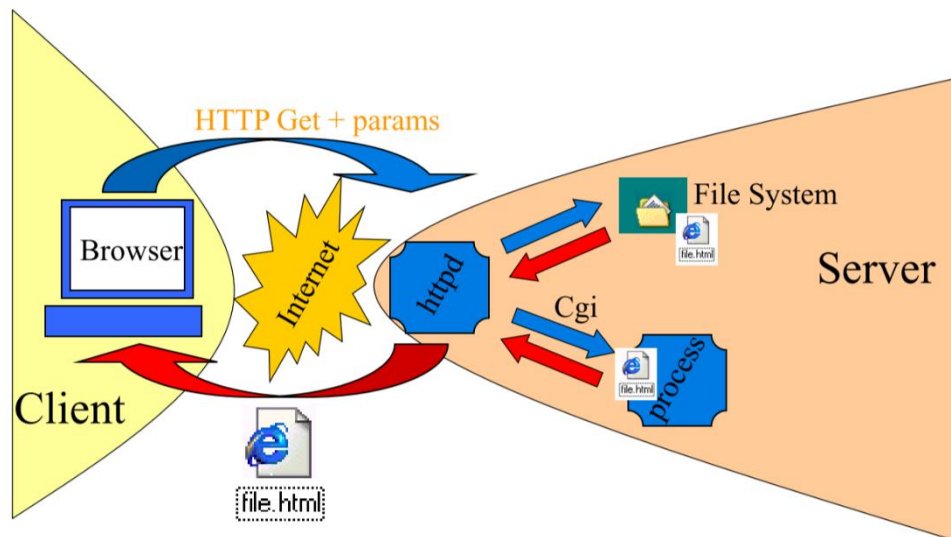
There are several position values:

- Static → the default position of all HTML elements. They are always positioned according to the normal flow of the page.
- Relative → setting the top, right, bottom and left properties of a relatively-positioned element will cause it to be adjusted away from its normal position,
- Absolute → setting the top, right, bottom, and left properties of an absolutely-positioned element will cause it to be adjusted in an absolute position with respect to the nearest positioned ancestor (if there is no ancestor, use the body)

## Server side programming

One of the main reason that represented the start of the server side programming was the need to have **dynamic pages**.

### First solution: CGI



**Httpd** = Daemon http, **process** of the **operating systems** that handles the dynamic requests. (in xampp: `apache/bin/httpd.exe`).

Request is **static** -> simple access to the file system.

Request is **dynamic** → daemon **calls** another **process** through an interface called **CGI** (common gateway interface). The process **elaborates** the **request** and then it **sends** it **back** to the daemon as an **http file**.

The daemon recognize that the request is dynamic because the url contain `"/cgi-bin"`.

This is a general purpose solution: the process called can use different languages.

The process called through the CGI can access to the File System.

The "next" version is FastCGI.

## Example

```
@echo off
echo Content-Type: text/html
echo.

set mydate=%date%
set mytime=%time%

echo ^<DOCTYPE html^>
echo ^<html lang="en"^>
echo ^<head^>
echo  ^<meta charset="utf-8"^>
echo  ^<meta name="viewport"
content="width=device-width, initial-
scale1.0"^>
echo  ^<title^> What time is it?^</title^>
echo ^</head^>
echo ^<body^>
echo  <p^>Hello, I am a bat script^</p^>
echo  <p^>Today is %mydate%^</p^>
echo  <p^>and time is %mytime%^</p^>
echo ^</body^>
echo ^</html^>
```

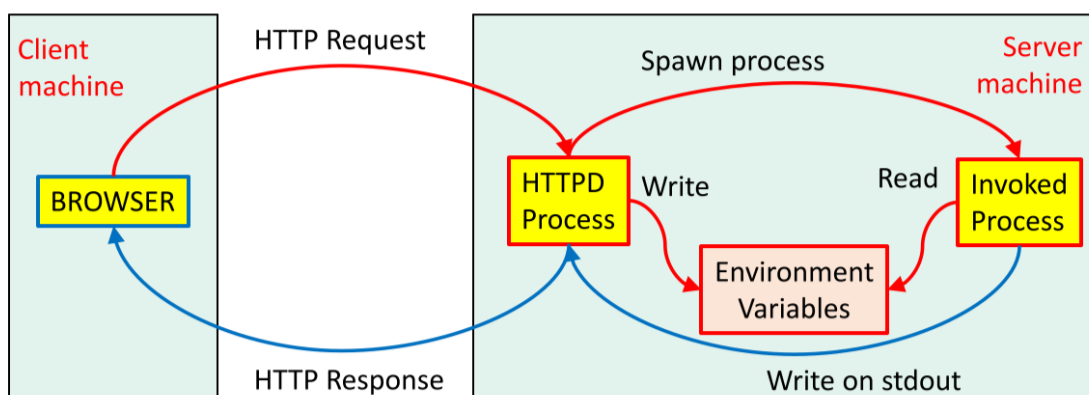
@echo off → Avoid printing all commands on the screen

echo. → Print a line vuota

date and time are system environment variables accessible by all processes

The "^" is used to prevent to interpret < and > as shell command

In this way the process prints on stdout the page and is sent to httpd that put the result inside the body of an http response.



Access to environment vars is supported in all programming languages!



CGI has many environment variables for the requests:

- `SERVER_PROTOCOL` : HTTP/*version*.
- `SERVER_PORT` : TCP port (decimal).
- `REQUEST_METHOD` : name of HTTP method (see above).
- `PATH_INFO` : path suffix, if appended to URL after program name and a slash.
- `PATH_TRANSLATED` : corresponding full path as supposed by server, if `PATH_INFO` is present.
- `SCRIPT_NAME` : relative path to the program, like `/cgi-bin/script.cgi`.
- `QUERY_STRING` : the part of URL after `?` character. The query string may be composed of *\*name=value* pairs separated with ampersands (such as `var1=val1&var2=val2...`) when used to submit form data transferred via GET method as defined by HTML application/x-www-form-urlencoded.
- `REMOTE_HOST` : host name of the client, unset if server did not perform such lookup.
- `REMOTE_ADDR` : IP address of the client (dot-decimal).
- `AUTH_TYPE` : identification type, if applicable.
- `REMOTE_USER` used for certain `AUTH_TYPE` s.
- `REMOTE_IDENT` : see `ident`, only if server performed such lookup.
- `CONTENT_TYPE` : Internet media type of input data if PUT or POST method are used, as provided via HTTP header.
- `CONTENT_LENGTH` : similarly, size of input data (decimal, in octets) if provided via HTTP header.
- Variables passed by user agent ( `HTTP_ACCEPT` , `HTTP_ACCEPT_LANGUAGE` , `HTTP_USER_AGENT` , `HTTP_COOKIE` and possibly others) contain values of corresponding HTTP headers and therefore have the same sense.

The following environment variables are not request-specific and are set for all requests:

- `SERVER_SOFTWARE`  
The name and version of the information server software answering the request (and running the gateway). Format: name/version
- `SERVER_NAME`  
The server's hostname, DNS alias, or IP address as it would appear in self-referencing URLs.
- `GATEWAY_INTERFACE`  
The revision of the CGI specification to which this server complies. Format: CGI/revision

Codice in perl per stampare ogni variabile (coppia nome, valore):

```
print "Content-type: text/plain; charser-iso-8859-1\n\n";
foreach $var (sort(keys($ENV))){
    $val= %ENV(%var);
    $val=~ s|\n|\\n|g;
    $val=~ s|"|\\"|g;
    print "${var}~\"${val}\"\\n";
}
```

## Parameters passing

```
<form action= "http://localhost:80/cgi-  
bin/params.html" method= "GET">  
  <label for="firstname">Insert first name</label>  
  <input type="text" name="firstname"> <br><br>  
  <input type="submit" value="Submit">  
  <input type="reset" value="reset">  
</form>
```

Insert first name

Submit

reset

The url will be: url..../?variable\_name1=value1&variable\_name2=value2&...

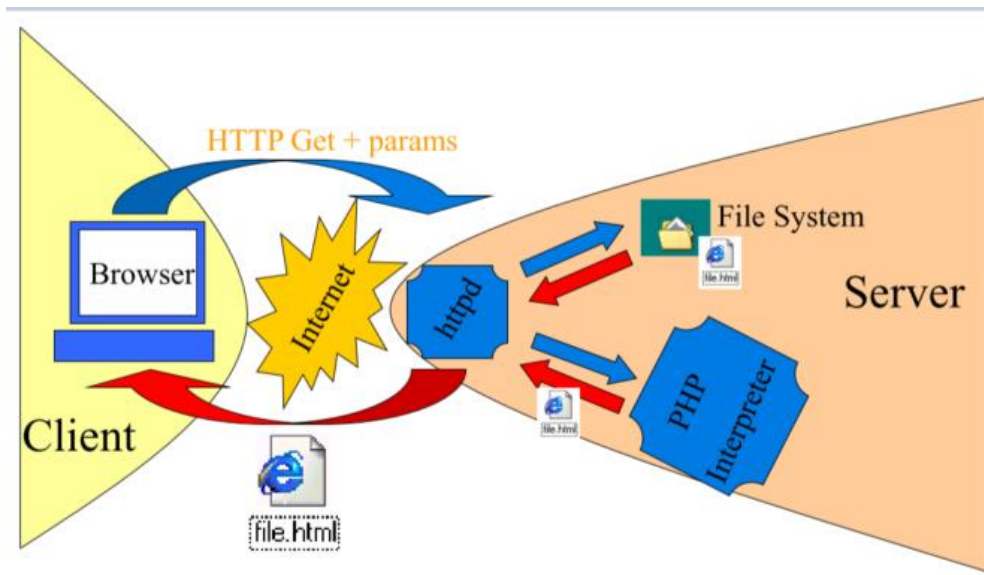
With a POST this won't happen.

### Disadvantages of this solution:

- There is some Html embedded in others languages.

A possible solution is to use the opposite approach → embed the code in the html.

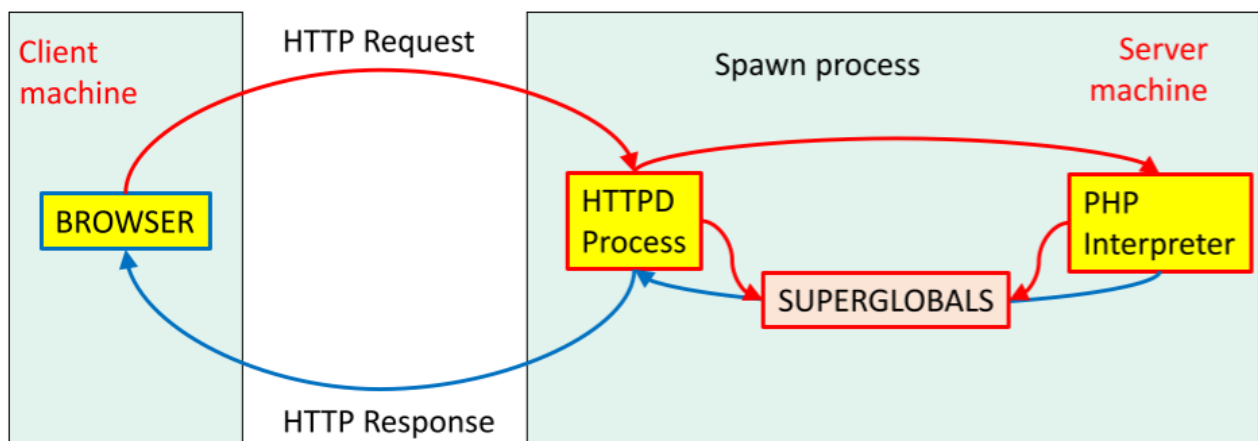
## Second solution: PHP



**PHP** = PHP hypertext processor. PHP is interpreted, not typeset (below php7) and case insensitive.

The dynamic part is now handled directly inside the html. Eg:

```
<html>
<body>
  <p> Hello! today is</p>
  <?php
    echo date("d/m/Y")
  ?>
</body>
```



## PHP variable scope

Variables can be declared anywhere in the script.

Three different variable scopes:

1) **Local** :

A variable declared within a function has a local scope and can only be accessed within that function

2) **Global** :

A variable declared outside a function has a global scope and can only be accessed outside a function.

3) **Static** :

The static keyword is used to declare properties and methods of a class as static. Static properties and methods can be used without creating an instance of the class.

The static keyword is also used to declare variables in a function which keep their value after the function has ended.

There are two **ways** to **access** a **variable** from inside a function:

- Using the keyword **global**
- Using the associative **array %GLOBALS[key]**. GLOBALS contains all the global key words of the program and is a superglobal variable

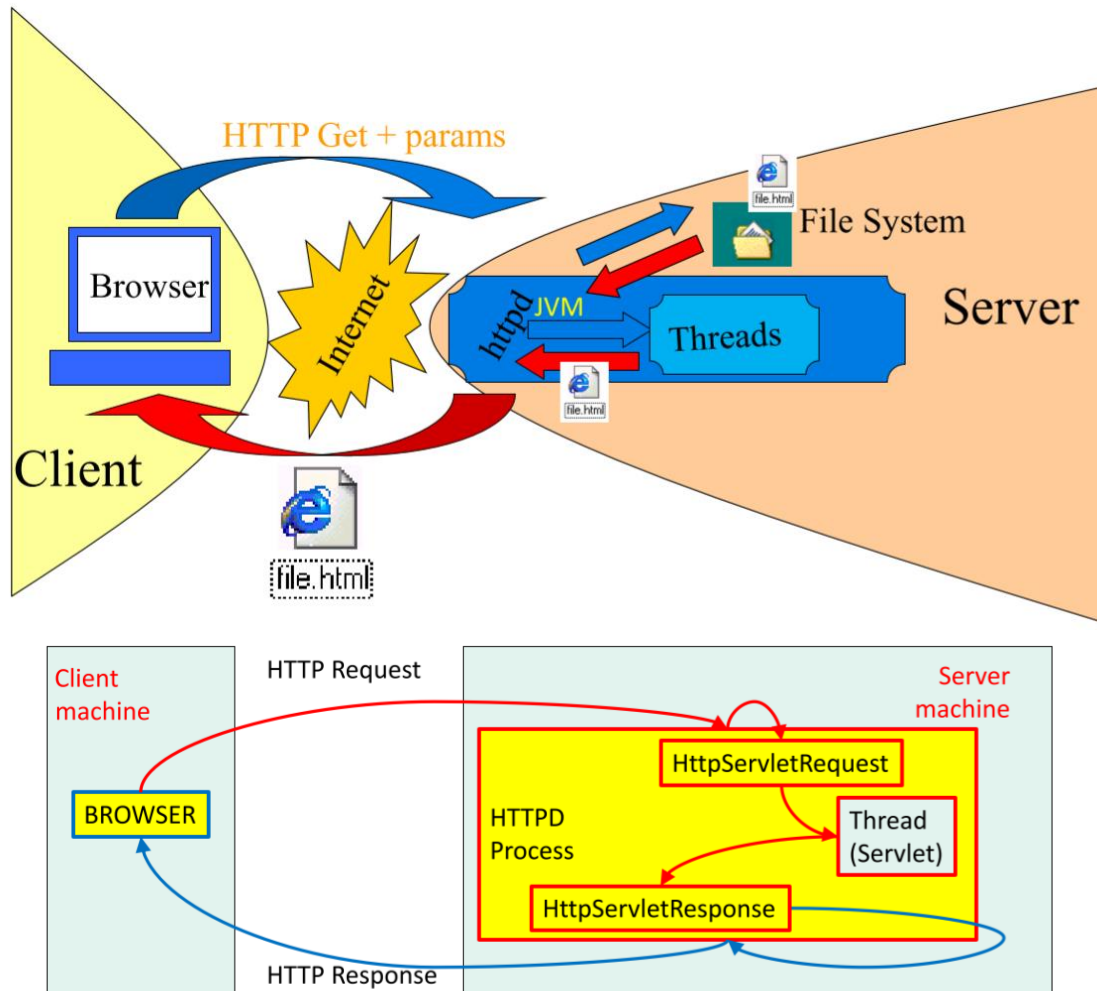
**Superglobal** variable = **built-in variables always accessible** from any function, class, file, that is accessible in all scopes throughout a script.

PHP superglobals:

- **\$GLOBALS**
- **\$\_SERVER**
- **\$\_REQUEST**
- **\$\_POST**
- **\$\_GET**
- **\$\_FILES**
- **\$\_ENV**
- **\$\_COOKIE**
- **\$\_SESSION**

**Disadvantages of this solution** → Each request creates a process.

### Third Solution: Java servlets



This solution uses **threads instead of processes**. The **request** is **firstly received** by the **httpd** that **passes** the **request** to the **servlet container** responsible for managing the servlets. The servlet container uses a system of threads, **when a request comes in the container assigns it to an available thread** which **processes the request using the appropriate servlet**. The daemon then prepares the response to be sent to the client.

**Apache Tomcat** is a **servlet container** run on a **Java Virtual Machine (JVM)**.

Tomcat utilizes the Java servlet specification to execute servlets generated by requests, often with the help of JSP pages, allowing dynamic content to be generated much more efficiently than with a CGI script.

### Servlet lifecycle

- 1) The **servlet** is **initialized** by calling the **init()** method.

```
public void init() throws ServletException {
    // Initialization code...
}
```

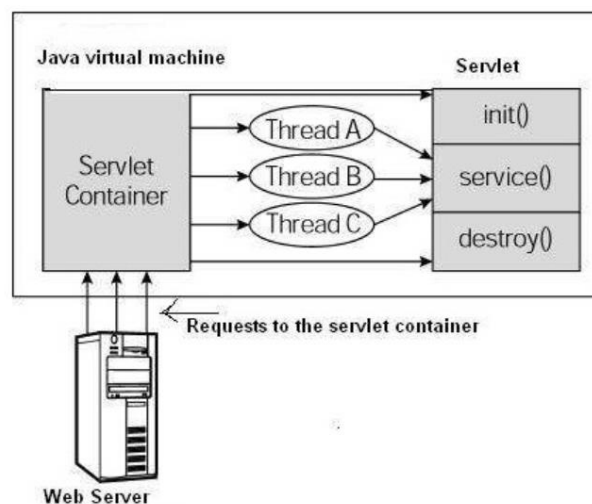
- 2) Then is called the **service()** method, the main method to perform an actual task, the **method checks** the **HTTP request** type (GET, PUT, DELETE, ecc..) and **calls one** between doGet, doPost, doDelete...

```
public void service(ServletRequest request, ServletResponse
response)
    throws ServletException, IOException {
}
```

Nb: the methods called can be overridden.

- 3) The **destroy()** method is called only once at the **end of the cycle** of a **servlet**. This method gives the servlet a chance to close database connections, halt background threads, write cookies lists or hit counts to disk and perform other activities. After the destroy() method is called the servlet object is marked for garbage collection. It's a generic servlet (higher than the http servlet class).

```
public void destroy() {
    // Finalization code...
}
```



#### Summary:

- First the HTTP request coming to the server are delegated to the servlet container
- The servlet container loads the servlet before invoking the service() method
- Then the servlet container handles multiple requests by spawning multiple threads, each thread executing the service() method of a single instance of the servlet.

Multiple users could want the same time.

Tomcat manages the memory and the multithreading.

## Interaction with IntelliJ Idea

How to **create a project**:

- 1) New project
- 2) Jakarta EE
- 3) Template: Web Application
- 4) Application server: Tomcat 8.5.79
- 5) Next →
- 6) Version JAVA EE 8
- 7) Specifications: web profile.
- 8) Create →
- 9) Delete src -> HelloServlet and resources->webapp->index.jsp

To perform **Local deployment**:

- 1) Services Tomcat, EditConfig, Server and Deployment, remove “\_war-exploded” from the name. And if wanted also in the deployment window of the run/debug configuration
- 2) Add in the web.xml the home file (html or the servlet name) if the file id different from index.html
- 3) In the folder of the project copy the -SNAPSHOT folder and copy in the xampp Tomcat webapps folder removing the version number and -snapshot from the name of the folder

Other options/info:

- Setup admin user in tomcat: go to the ApacheXamp\tomcat\conf folder, and add  
    <role rolename="manager-gui"/>  
    <user username="tomcat" password="admin" roles="manager-gui"/>

The manager of Tomcat is accessible from <http://localhost:8080/manager/html> or clicking on “Manager App” at <http://localhost:8080/> only if run from XAMPP.

- Usually the HTML and jsp files are inside the “webapp” folder, this files are **directly accessible** by using the correct **url**. To **avoid** this is possible to insert the html or jsp file in the “WEB-INF” folder. T

## Code Example n.1: Basic Servlet

With all the setup of the previous paragraph two files are modified:

- The servlet in the src/main/java/ folder → Basic\_Servlet\_Test.java
- The xml file in the webapp/WEB-INF/ folder → web.xml

### Basic Servlet Test.java

```
@WebServlet(name = "Basic_Servlet_Test", value = "/Basic_Servlet_Test")
// Annotation
public class Basic_Servlet_Test extends HttpServlet {

    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException {
        processRequest(request,response);
    }

    protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws IOException {
        response.setContentType("text/html");
        processRequest(request,response);
    }

    protected void processRequest(HttpServletRequest request, HttpServletResponse
        response) throws SecurityException, IOException{

        response.setContentType("text/html;charset=UTF-8");
        try(PrintWriter out = response.getWriter()){
            out.println("<!DOCTYPE html>");
            out.println("<html><head>");
            out.println("<title>Servlet ReadPost</title>");
            out.println("</head><body>");
            out.println("<h1>Servlet Basic_Servlet_Test at
                "+request.getContextPath()+"</h1>");
            out.println("</body>");
            out.println("</html>");
        }
    }
}
```

### Output:



## Servlet Basic\_Servlet\_Test at /BasicServlet

**Note:** if we try to access to localhost:8080/BasicServlet we get a 404 not found. This is because there is not the index.html file that is the default handler neither there is a specification of an other file that handles the root path.

**Note 2:** The annotations are used for adding metada to the servlet, this data can be set int the web.xml (using servlet mapping), if the data are incoherent web.xml overwrites annotation.



Code Example n.2: Basic Servlet, Date and Time

It has the same files like the previous example but now the servlet is called DateAndTime.

DateAndTime.java

```
@WebServlet(name = "DateAndTime", value = "/DateAndTime")
public class DateAndTime extends HttpServlet {

    public void create_response(HttpServletResponse response)
        throws IOException {

        DateTimeFormatter day = DateTimeFormatter.ofPattern("dd/MM/yyy");
        DateTimeFormatter time = DateTimeFormatter.ofPattern("HH:mm:ss");
        LocalDateTime now = LocalDateTime.now();
        response.setContentType("text/html");
        response.setCharacterEncoding("UTF-8");
        try (PrintWriter out = response.getWriter()) {
            out.println("<!DOCTYPE html>");
            out.println("<html lang=\"en\">");
            out.println("<head><title>What time is it?</title></head>");
            out.println("<body>");
            out.println("<p>Hello, I am a servlet</p>");
            out.println("<p> Today is ");
            out.println(day.format(now));
            out.println("</p>");
            out.println("<p> and time is ");
            out.println(time.format(now));
            out.println("</p>");
            out.println("</body></html>");
        }
    }

    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        create_response(response);
    }

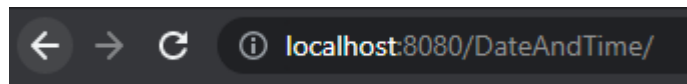
    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        create_response(response);
    }
}
```

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd"
    version="4.0">
    <welcome-file-list>
        <welcome-file>DateAndTime</welcome-file>
    </welcome-file-list>

</web-app>
```

**Output:**



Hello, I am a servlet

Today is 04/04/2023

and time is 18:16:23

**Note:** now the servlet is accessible both from to localhost:8080/DateAndTime that is the root directory ( the fact that the project name/root has the same name as the servlet it could has ben called Pippo) and the respective Servlet URL: to localhost:8080/DateAndTime/DateAndTime.

## HttpServletRequest

One of the most useful interfaces for **handling servlets** is the **HttpServletRequest** which is a subinterface of **ServletRequest**.

Methods used to **get parameters**:

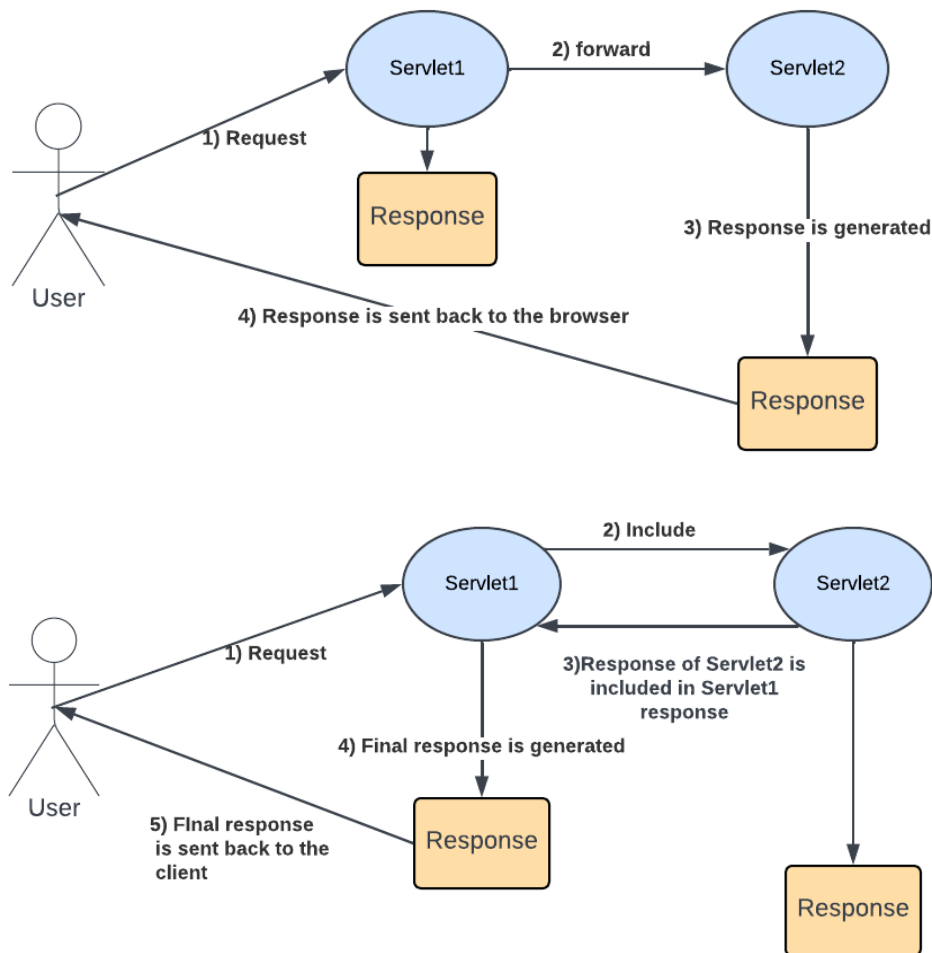
<b>getParameter(String name) → String</b>	Returns the value of a request parameter as a String, or null if the parameter does not exist
<b>getParameterMap() → Map&lt;String,String[]&gt;</b>	Returns a java.util.Map of the parameters of this request (key-value pair)
<b>getParameterNames() → Enumeration&lt;String&gt;</b>	Returns an Enumeration of String objects containing the names of the parameters contained in this request
<b>getParameterValues(String name) → String []</b>	Returns an array of String objects containing all of the values the given request parameter has or null if the parameter does not exist.

There are different methods both from **ServletRequest** and **HttpServletRequest** that cover different topics like:

- Character encoding
- Content length
- Content Type
- Locale
- Protocol
- Local/remote name, address port
- Server name
- Security
- Headers
- Method
- Path info
- Query String
- Session
- Cookie
- Attributes (like `setAttribute` and `getAttribute`)

See: <https://docs.oracle.com/javaee/7/api/javax/servlet/ServletRequest.html>.

## Include vs Forward



Both include and forward are managed from the `getRequestDispatcher`.

Eg: `request.getRequestDispatcher("/footer.html").include(request, response);`

- When a servlet calls **`include(request, response)`** on the `RequestDispatcher` object, it is essentially asking the web container to **insert** the content of `"/header.html"` into the **current response being built** by the **servlet**. This means that the control remains with the servlet, but it can utilize the content of `"/header.html"` in its own response.

In other words, when you call `include(request, response)`, you are telling the web container to include the output of the specified resource in the current response being generated by the current servlet, and to use the same request and response objects that were originally used to invoke the current servlet.

This allows the included resource to access and modify the same request and response objects that the current servlet is using. For example, the included resource could add headers to the response, set cookies, or modify the request attributes that were set by the current servlet.

- Similarly, if a servlet calls **forward(request, response)** on the RequestDispatcher object instead of `include(request, response)`, it will **forward** the **entire request** and **response objects** to the specified resource. This means that the **control will be transferred** to that resource, and it will handle the request and generate the response.

So usually if we forward to a static file the servlet will not complete the execution

### Code Example n.3: Structured code and Parameters, Calculator

In the first examples we considered projects with only **one servlet** that **handled both** the **html** code and the **logic** code. This approach is not very effective because a lot of the HTML code would be repeated. In this example we are using the **fragments** approach to solve the problem. In the project we have the 3 html files:

- index.html → the Homepage accessible from the root (localhost:8080/projectName/).
- header.html → used to avoid to rewrite the initial part of code of the HTML.
- footer.html → used to avoid to rewrite the last part of code of the HTML.

#### header.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width,
    initial-scale=1.0">
  <title>Calculator</title>
</head>
<body>
<header style="text-align: center"> Calculator Project</header>
```

#### footer.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Calculator</title>
</head>
<body>
<header style="text-align: center"> Calculator Project</header>
```

#### index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title> Calculator Project</title>
</head>
<body>
<header style="text-align: center"> Calculator Project</header>
  <br>
  <h1 style="text-align: center">Welcome</h1>
  <form action="."input">
    <input type="hidden" name="first_iteration" value="true">
    <label for="textA1">What is your name?</label><br>
    <input type="text" id="textA1" name="name" required> <br><br>
    <input type="submit" value="Send">
  </form>
</body>
</html>
```

The first two files are used in the servlets of the projects with this command syntax:

```
request.getRequestDispatcher("/footer.html").include(request, response);
```

The second aspect that this example introduces is **parameter handling**.

The index has a form that passes a hidden parameter called "first-iteration," which is set to true and is handled by the input servlet.

#### input.java

```
@WebServlet(name = "input", value = "/input")
public class input extends CalculatorServlet {
    protected void process_request(HttpServletRequest request, HttpServletResponse
        response) throws IOException, ServletException {

        response.setContentType("text/html");
        response.setCharacterEncoding("UTF-8");

        try (PrintWriter out = response.getWriter()) {
            request.getRequestDispatcher("/header.html").include(request, response);
            // Use the static HTML to avoid code repetition

            String name = request.getParameter("name");
            String first_iteration = request.getParameter("first_iteration");
            // obtain parameters from the url. NB: it doesn't work for a single char

            if (first_iteration.equals("true")) {
                out.println("<b>Nice To meet you, " + name + "!</b><br><br>");
                out.println("Let's try an operation: <br><br>");
            }
            else
                out.println("Go on, " + name + "!<br><br> Insert new data:
                    <br><br>");
            print_input_form(out, name); // method inherited from CalculatorServlet
            request.getRequestDispatcher("/footer.html").include(request, response);
        }
    }
}
```

The input servlet handles the request based on the value of the first iteration parameter. Then uses the print\_input\_form method defined in CalculatorServlet:

#### Part of code in CalculatorServlet.java

```
protected void print_input_form(PrintWriter out, String name) {
    out.println("<form action=\"./output\">");
    out.println("<input type=\"hidden\" name=\"name\" value=\"" + name + "\"/>");
    out.println("<input type=\"hidden\" name=\"first_iteration\"
        value=\"false\"/>");
    out.println("<label for=\"op1\">Operand 1:</label><br>");
    out.println("<input type=\"text\" id=\"op1\" name=\"op1\" required ><br><br>");
    out.println("<input type=\"radio\" id=\"add\" name=\"op\" value=\"+\" required
        >");
    out.println("<label for=\"add\">+</label><br>");
    out.println("<input type=\"radio\" id=\"dif\" name=\"op\" value=\"-\"/>");
    out.println("<label for=\"dif\">-</label><br>");
}
```

```

        out.println("<input type=\"radio\" id=\"mul\" name=\"op\" value=\"*\">");
        out.println("<label for=\"mul\">*</label><br>");
        out.println("<input type=\"radio\" id=\"div\" name=\"op\" value=\"/\">");
        out.println("<label for=\"div\"></label><br><br>");
        out.println("<label for=\"op2\">Operand 2:</label><br>");
        out.println("<input type=\"text\" id=\"op2\" name=\"op2\" required");
        ><br><br>");
        out.println("<input type=\"submit\" value=\"Submit\">");
        out.println("</form>");
    }

```

when “Submit” button is is clicked the control pass over the output Servlet:

#### output.java

```

@WebServlet(name = "output", value = "/output")
public class output extends CalculatorServlet {

    protected void process_request(HttpServletRequest request, HttpServletResponse
    response) throws IOException, ServletException {

        response.setContentType("text/html");
        response.setCharacterEncoding("UTF-8");
        try (PrintWriter out = response.getWriter()) {
            request.getRequestDispatcher("/header.html").include(request, response);

            String op1 = request.getParameter("op1");
            String op = request.getParameter("op");
            String op2 = request.getParameter("op2");
            String name = request.getParameter("name");

            out.println(compute_operation(op1, op, op2, name));
            //method for computing an operation implemented in Calculator Servlet

            out.println("Do you want to execute another operation?<br>");

            print_continue_form(out, name);

            request.getRequestDispatcher("/footer.html").include(request, response);

        } catch (ParseException e) {
            throw new RuntimeException(e);
        }
    }
}

```



The output servlets handles the request based on the parameters passed by the input servlet. Then uses the print\_continue\_form method defined in CalculatorServlet:

Part of code in CalculatorServlet.java

```
protected void print_continue_form(PrintWriter out, String name) {
    out.println("<form>");
    out.println("<input type=\"hidden\" name=\"name\" value=\"\" + name + \"\"/>");
    out.println("<input type=\"hidden\" name=\"first_iteration\" value=\"false\"/>");
    out.println("<input type=\"submit\" formaction=\"./input\" value=\"Yes\">");
    out.println("<input type=\"submit\" formaction=\"./end\" value=\"No\">");
    out.println("</form>");
}
```

Then there are two options:

- Yes → returns the control to input. (Now the first iteration parameter is set as false).
- No → gives the control to end servlet.

end.java

```
@WebServlet(name = "end", value = "/end")
public class end extends CalculatorServlet {

    protected void process_request(HttpServletRequest request,
        HttpServletResponse response) throws IOException, ServletException {
        response.setContentType("text/html");
        response.setCharacterEncoding("UTF-8");
        try (PrintWriter out = response.getWriter()) {
            String name = request.getParameter("name");
            request.getRequestDispatcher("/header.html").include(request,response);
            out.println("<b> Thanks, " + name + "! End of the conversation!!!</b><br><br>");
            request.getRequestDispatcher("/footer.html").include(request,response);
        }
    }
}
```

## Track the servlet execution

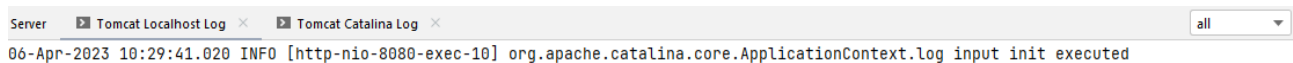
It's useful to **track** the **execution** of the **servlets** this can be done overriding the **init()** and **destroy()** method that are called by the web container when the servlet is being loaded and unloaded.

In the previous example all the servlets were subclasses of CalculatorServlet.java, we can take advantage from this adding this code:

```
public void log(String msg) {
    getServletContext().log(getServletName()+ " "+ msg);
}
@Override public void destroy() {
    log("destroy executed");
}
@Override public void init() {
    log("init executed");
}
```

the getServletContext() is use to access to the log of the container.

After have set the name we obtain:



Server Tomcat Localhost Log Tomcat Catalina Log all

06-Apr-2023 10:29:41.020 INFO [http-nio-8080-exec-10] org.apache.catalina.core.ApplicationContext.log input init executed

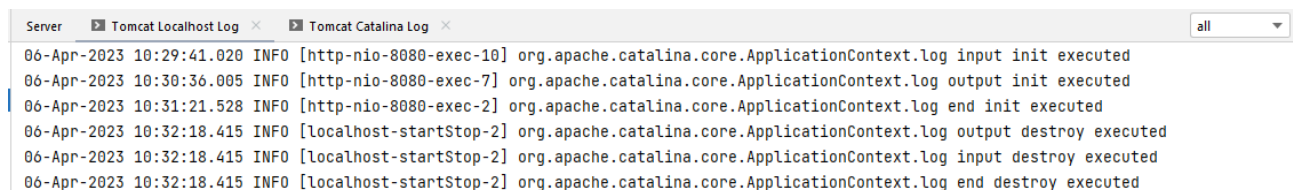
After have selected the operation and selected to not continue:



Server Tomcat Localhost Log Tomcat Catalina Log all

06-Apr-2023 10:29:41.020 INFO [http-nio-8080-exec-10] org.apache.catalina.core.ApplicationContext.log input init executed  
06-Apr-2023 10:30:36.005 INFO [http-nio-8080-exec-7] org.apache.catalina.core.ApplicationContext.log output init executed  
06-Apr-2023 10:31:21.528 INFO [http-nio-8080-exec-2] org.apache.catalina.core.ApplicationContext.log end init executed

After server closure:



Server Tomcat Localhost Log Tomcat Catalina Log all

06-Apr-2023 10:29:41.020 INFO [http-nio-8080-exec-10] org.apache.catalina.core.ApplicationContext.log input init executed  
06-Apr-2023 10:30:36.005 INFO [http-nio-8080-exec-7] org.apache.catalina.core.ApplicationContext.log output init executed  
06-Apr-2023 10:31:21.528 INFO [http-nio-8080-exec-2] org.apache.catalina.core.ApplicationContext.log end init executed  
06-Apr-2023 10:32:18.415 INFO [localhost-startStop-2] org.apache.catalina.core.ApplicationContext.log output destroy executed  
06-Apr-2023 10:32:18.415 INFO [localhost-startStop-2] org.apache.catalina.core.ApplicationContext.log input destroy executed  
06-Apr-2023 10:32:18.415 INFO [localhost-startStop-2] org.apache.catalina.core.ApplicationContext.log end destroy executed

## Serialization

All the previous examples were not saving any data in the server. One useful option to do so is **serialization** = Java Serialization is the process of **converting an object's state into a byte stream** that can be **stored** in a file, sent over a network or transmitted across different JVMs. The byte stream can later be **de-serialized to recreate the object's state**. In other words, serialization is the mechanism of converting an object into a sequence of bytes, and deserialization is the reverse process of converting bytes back into an object.

Example:

```
public class A implements Serializable {  
    }  
    A a1= new A();  
    A a2;  
  
    File myfile= new File("filepath");  
  
    ObjectOutputStream oi;  
    ObjectInputStream oi2;  
  
    {  
        try {  
            // serialization:  
            oi = new ObjectOutputStream(new FileOutputStream(myfile));  
            oi.writeObject(a1);  
  
            // deserialization  
            oi2= new ObjectInputStream(new FileInputStream(myfile));  
            a2 = (A) oi2.readObject();  
        } catch (IOException | ClassNotFoundException e) {  
            throw new RuntimeException(e);  
        }  
    }  
}
```

Code Example n.4: Serialization, Hit Counter

If we want to build a hit counter that stores the number of visits of a website we could insert a “counter” variable in the servlet. However this approach has a serious problem: it works only if the server remains online.

Example of this “**Wrong approach**”:

Counter.java

```
public class Counter {
    int count = 0;
    Calendar timeStamp = Calendar.getInstance();

    public void increase() {
        count++;
        timeStamp = Calendar.getInstance();
    }

    @Override
    public String toString() {
        StringBuffer s = null;
        if (count == 0) s = new StringBuffer("<p>no hits yet</p>");
        else {
            s = new StringBuffer("<p>hits = ");
            s.append(count).append("<br>last hit on ");
            s.append(timeStamp.getTime().toString());
        }
        return s.toString();
    }
}
```

CounterServlet.java

```
@WebServlet(name = "CounterServlet", urlPatterns = {"/CounterServlet"})
public class CounterServlet extends HttpServlet {
    Counter counter = new Counter();

    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse
        response)
        throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        try (PrintWriter out = response.getWriter()) {
            request.getRequestDispatcher("/fragment1.html").include(request,
                response);
            counter.increase();
            out.println(counter);
            request.getRequestDispatcher("/fragment2.html").include(request,
                response);
        }
    }
}
```

Now let's see the **right approach** that uses serialization:

#### Counter.java

```
public class Counter implements Serializable {

    protected int count;
    protected Calendar timeStamp;
    private static final String FILE_PATH = "./data.ser";

    public Counter(){
        this.count=0;
        this.timeStamp=Calendar.getInstance();
    }

    public int getCount() {
        return count;
    }

    public void setCount(int count) {
        this.count = count;
    }

    public Calendar getTimeStamp() {
        return timeStamp;
    }

    public void setTimeStamp(Calendar timeStamp) {
        this.timeStamp = timeStamp;
    }

    public void increase() throws IOException, ClassNotFoundException {

        // update the Counter value and writes it in the file
        this.setCount(++count);
        this.setTimeStamp(Calendar.getInstance());

        ObjectOutputStream in= new ObjectOutputStream(new
            FileOutputStream(FILE_PATH));
        in.writeObject(this);
    }

    @Override
    public String toString() {
        StringBuffer s = null;
        if (count == 0) s = new StringBuffer("<p>no hits yet</p>");
        else {
            s = new StringBuffer("<p>hits = ");
            s.append(count).append("<br>last hit on ").
                append(timeStamp.getTime().toString());
        }
        return s.toString();
    }
}
```

### CounterServlet.java

```
@WebServlet(name = "CounterServlet", value = "/CounterServlet")
public class CounterServlet extends HttpServlet {
    private static final String FILE_PATH = "./data.ser";

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

        Counter obj;

        // Load the serialized object from the file
        File file = new File(FILE_PATH);

        if (file.exists() && file.length() > 0) {
            // File exists and is not empty, read the serialized object

            try (ObjectInputStream in = new ObjectInputStream(new
                FileInputStream(FILE_PATH))) {

                obj = (Counter) in.readObject();
                obj.increase();

            } catch (IOException | ClassNotFoundException e) {
                throw new ServletException("Error loading serialized object", e);
            }
        } else {
            // File does not exist or is empty, create a new serialized object with
            // default values
            try {
                obj = new Counter();
                obj.increase();
            } catch (ClassNotFoundException e) {
                throw new RuntimeException(e);
            }
        }
        try (PrintWriter out = response.getWriter()) {
            response.setContentType("text/html; charset=UTF-8");
            request.getRequestDispatcher("/fragment1.html");
            out.println(obj);
            request.getRequestDispatcher("/fragment2.html");
        }
    }
}
```

## Cookies

Http has a base problem: it's stateless. A possible solution is using hidden but this has different downsides:

- Works only if the user doesn't close the browser and follows the correct "pattern" of actions (if the back button is pressed all the information can be lost)
- The url length is limited
- It's not secure
- Requires a form

**Cookies** = A Cookie is a small amount of information **sent by a servlet** to a **Web browser**, **saved** by the **browser** and later sent back to the server.

A **cookie value** can **uniquely** identify a **client** → The **website** decides which information to **store** on the **client browser** and which in the **server side**. This decision is taken considering different aspects: for example storing data on the server is more secure, permits to share information across devices and to use session while storing data is faster and save some memory.

A cookie has: name, a single value and optional attributes such as comment, path and domain qualifiers, a maximum age, and a version number.

Cookies are placed on the header of the request and response.

Usually a browser support at least 20 cookies for each Web server, 200 in total and may limit the cookie size to 4 KB.

## Methods

-From **javax.servlet.http.HttpServletResponse**:

- `addCookie(Cookie Cookie)` → send cookie to the browser.

-From **javax.servlet.http.HttpServletRequest**:

- `getCookies()` → retrieve cookie from browser.

-From **javax.servlet.http.Cookie**:

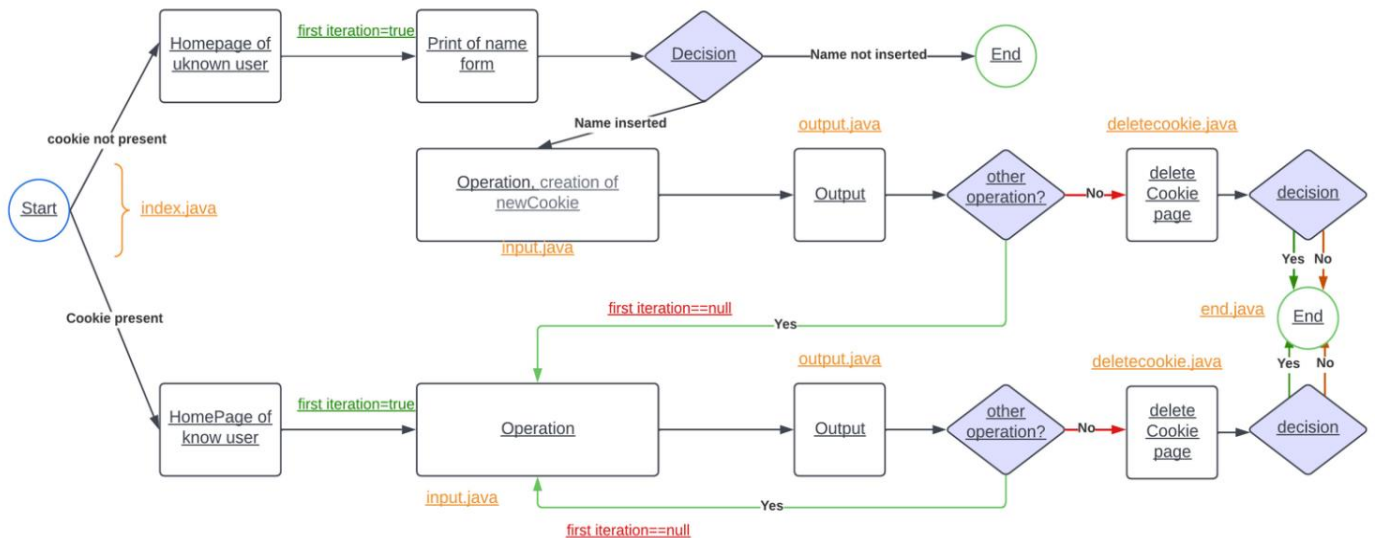
- `String getComment() / void setComment(String s)` → gets/sets a comment associated with this cookie.
- `String getDomain() / String setDomain()` → gets/sets the domain to which cookie applies. Normally cookies are returned only to the exact hostname that sent them. This method is used to instruct the browser to return them to other hosts within the same name. (The domain should start with a dot and must contain two dots for a non-country domains like .com, .edu.. and three dots for country domain like .co .us ...).
- `int getMaxAge() / void setMaxAge(int i)` → Gets/sets how much time (in seconds) should elapse before the cookie expires. If it's not set (or negative) the cookie lasts only for the current session and it will not be stored.

- String getName() /void setName(String s) → Gets/sets the name of the cookie. Since the get\_cookies method of HttpServletRequest returns an array of Cookies object, it is common to loop down on this array until you have the needed name, then check the value with getValue.
- String getValue / void setValue(String s) → Gets/sets the value associated with the cookie. In few cases a name is used a boolean flag and its value is ignore (the existence of the name means true).
- int getVersion() / void setVersion(int i) → Gets/sets the cookie protocol version this cookie complies with. Version 0, the default, adheres to the original Netscape specification. Version 1, not yet widely supported, adheres to RFC 2109.
- String getPath() / void setPath(String s) → Gets/sets the path to which this cookie applies. If the path is not specified the cookie is returned for all URLs in the same directory as the current page as well as all subdirectories. This method can be used to specify something more general. For example, someCookie.setPath("/") specifies that all pages on the server should receive the cookie. Note that the path specified must include the current directory.
- boolean getSecure / void setSecure(boolean b) → Gets/sets the boolean value indication where the cookie should only be sent over encrypted connections.



## Code Example n.5: Cookies, Calculator with cookies

This code is a revision of "Code example n.3" in which was implemented a Calculator, now with the usage of cookies. Let's see a the user flow:



```

@WebServlet(name = "index", value = "/index")
public class index extends Calculator {

    protected void process_request(HttpServletRequest request, HttpServletResponse
    response) throws ServletException, IOException {
        String name = getValueFromCookie(request, "name");
        /*
        two options:
        Cookie present -> go to the input
        Cookie not present -> get the name
        */

        if (name != null) {
            request.getRequestDispatcher("./input").forward(request, response);
        } else {
            response.setContentType("text/html");
            response.setCharacterEncoding("UTF-8");
            try (PrintWriter out = response.getWriter()) {
                request.getRequestDispatcher("./header.html").include(request,
                response);
                out.println("Hi! <br>");
                print_name_form(out);
                request.getRequestDispatcher("./footer.html").include(request,
                response);
            }
        }
    }
}

```

```

@WebServlet(name = "input", value = "/input")
public class input extends Calculator {

    protected void process_request(HttpServletRequest request, HttpServletResponse
    response) throws IOException, ServletException {
        String msg;
        boolean newUser = false;
        boolean first = false;

        String nameC = getValueFromCookie(request, "name");
        if (nameC == null) {
            /*
             Option 1: new user coming from the homepage page
             need to create the cookie
            */
            String nameP = getValueFromParameters(request, "name");
            newUser = true;
            Cookie newCookie = new Cookie("name", nameP);
            newCookie.setMaxAge(60 * 60 * 24 * 365);
            response.addCookie(newCookie);
            nameC = nameP;
        }
        /*
         Here i have to check if the user has arrived to the input from the homepage
         or if it is another operation.
        */
        String first_iteration = getValueFromParameters(request, "first_iteration");
        if (first_iteration == null) {
            first = true;
        }

        if (newUser && first) {
            // option 1
            msg = "<b>Nice to meet you, " + nameC + "!</b><br><br> Let's do an
            operation: <br><br>";
        } else if (!newUser && first) {
            // option 2, known user, coming back
            msg = "<b>Welcome Back, " + nameC + "!</b><br><br> Let's do an operation:
            <br><br>";
        } else {
            // option 3, known or new user, another operation
            msg = "<b> Let's Continue, " + nameC + "!<b><br><br> Insert new data:
            <br><br>";
        }
        response.setContentType("text/html");
        response.setCharacterEncoding("UTF-8");
        try (PrintWriter out = response.getWriter()) {
            request.getRequestDispatcher("./header.html").include(request, response);
            out.println(msg);
            print_input_form(out, nameC);
            request.getRequestDispatcher("./footer.html").include(request, response);
        }
    }
}

```

```

@WebServlet(name = "output", value = "/output")
public class output extends Calculator {
    @Override
    protected void process_request(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException, ParseException {
        String name = getValueFromCookie(request, "name");
        if (name == null) {
            /*
             * Actually the Cookie is already set even if the user is new so this
             * passage could be avoided.
             * However, it can happen that the user has the cookies disabled.
             */
            name = getValueFromParameters(request, "name");
        }
        response.setContentType("text/html");
        response.setCharacterEncoding("UTF-8");
        try (PrintWriter out = response.getWriter()) {
            request.getRequestDispatcher("./header.html").include(request,
response);
            String op1 = request.getParameter("op1");
            String op = request.getParameter("op");
            String op2 = request.getParameter("op2");
            out.println(compute_operation(op1, op, op2, name));
            out.println("Do you want to do another operation?<br>");
            print_continue_form(out, name);
            request.getRequestDispatcher("./footer.html").include(request,
response);
        }
    }
}

```

```

@WebServlet(name = "deleteCookies", value = "/deleteCookies")
public class deleteCookies extends Calculator {

    @Override
    protected void process_request(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
        String name = getValueFromCookie(request, "name");
        /*
         * As explained before, the name is passed as hidden and not only kept
         * in the cookie because they can be disabled
         */
        if (name == null) {
            name = getValueFromParameters(request, "name");
        }
        response.setContentType("text/html");
        response.setCharacterEncoding("UTF-8");
        try (PrintWriter out = response.getWriter()) {
            request.getRequestDispatcher("./header.html").include(request,
response);
            out.println("<b>Good, " + name + "! See you soon!!!</b><br><br>");
            out.println("Delete cookies?<br>");
            print_delete_cookies_form(out, name);
            request.getRequestDispatcher("./footer.html").include(request,
response);
        }
    }
}

```

```
@WebServlet(name = "end", value = "/end")
public class end extends Calculator {

    @Override
    protected void process_request(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException {
        String name = getValueFromCookie(request, "name");
        /*
         As explained before, the name is passed as hidden and not only kept
         in the cookie because they can be disabled
        */
        if (name == null) {
            name = getValueFromParameters(request, "name");
        }
        String ds = request.getParameter("ds");
        if (ds.equals("true")) {
            /*
             I have to destroy the cookies: iterate over all the cookies and
             add a cookie with the same name and expiration time=0
            */
            Cookie[] cookies = request.getCookies();
            for (Cookie c : cookies) {
                c.setMaxAge(0);
                response.addCookie(c);
            }
        }

        response.setContentType("text/html;charset=UTF-8");
        try (PrintWriter out = response.getWriter()) {
            request.getRequestDispatcher("./header.html").include(request,
                response);
            out.println("<b>See you next time, " + name + "!!!</b>");
            request.getRequestDispatcher("./footer.html").include(request,
                response);
        }
    }
}
```

```

public class Calculator extends HttpServlet {

    // COOKIES :
    protected String getValueFromCookie(HttpServletRequest request, String name) {
        Cookie[] cookies = request.getCookies();
        if (cookies != null) {
            for (Cookie c : cookies) {
                String x = c.getName();
                if (x.equals(name)) {
                    return c.getValue();
                }
            }
        }
        return null;
    }

    // PARAMS :
    protected String getValueFromParameters(HttpServletRequest request, String
name) {
        String x = request.getParameter(name);
        return x;
    }

    // FORMS :
    protected void print_name_form(PrintWriter out) {
        out.println("<form action=\"./input\" method = post>");
        out.println("<label for=\"name\">How should i call you?</label><br>");
        out.println("<input type=\"text\" id=\"name\" name=\"name\" required
><br><br>");
        out.println("<input type=\"submit\" value=\"Send\">");
        out.println("</form>");
    }

    protected void print_input_form(PrintWriter out, String name) {
        out.println("<form action=\"./output\" method = post>");
        out.println("<input type=\"hidden\" name=\"name\" value=\"" + name +
        "\"/>");
        out.println("<input type=\"hidden\" name=\"first_iteration\"
value=\"false\"/>");
        out.println("<label for=\"op1\">Operand 1:</label><br>");
        out.println("<input type=\"text\" id=\"op1\" name=\"op1\"
required><br><br>");
        out.println("<input type=\"radio\" id=\"add\" name=\"op\" value=\"+\"
required>");
        out.println("<label for=\"add\">+</label><br>");
        out.println("<input type=\"radio\" id=\"dif\" name=\"op\" value=\"-\">");
        out.println("<label for=\"dif\">-</label><br>");
        out.println("<input type=\"radio\" id=\"mul\" name=\"op\" value=\"*\">");
        out.println("<label for=\"mul\">*</label><br>");
        out.println("<input type=\"radio\" id=\"div\" name=\"op\" value=\"/\">");
        out.println("<label for=\"div\">/</label><br><br>");
        out.println("<label for=\"op2\">Operand 2:</label><br>");
        out.println("<input type=\"text\" id=\"op2\" name=\"op2\"
required><br><br>");
        out.println("<input type=\"submit\" value=\"Invia\">");
        out.println("</form>");
    }
}

```

```

protected void print_continue_form(PrintWriter out, String name) {
    out.println("<form method = post>");
    out.println("<input type=\"hidden\" name=\"name\" value=\"\" + name + \"\"/>");
    out.println("<input type=\"hidden\" name=\"first_iteration\" value=\"false\"/>");
    out.println("<input type=\"submit\" formaction=\"./input\" value=\"Yes\"/>");
    out.println("<input type=\"submit\" formaction=\"./deleteCookies\" value=\"No\"/>");
    out.println("</form>");
}

protected void print_delete_cookies_form(PrintWriter out, String name) {
    out.println("<form action=\"./end\" method = post>");
    out.println("<input type=\"hidden\" name=\"name\" value=\"\" + name + \"\"/>");
    out.println("<label> Yes <input type = \"radio\" name = \"ds\" value = \"true\" required> </label>");
    out.println("<label> No <input type = \"radio\" name = \"ds\" value = \"false\"> </label><br><br>");
    out.println("<input type=\"submit\" value=\"Send\"/>");
    out.println("</form>");
}

// DATA PROCESSING
protected String compute_operation(String op1, String op, String op2, String name)
throws ParseException, ParseException {
    String result = null;
    Number n1 = NumberFormat.getInstance().parse(op1);
    Number n2 = NumberFormat.getInstance().parse(op2);
    double res = 0;
    int err = 0;
    switch (op) {
        case "+":
            res = n1.doubleValue() + n2.doubleValue();
            break;
        case "-":
            res = n1.doubleValue() - n2.doubleValue();
            break;
        case "*":
            res = n1.doubleValue() * n2.doubleValue();
            break;
        case "/":
            if (n2.intValue() != 0) res = n1.doubleValue() / n2.doubleValue();
            else err = 1;
            break;
        default:
            err = 2;
    }
    switch (err) {
        case 0:
            result = "Great, " + name + "! <br><br> This is the result:<br>" + op1 + " " + op + " " + op2 + " = " + res + "<br><br>";
            break;
        case 1:
            result = "<b>Attention, " + name + ": Division by zero is not allowed!</b><br><br>";
            break;
        case 2:
            result = "<b>Attention, " + name + ": Operation not supported</b><br><br>";
            break;
    }
    return result;
}

```

```
protected void process_request(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException, ParseException {
    }

    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
        try {
            process_request(request, response);
        } catch (ParseException e) {
            throw new RuntimeException(e);
        }
    }

    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
        try {
            process_request(request, response);
        } catch (ParseException e) {
            throw new RuntimeException(e);
        }
    }
}
```

## Cookie Regulation

European law → <https://gdpr.eu/cookies/>

### How to classify Cookies:

- **Duration:**
  - **Session Cookies** → expire when the browser is closed (or session expires).
  - **Persistent Cookies** → saved on the browser, should not last longer than 12 months.
- **Provenance:**
  - **First-party Cookies** → Created by the website that is being visited
  - **Third-party Cookies** → Created by another entity like an advertiser or an analytic system
- **Purpose:**
  - **Technical Cookies** → Essential to brows the website and use the site. It's not required to obtain consent for these cookies, should be explained to the use what they do and why are necessary.
  - **Preferences/Functionality Cookies** → Allow the website to remember the choices made in the past by the user (like language, region, username and password). It's required to give information about the presence of these cookies.
  - **Statistics/Performance Cookies** → Collect information about how the user use the website (like which pages the user visits and what clicks). They are aggregated and so anonymized, their only purpose is to improve website functions. This includes cookies from third-party analytics services as long as the cookies are for the exclusive use of the owner of the website visited.
  - **Marketing cookies** → These cookies track the only activity of the user to help advertisers deliver more relevant advertising or to limit how many times the user see an ad. These cookies can share that information with other organizations or advertisers. These are persistent cookies and almost always of third-party provenance. They need to be explained and accepted.

When the users visits a website, the website should provide a summarized informative in the home page that contains a reference to the extended one accessible by the user.

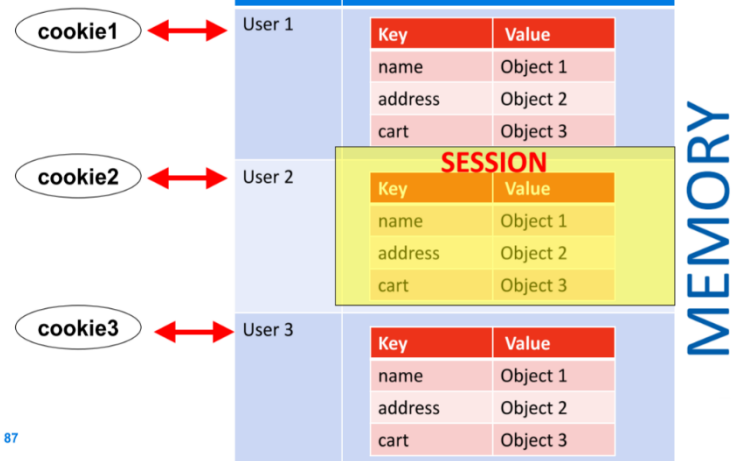


## Session

Another possible solution to the problem of the stateless behavior of http is the usage of sessions.

**Session** = logical sequence of HTTP requests and responses that are made by a user within a certain time period. The session is created when the user interacts with the server for the first time, and it is typically associated with a **specific user** by their **session ID**. Sessions allow the server to keep track of user data and state across multiple requests, and they can be used to **store information temporarily on the server** or pass information between different parts of an application.

The idea:



87

The server handles a map in which entry represents a distinct working session:

- The key of the entry is the session id
- The value of the entry is an object containing the information associated with that session

There are two possible **ways** to **implement session**:

- **Hidden type input**,
- **Cookies** → a special cookies called “session cookie” that are eliminated when the session expires. The server always add to the response this cookie that contains the session id.

Differences between cookies and sessions:

- Session are a server side solution
- Session can be cookies dependent
- Session ends when user closes the browser or logs out.
- A session has no limit of data storage

Methods from **HttpSession** :

- **HttpSession getSession(Boolean x)** → return the current session associated with this request.
  - X = true → if the session does not exist is created. same behavior as getSession().  
Creation of:
    - New id;
    - New session object;
    - Related entry in the session table ;
    - Session cookie to be put in the response.
  - X = false → if the session does not exist returns null.

- **public void setAttribute(String name, Object value)** → binds an object to this session, using the name specified. If an object of the same name is already bound to the session, the object is replaced.
- **public Object getAttribute(String name)** → Returns the object bound with the specified name in this session, or null if no object is bound under the name.
- **public Enumeration getAttributeNames()** → Returns an enumeration of String objects containing the names of all the objects bound to this session.
- **public void removeAttribute(String name)** → Removes the object bound with the specified name from this session. If the session does not have an object bound with the specified name, this method does nothing.
- **public long getCreationTime()** → Returns the time when this session was created, measured in milliseconds since midnight January 1, 1970 GMT.
- **public long getLastAccessedTime()** Returns the last time the client sent a request associated with this session, as the number of milliseconds since midnight January 1, 1970 GMT, and marked by the time the container received the request.
- **public void setMaxInactiveInterval(int interval)** → Specifies the time, in seconds, between client requests before the servlet container will invalidate this session. A negative time indicates the session should never timeout.
- **public int getMaxInactiveInterval()** → Returns the maximum time interval, in seconds, that the servlet container will keep this session open between client accesses. After this interval, the servlet container will invalidate the session.
- **public java.lang.String getId()** → Returns a string containing the unique identifier assigned to this session. The identifier is assigned by the servlet.
- **public boolean isNew()** → Returns true if the client does not yet know about the session or if the client chooses not to join the session (e.g., if client had disabled the use of cookies).
- **public void invalidate()** → Invalidates this session then unbinds any objects bound to it.

To **set a session global Timeout** → add this code in the web.xml:

```
<session-config>  
    <session-timeout> insert here the time in minutes (default=30) </session-timeout>  
</session-config>
```

Possible problem → **Cookies disabled**

Solution → URL rewriting:

- HttpServletResponse interface has the method `String encodeURL(String URL)` that takes the **original URL** as **parameters** and **rewrites** it **adding** the **session ID** **only if it's needed**. The first response will use both methods (cookie and URLencoded), the next response checks if the ID can be obtained from cookie or not.
- Anyway this solution has many downfalls:
  - It works only if the user navigates through the link on the response page (skipping with backward key ← to previous page).
  - Don't work with bookmarks.
  - The URL is directly written by the user.

## Concurrency

Some threads are concurrent if they are executed at the same time, the operating system can decide to interleave them in an order that cannot be directly known. For this reason it is important to understand that threads are **independent** meaning that they **don't communicate among** themselves and the behavior of a thread **does not generate any side effect impacting other threads**. If threads **require synchronization** then they are defined as **asynchronous**.

**Servlet** can be invoked by multiple threads (multiple requests) so they are **not automatically thread-safe**.

*For Example, the implementation of the HitCounter of code example n.4. was not thread-safe.*

Threads mainly communicate in two ways:

- Sharing memory
- Exchanging messages

This can lead to two possible issues:

- Thread interference → multiple thread access and modify shared data.
- Memory consistency errors → inconsistent views of shared memory

### How to implement synchronization:

- **Synchronized methods** → add the `synchronized` keyword to the method declaration.  
It is not possible for two invocations of synchronized methods on the same objects to interleave. If a thread is executing a synchronized method for an object all other threads that invoke synch. methods for the same object suspend execution.
- **Synchronized statements** (blocks) → add `synchronized (this) {synchronized part}`.  
A synchronized block is used to lock an object for any shared resource. It is useful to synchronize only a part of a method.

A session belongs to a user, when different users activate the same servlet and this requests a session object it gets a different object for every user so no problems with multithreading. Anyway it can be a problem if a user opens two windows on the same browser and accesses the same servlet.

## Sharing information between servlets: Servlet Context

Information can be shared between the same request with the **include** and **forward** mechanism (setAttribute to the request), instead among different requests by the same user **session object** can be used.

There is another possibility→

**ServletContext** = object containing meta information on the web application. It has **attributes** that are **available to all servlets** in the application, between request and sessions. These attributes are stored in the memory of the servlet container.

Main methods/Usage:

- `context.setAttribute(String, Object)`
- `context.getAttribute(String)`
- `getServletConfig().getServletContext()`

Among different invocations of:

- Same servlet → use instance variables or static variables.
- Different servlets → use Servlet Context.

Lifetime:

- 1) **Request** → start with the HTTP request, ends when the interaction with the client is closed.
- 2) **Session** → more than one request, depends on the implementation.
- 3) **Servlet context** → from deploy to undeploy.

See: <https://stackoverflow.com/questions/3106452/how-do-servlets-work-instantiation-sessions-shared-variables-and-multithreadi>.

Rules:

- 1) `doGet()`, `doPost()`, `doXXX()` methods **should not update or modify instance variables** as instance variables are share by all threads of the same instance. Explanation from the link before:

“That said, your major concern is possibly **thread safety**. You should now know that servlets and filters are shared among all requests. That's the nice thing about Java, it's multithreaded and different threads (read: HTTP requests) can make use of the same instance. It would otherwise be too expensive to recreate, `init()` and `destroy()` them for every single request. You should also realize that you should **never** assign any request or session scoped data as an *instance* variable of a servlet or filter. It will be shared among all other requests in other sessions. That's **not** thread-safe! The below example illustrates this:”

```
public class ExampleServlet extends HttpServlet {  
  
    private Object thisIsNOTThreadSafe;  
  
    protected void doGet(HttpServletRequest request, HttpServletResponse  
        response)  
        throws ServletException, IOException {  
        Object thisIsThreadSafe;  
  
        thisIsNOTThreadSafe = request.getParameter("foo"); // BAD!! Shared  
            among all requests!  
        thisIsThreadSafe = request.getParameter("foo"); // OK, this is thread  
            safe.  
    }  
}
```

- 2) If you have a requirement which requires modification of instance variable then do it through synchronization.
- 3) Above two rules are applicable for **static variables** also because they are also shared.
- 4) **Local variables** are always **thread safe** (unless they refer to global objects).
- 5) The **request** and **response** objects are **thread safe** because new instance of these are created for every request into the servlet, and thus for every thread executing in the servlet.

## Associating events with session objects

Session listener objects are notified of certain events related to their association with a session such as the following:

- When the object is added or removed from a session. To receive this notification, your object must implement the `javax.servlet.http.HttpSessionBindingListener` interface.
- When the session to which the object is attached will be **passivated** or **activated**. A session will be passivated\* or activated when it is **moved between virtual machines** or **saved to and restored from persistent storage**. to receive this notification, your object must implement the `javax.servlet.http.HttpSessionActivationListener` interface.

\* Passivation is used to freeze sessions saving them on the disk. This is done in two cases:

- Container needs to create a new session but the number of currently active is at the limit.
- Session is not being used (but still valid).

In both case this operation saves memory. When the user performs some action the session is activated.

Configuration of passivation in web.xml:

```
<max-active-session>30</max-active-sessions>      (-1 = no limit)
<passivation-config>

    <use-session-passivation>true</use-session-passivation>

    <passivation-min-idle-time>60</passivation-min-idle-time>
    //minimum inactive time after which a session can be
    passivated. (cfr case 1)

    <passivation-max-idle-time>300</passivation-max-idle-time>
    //max inactive time after which the session is passivated
    (cfr case 2)

</passivation-config>
```

## Jsp

**JSP** = **Java Servlet Pages**. Server side scripting Java-like **HTML embedded**. It is usually used with the java servlet and “solves” the problem of printing directly HTML.

Example:

### file.jsp

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

<html>

    <body>

        <% Calendar x= Calendar.getInstance() %>

        <%= x %>

    </body>

</html>
```

The framework (in this case Tomcat) has a container (in this case Jasper) that takes care of managing jsp files and turns it into a servlet-like code:

```
import java.util.Calendar;           //<%@ directives %>

@WebServlet(name = "helloServlet", value = "/hello-servlet")
public class SimpleServlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse
response) throws IOException {
        PrintWriter out = response.getWriter();
        response.setContentType("text/html");
        out.println("<html><body>");
        Calendar x = Calendar.getInstance();           //<% scriptLes%>
        out.println(x);                                //<%= expressions %>
        out.println("</body></html>");
        out.close();
    }
}
```

NB: this is not actually the result of the translation but it does the same thing. A more complete view of these passages is provided in the next chapters.

## JSP Elements

JSP elements help the developer to write the Java code within the tags.

**The elements are divided in two categories and an “extra” one:**

- **Scripting Elements**
  - **Comment tag**: `<% -- JSP Comment -- %>`
  - **Expression tag**: `<%= expression %>`. They can be logical, arithmetical.. The result is converted in String and printed.
  - **Scriptlet tag**: `<% code snippet %>`. Everything inside the scriptlet is Java code, and goes in the service, so local variables of the thread.
  - **Declaration tag**: `<#! Declaration; [declaration;] +... %>`. Java code in which a variable or method is declared. The declaration can be written in every position of the code but when are translated are put in the top of the servlet. These are **instance variables**. Initialized when the JSP is initialized.
- **Directives** → Syntax: `<%@ directive type attribute="value" %>`

Are used to provide messages to the container about how to manage the client requests while converting a JSP page into a servlet.

**Three kinds of directives:**

- **Page**: defines **global information** about the whole JSP page that contains it. The settings it changes directly affect the page's compilation.  
Main Attributes:
  - `<%@ page import= "java.awt.*", "java.util.*" %>`. Specify the list of packages
  - `<%@ page language= "java", session=true %>`. The original idea was that JSP could “use” other languages, actually support only java. Session specify if jsp participates to the session, default is true. If there is no session is created.
  - `<%@ page errorPage=URL %>`.
  - `<%@ page isErrorPage=true %>`.
  - `<% extends= "className" %>`.
  - `<% contentType= "text/html"%>`.
  - `<% isThreadSafe=true %>`. Specifies if the page supports multi-threading or not, the default is true.
- **Include**: It is used to **include one file** in a JSP page at the **translation step**. Generally files are **static** pages such as HTML, text... Eg: header, footer...
- **Taglib**: declares that the JSP page uses **personalized tags** and reports the URL of where they are defined.



- **Actions** → Syntax: `<jsp:action_name attribute= "attribute_value">`
  - `<jsp:include page="URL"/>`. For including **static** or **dynamic** resources at **request** time.
  - `<jsp:forward page="URL" />`
  - `<jsp:UseBean`
    - `id="instanceName"`
    - `scope= "page | request | session | application"`
    - `class= "packageName.className"`
    - `type= "packageName.className"`
    - `beanName= "packageName.className | <%= expression>"`
  - `</jsp:useBean>`. Load external java class, but if the objects is already present is only referenced (different from import).

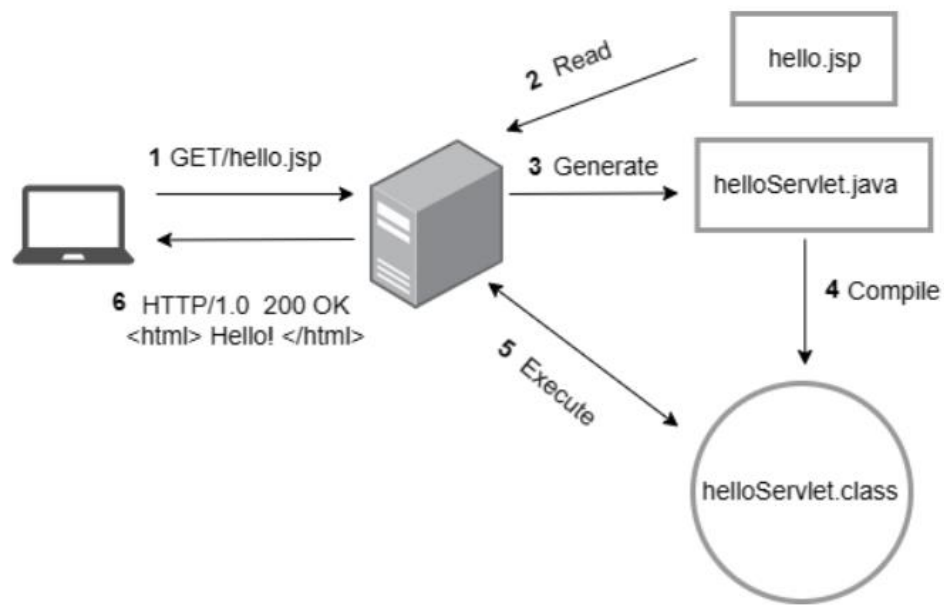
### Example 1: Date and Time

#### index.jsp

```
<%@ page import="java.time.format.DateTimeFormatter" %>
<%@ page import="java.time.LocalDateTime" %>
<%@ page contentType="text/html; charset=UTF-8"%>
<!DOCTYPE html>
<html>
<head>
  <title>What time is it?</title>
</head>
<body>
<%
  DateTimeFormatter day = DateTimeFormatter.ofPattern("dd/MM/yyyy");
  DateTimeFormatter time = DateTimeFormatter.ofPattern("HH:mm:ss");
  LocalDateTime now = LocalDateTime.now();
%>
<p>Hello, I am a jsp script</p>
<p> Today is
  <%= day.format(now) %>
</p>
<p> and time is
  <%= time.format(now) %>
</p>
</body>
</html>
```

In the project the .jsp file are placed in the same position of the HTML files.

## JSP Processing and Lifecycle



There is a **performance degradation** compared to servlets, the servlets where already compiled, **Jsp** needs to be **translated** and **compiled**. Usually the solution is that **Jasper search** if there's already a **servlet corresponding** to the Jsp file compiled before the last modification of the Jsp file. This solution is very important because it permits to run the servlet once before the actual deployment in order to compile them, avoiding an excessive amount of load for the first user.

### JSP LifeCycle:



The last 3 passages corresponds to the actions of the servlets:

- `jspInit()` → initialization actions.
- `_jspService()` → selects between `doGet()`, `doPost()`...
- `jspDestroy()` → removes jsp from container.

## Request, Response and Sessions in JSP

There are 0 so called “Implicit Objects” or predefined variables provided by the JSP container (Jasper) to avoid declaring them explicitly:

Object	Represents
<b>out</b>	Writer
<b>request</b>	HttpServletRequest
<b>response</b>	HttpServletResponse
<b>session</b>	HttpSession
<b>page</b>	<i>inside the Servlet, instance variables</i>
<b>application</b>	servlet.getServletContext
<b>config</b>	ServletConfig
<b>exception</b>	<i>only in an ErrorPage</i>
<b>pageContext</b>	

Example:

```
<%@ page errorPage="errorpage.jsp" %>
<!DOCTYPE html>
<html>
<head>
  <title>Example</title>
</head>
<body>
<%
  // Get the User's ame fromt the request;
  out.println("<b>Hello:" + request.getParameter("user")+"</b>");
%>

</body>
</html>
```

Nb: implementing only with jsp means that the doGet(), doPost().. are now used implicitly.

## Best Practices

1. **Don't overuse Java code in HTML pages.**
2. **Choose the right include mechanism:**
  - Static data such as headers, footers, is best kept in separate files and not regenerated dynamically.
  - Once such content is in separate files, they can be included using one of the following include mechanisms:
    - `<% @ include file="filename" %>`
    - `<jsp: include page = "page.jsp" />`

3. **Don't mix business logic with presentation.**
  - JSP code should be limited to front-end presentation.
4. **Use filters if necessary** (See after).
5. **Use a database for persistent information** (See after).

## Java Beans

**Java bean** = It's a java class (POJO) that:

- Provides a public no-argument constructor
- Implements java.io.Serializable
- Has set/get methods for properties
- Is thread safe/security conscious

Example:

```
public class SimpleBean implements Serializable{
    private int counter;
    SimpleBean(){counter=0;}
    int getCounter() {return counter;}
    void setCounter(int c) {counter=c;}
}
```

Since the class implements Serializable the class **can be passivated saving the status**. When is **activated** the first step is to **create** a new **instantiation** with the **default constructor** and then insert the “old” data stored in the persistent memory with **getters** and **setters**.

### Standard actions involving beans:

- **useBean** → `<jsp:useBean id="id" class="bean's class" scope="bean's scope">`. Associates an instance of a java Bean defined in a scope with an ID to a script variable of the JSP with the same ID. If the bean does not exist is created.
- **setProperty** → `<jsp:setProperty name="bean's id" property="property name" value="value"/>`
- **getProperty** → `<jsp:getProperty name="bean's id" property="property name" />`

## Code Example: JavaBean

### BeanOne.java

```
import java.io.Serializable;

public class BeanOne implements Serializable {
    private String name;
    private String surname;
    public BeanOne(){};

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getSurname() {
        return surname;
    }

    public void setSurname(String surname) {
        this.surname = surname;
    }

    @Override
    public String toString() {
        return "BeanOne{" +
            "name='" + name + '\'' +
            ", surname='" + surname + '\'' +
            '}';
    }
}
```

### index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>BeanProject</title>
</head>
<body>
<br/>
<a href="jsp0ne.jsp">jsp0ne</a><br/>
<a href="jspTwo.jsp">jspTwo</a><br/>
<a href="addZ">add Z</a><br/>
<a href="InvalidateServlet">Invalidate session</a><br/>
</body>
</html>
```

### jspOne.jsp

```
<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" >
    <title>JSP PageOne</title>
</head>
<body>
<h1>Setting the property...</h1>
<jsp:useBean id="myBean1" class="com.example.beanproject.BeanOne" scope="session"/>
<jsp:setProperty name="myBean1" property="surname" value="Damon"/>
<jsp:setProperty name="myBean" property="name" value="Matt"/>
<p><%= myBean1.toString()%></p>
<hr>
<a href="index.html"> Home </a>
</body>
</html>
```

The first JSB instruction (useBean) is related to a non-existing **bean** that is therefore **instantiated**; the **session** value is as default true so also the session is being **created**.

The other two instructions are **setting** two **properties** and the third one is **printing** the bean.

### jspTwo.jsp

```
<%@ page contentType="text/html; charset=UTF-8" %>
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" >
    <title>JSP PageTwo</title>
</head>
<body>
    <h1>Gettind the property...</h1>
    <jsp:useBean id="myBean1" class="com.example.beanproject.BeanOne"
        scope="session"/>
    <jsp:getProperty name="myBean1" property="surname" />
    <jsp:getProperty name="myBean" property="name" />
    <p><%= myBean1.toString()%></p>
    <hr>
    <a href="index.html"> Home </a>
</body>
</html>
```

Now the result of this code is depending on the usage's flow: if the user has already visited jspOne than the bean is already existing, the code retrieves it with useBean, gets and prints the properties. If the users arrives to jspTwo before the jspOne the bean is instantiated but the default constructor doesn't sets name and surname so when the two getProperty function are called they return null.

addZ.java

```

@WebServlet(name = "addZservlet", value = "/addZ")
public class addZ extends HttpServlet {
    protected void doGet(HttpServletRequest request, HttpServletResponse
        response) throws ServletException, IOException {
        HttpSession session = request.getSession();
        if (session.getAttribute("myBean1") == null) {
            session.setAttribute("myBean", new BeanOne());
            // if bean not found create a new one with the default
            // constructor
        }
        BeanOne aBean = (BeanOne) session.getAttribute("myBean1");
        aBean.setName(aBean.getName() + "z"); // adding a "z" to the name
        request.getRequestDispatcher("jspTwo.jsp").forward(request,
        response); // forward control to jspTwo
    }
}

```

Mind the scope →

JSP	Servlet
<jsp:useBean id="myBean1" class=com.example.beanproject.BeanOne" scope="session"/>	BeanOne x = (BeanOne)session.getAttribute("myBean1");
<jsp:useBean id="myBean1" class=com.example.beanproject.BeanOne" scope="application"/>	BeanOne x = (BeanOne)context.getAttribute("myBean1");
<jsp:useBean id="myBean1" class=com.example.beanproject.BeanOne" scope="request"/>	BeanOne x = (BeanOne)request.getAttribute("myBean1");

InvalidateServlet.java

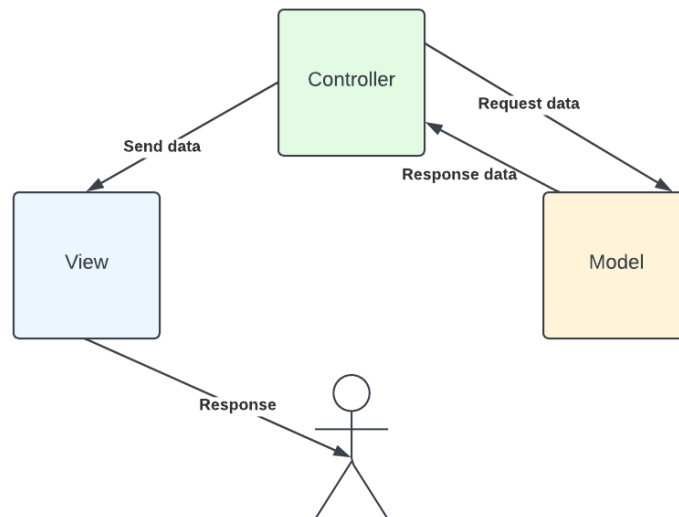
```

@WebServlet(name = "InvalidateServlet", value = "/InvalidateServlet")
public class InvalidateServlet extends HttpServlet {
    protected void doGet(HttpServletRequest request, HttpServletResponse
        response){
        HttpSession session = request.getSession(false);
        // false avoids to create a new session if there isn't one
        if(session != null) {
            session.invalidate();
        }
    }
}

```

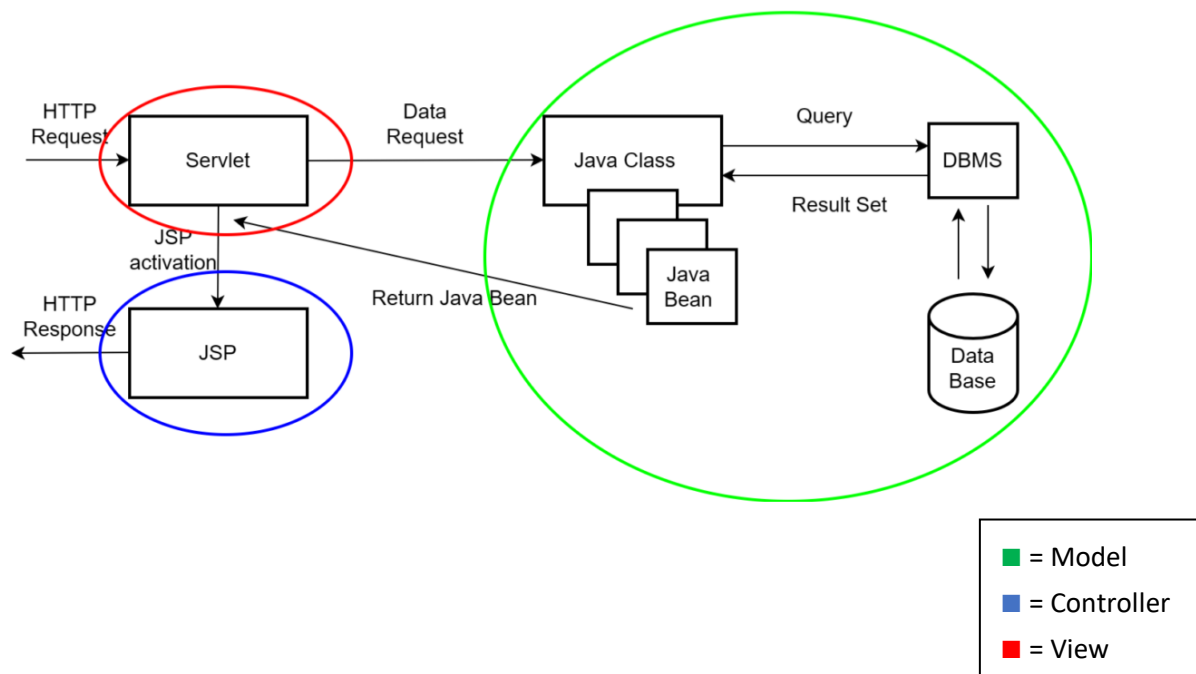
## MVC pattern

**MVC** = model view controller → Objective: **separate business logic from presentation.**



- **Model** = representation and elaboration of data.
- **Controller** = manager of the business logic, receives commands from the user through the View and interacts with data thanks to the Model.
- **View** = presentation layer. Presents data to the user, specifying the modality. Usually manages also the input.

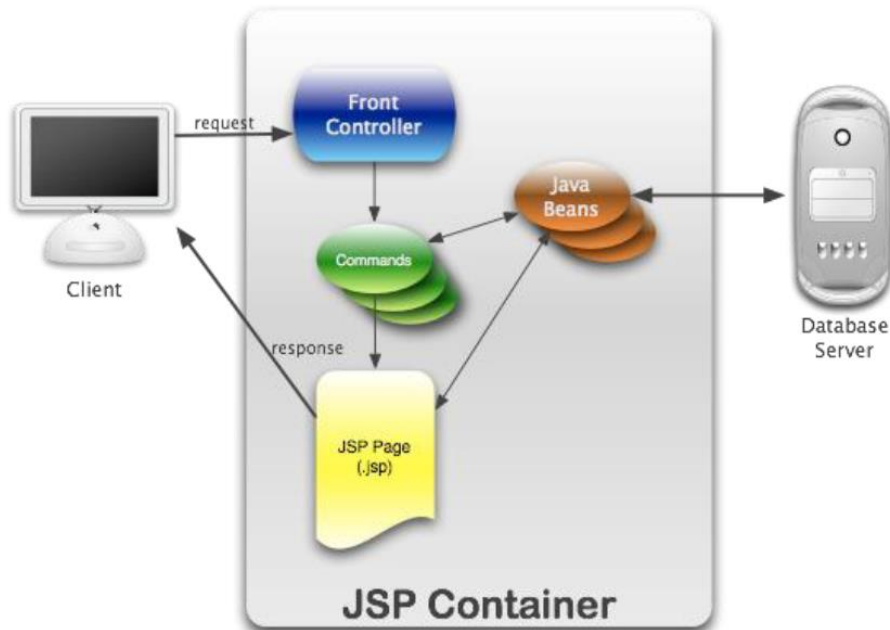
This **theoretical** design pattern can be **translated** into different “**practical**” implementations such as the **server centric approach**:





The Http request arrives to the **controller** (implemented as one or more Servlets), if data is needed the **model** retrieves/stores data in a database. The data is placed and managed through java classes, usually JavaBeans. The JSP manages the **view**.

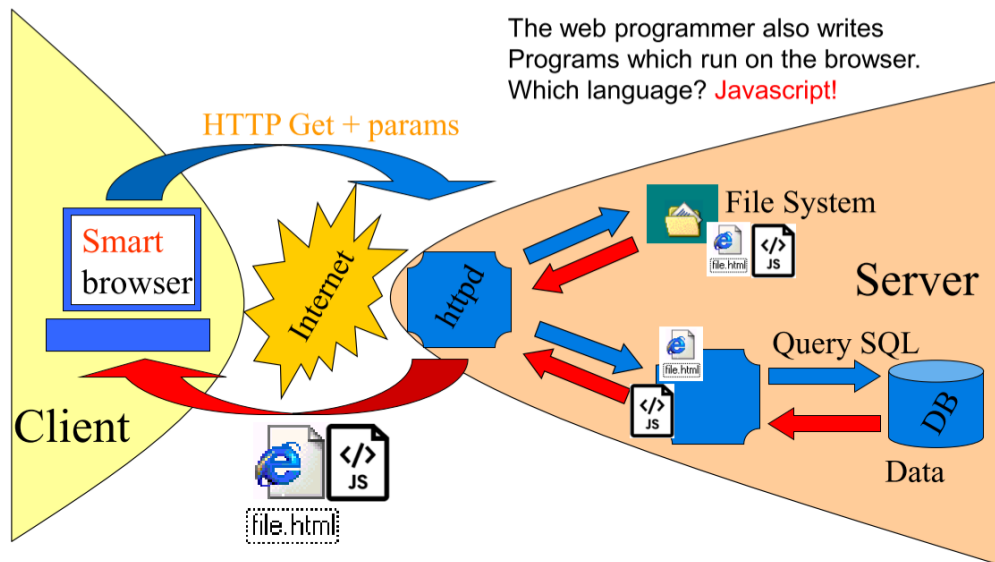
There are other many patterns like the **front controller pattern**:



It is more complex: there is only one servlet called “front controller”, when a request arrives the front controller decides which command (java class) has to handle the request. The interaction with data is similar to the previous one.

In general the design patterns are universal and simplify the work.

## Javascript



Javascript History → see

- [https://www.w3schools.com/js/js\\_history.asp#:~:text=JavaScript%20was%20invented%20by%20Brendan,latest%20version%20was%201.8.5.](https://www.w3schools.com/js/js_history.asp#:~:text=JavaScript%20was%20invented%20by%20Brendan,latest%20version%20was%201.8.5.)
- <https://dev.to/dboatengx/history-of-javascript-how-it-all-began-92a> .

EcmaScript engine = program that executes source code written in a version of the ECMAScript language standard.

Example:

```
<html>
<body>
<div onmouseover="this.style.background='cyan'"> this is a
text </div>
</body>
</html>
```



this is a text



this is a text

**JavaScript adds dynamicity on server Side!**

Useful test editors online:

- <https://codepen.io/trending>
- [https://www.w3schools.com/js/tryit.asp?filename=tryjs\\_myfirst](https://www.w3schools.com/js/tryit.asp?filename=tryjs_myfirst)

## Basics

### Events

**JavaScript is event based**, one of the most important category is represented by “UiEvents”:

- DocusEvent
- InputEvent
- KeyboardEvent
- MouseEvent
- TouchEvent
- WheelEvent

**Listeners** are objects that are notified by the operating systems for specific events. The **handler** manages what to do as **action** in **response** of the **specific event** that occurred. “onmouseover” is a pre-defined handler.

### Javascript and HTML

There are three way to use Javascript in HTML (as it was for CSS):

- **Embedding code:** `<script>... </script>`
- **Inline code:** adding the code in the tag
- **External file:** `<script src="file.js" > </script>`

### Syntax

C like but without pointers, case-sensitives.

### Operators

- **Standard ones:**
  - + - \* /
- **Mathematical/Assignment:**
  - \*\*= →  $a=a^3$
- **String operators:**
  - + → concatenation
- **Comparison operators:**
  - === → equal value and equal type
  - !== → not equal value or not equal type
- **Logical and bitwise operators:** standard
- **Type operators:**
  - typeof → returns the type of a variable
  - instanceof → returns true if an object is an instance of an object type

## Data Types

JavaScript is dynamic typed and there is no need of explicit declaration of variables.

- Primitive data types:
  - number → internally floating point.
  - string
  - boolean
  - null
  - undefined → when a variable is not initialized.
- Complex data types:
  - object (Number, String..)
  - function

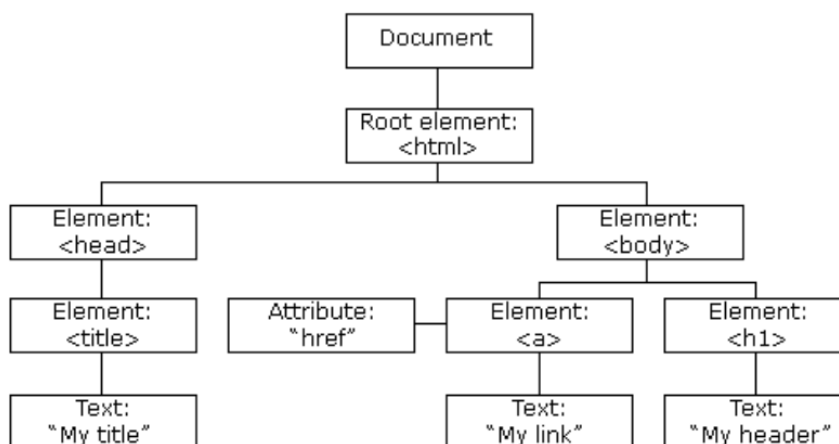
```
typeof(x) → undefined
var x=5;
typeof(x) → number
x="John"
typeof(x) → string
```

## DOM

Javascript has **Objects**, objects have variables and methods. They can be printed: in such case they use their customized toString() method, or give a generic indication such as [object HTMLDivElement]

Some Javascript Objects represent fragments of an HTML document. The collection of these objects represent the whole page. Such representation is called Document Object Model.

When a web page is loaded, the browser creates a Document Object Model of the page. The **HTML DOM model** is constructed as a **tree** of Objects.



With the **object** model, JavaScript gets all the power it needs to create dynamic HTML.

The HTML DOM is a standard object model and programming interface for HTML. It defines:

- The HTML elements as objects
- The properties of all HTML elements
- The methods to access all HTML elements
- The events for all HTML elements

In other words: The **HTML DOM** is a standard for how to get, change, add, or delete HTML elements.

## JavaScript HTML DOM Document\*

The **HTML DOM document object** is the **owner** of all other **objects** in the **web page**. If you want to **access any element in an HTML page**, you always start with **accessing** the **document** object.

### Finding HTML elements

<code>document.getElementById(id)</code>	Find an element by element id
<code>document.getElementsByTagName(name)</code>	Find elements by tag name
<code>document.getElementsByClassName(name)</code>	Find elements by class name
<code>document.querySelectorAll(selector)</code>	Find elements that match a specific Css selector

It is also possible to get elements belonging to a specific **collection**. the collections are:

anchors, body, documentElement, embeds, forms, head, images, links, scripts, title.

### Changing HTML elements

<code>element.innerHTML = new html content</code>	Change the inner HTML of an element
<code>element.attribute = new value</code>	Change the attribute value of an HTML element.
<code>element.style.property = new style</code>	Change the style of an HTML element
<code>element.setAttribute(attribute, value)</code>	Change the attribute value of an HTML element

These operations can be done in two ways:

- Two steps:
  - 1) Assign the HTML element to a variable
  - 2) Modify the element

Eg:

```
const x = document.getElementById("id01");  
x.innerHTML = "New Heading";
```

- Directly:

```
document.getElementById("id01").innerHTML = "New Heading"
```

Also attributes can be changed. Eg: `document.getElementById("myImage").src = "landscape.jpg";`

### Adding/Deleting

<code>document.createElement(element)</code>	Create an HTML element
<code>document.removeChild(element)</code>	Remove an HTML element
<code>document.appendChild(element)</code>	Add an HTML element
<code>document.replaceChild(new, old)</code>	Replace an HTML element
<code>document.write(text)</code>	Write into the HTML output stream

## Input, Output

The Browser Object Model (BOM) allows JavaScript to directly interact with the browser. Although there are no official standards it is widely used. One of the most important object is the **window**.

### **There are 3 main ways to output information with JS:**

- **document.write()**: write the information like an HTML code.
- **window.alert()**: “pop-up” alert box.

```
<body>
<div onmouseover="window.alert(this.*****);">
  This element contains
  <code>code</code>, <span style="visibility:hidden">hidden information,
</span>
  and <strong>strong language</strong>
.</div>
</body>
```

\*\*\*\*\* can have 3 values:

- innerHTML → Full HTML code
  - innerText → Text only, css aware
  - textContent → Text only, css aware
- **console.log()**: writes in the browser console, not a user feed.

## Strings

In JS strings can be:

- **string** → primitive data type  
var name1= “John”
- **String** → object  
var name2= new String(“John”)

Therefore, using the previous example we can understand that:

- name1 == name2 → **true**
- name1 === name2 → **false**

### **Operators:**

- “+” → concatenates strings.
- “>” and “<” → compares the strings alphabetically.

### **Main methods:**

charAt(0), indexOf(substring), lastIndexOf(substring), charCodeAt(n), fromCharCode(value,...)  
concat(value,...), slice(start,end), toLowerCase(), toUpperCase(), replace(regex,string), search(regex).

## “+” Operator

The operator “+” is overloaded for **two different operations**:

- **Numeric addition**
- **Concatenation of strings.**

### When a “+” is evaluated:

- 1) The operands are **converted** to their **primitive data**
- 2) The types of operands are checked:
  - If an operand is a **string**, the other **operand** is converted to a **string** and they are concatenated.
  - Otherwise, both the operands are converted to numbers, and numeric addition is performed.

So: number+ number+ string=string ; but number\*string = number.

<b>x</b>	<b>y</b>	<b>x+y</b>
1	2	3
"1"	2	12
1	null	1
"1"	null	1null
1	undefined	NaN
"1"	undefined	1undefined
1	true	2
"1"	true	1true
false	true	1
true	null	1
true	undefined	NaN
null	null	0
null	undefined	NaN

It can be used as an **unary operator** to **convert strings** to **numbers**.

*Eg:*

```
var y="5"; // y is a string
var x = + y; // x is a number
```



## Functions

Functions are **defined** with the **function keyword**. Functions can be written in three ways:

- **Function declarations:**

```
function functionName(param1, param2..){  
    operations  
    return something;  
}  
Usage: let z=x(4,3)
```

Nb: in JS the semicolons are used to separate executable statement, so it's not common to end a function with a semicolon.

- **Function expressions :**

```
const x = function (param1, param2..){  
    operations  
    return something;  
};  
Usage: let z=x(4,3)
```

Nb1: given the notes above is clear that in these cases the semicolon is needed.

Nb2: this type of functions are called "*anonymous functions*".

N3: there are different equivalent syntax to the previous one :

- `const x= (param1, param2) => { operations, return something;}`
- `const x= param1=> something;`

NB: parenthesis can be removed if there is only one parameter and the return if the function has only one instruction.

- **Function constructor :**

```
const x = new Function("param1", "param2", "operations; return  
    something");
```

Avoid using this approach if possible.

### General notes about the functions:

- Parameters' data types are not needed.
- Type checking on parameters is not done.
- Parameters' number is not checked. → a function called with missing arguments sets the missing ones to undefined.
- Parameters can have default values

## Hoisting

Hoisting is JavaScript's default behavior of **moving declarations to the top of the current scope** (the assignments are left in place). Hoisting applies to **variable declarations** and to **function declarations**.

What does this mean implies? That in this code:

```
Ex1.js
x=3;
doSomething();
let x;
function doSomething(){
    return console.log("Hi!");
}
```

There is no problem that *x* is used before the declaration or that *doSomething* is called before it's declaration. This is because in JS **declarations are moved to the top of their scope**.

So the previous code is actually the same as:

```
Ex2.js
let x;
function doSomething(){
    return console.log("Hi!");
}
x=3;
doSomething();
```

In "Ex1.js" the function *doSomething* is **hoisted**.

An important note is that **function expressions are not hoisted**.

## Special functions properties\*

- **Rest parameter** → Allows a function to treat an indefinite number of arguments as an array. Eg:

```
function sum(...args) {
    let sum = 0;
    for (let arg of args) sum += arg;
    return sum;
}
let x = sum(4, 9, 16, 25, 29, 100, 66, 77);
```

- **Arguments Object** → Similarly to the previous one, JS has an a built-in object called the arguments object. The argument object contains an array of the arguments used when the function was called (invoked). Eg:

```
function findMax() {
    let max = -Infinity;
    for(let i = 0; i < arguments.length; i++)
    {
        if (arguments[i] > max) {
            max = arguments[i];
        }
    }
    return max;
}
let x=findMax(3, 4, 4);
```

## Array methods

**Mapping** → Syntax: `array.map(function)`. Eg:

```
a=[1,2,3,4];  
let x=a.map(x=>x*2);  
  
// result: x=[2,4,6,8]
```

**Filter** → Syntax: `array.map(function)`. Eg:

```
a=[1,2,3,4];  
let x=a.filter(x => x%2==0);  
  
// result: x=[2,4,6,8]
```

## Variables

Variables declared without a value has the value equal to “undefined”.

A **ReferenceError** is thrown when trying to **access** an **undeclared variable** but **not** if the value is **undefined**.

Eg:

```
let x;
console.log(x);    // output=undefined
console.log(y);    // Reference Error
```

### Variable scopes:

Type of declaration	Scope
<i>variableName = value</i>	<b>global</b> : visible inside the block, inside the sub-blocks and outside the block
<u>var</u> <i>variableName = value</i>	<b>outside a function</b> → <b>global</b> , visible in the block, inside the sub-block and outside the block <b>inside a function</b> → <b>function scope</b> : visible only inside the function, not outside
<u>let</u> <i>variableName = value</i>	<b>block scope</b>

<pre>{   x=4;   var y=5;   let z=6; } console.log(x);    // output=4 console.log(y);    // output=5 console.log(z);    // Reference Error</pre>	<pre>function doSomething(){   x = 4;   var y = 5;   let z = 6; } doSomething(); console.log(x);    // output=4 console.log(y);    // Reference Error console.log(z);    // Reference Error</pre>
---	---

NB: In the second example if the function is not called then also the first `console.log(x)` would throw the same error.

### Variable redefinition:

- The redefinition follows the same rules of the scopes. If the redefinition augments the visibility than is “applied” otherwise is not.

```
var x = 10;
// Here x is 10

{
  x = 2;
  // Here x is 2
}

// Here x is 2
```

```
var x = 10;
// Here x is 10

{
  var x = 2;
  // Here x is 2
}

// Here x is 2
```

```
var x = 10;
// Here x is 10

{
  let x = 2;
  // Here x is 2
}

// Here x is 10
```

**Function scope:** follows the same rules

## Objects

In java everything is an **object**, even the classes are just a function object.

JavaScript objects are collections of named values (properties), they can contain also methods. There fore they can be thought as maps. *E.g.*:

```
var person = { firstName: "Mark", lastName: "Buffalo",
  fullName: function(){
    return this.firstName + " " + this.lastName;}
}
```

### How to access to properties:

- **object.property** → person.firstName
- **object[property]** → person["firstName"]
- **var x = property; object[x]** → var x="firstName"; person[x]

Properties can be added or deleted dynamically. *E.g.*:

- add → person.birthplace="Rome";
- delete → delete person.firstName;

### Object() method and defineProperty

Another method to create an Object is to use **the Object()** method and the **defineProperty**. **defineProperty** takes as parameter an **object**, a **property name** and a **descriptor**. The descriptor can contain different attributes such as:

- **value** → value of the property, If not specified than is set to undefined.
- **writable** → if true the value can be rewritten
- **enumerable** → if true it can enumerate the properties.

```
var object1 = new Object();
Object.defineProperty(
  object1, 'name',
  {value: "AA", writable: false}
);
object.name=77;
console.log(object1.name);

→ output = AA
```

There is a variable: **Object.defineProperty** that allows to defines different properties at the same time, Syntax →

```
Object.defineProperty(obj1, {property1:{ attribute1, attribute2,...}
property2:{ attribute1, attribute2,...}..})
```

## Object constructors

Object constructors: **templates** for **creating objects** of a certain type. Eg:

```
function Person(n,s){
  this.name=n;
  this.surname=s;
  this.fullName=function(){return this.name +
  " " + this.surname}
}

a=new Person("Mike", "Sullivan");
// a.fullName() => "Mike Sullivan"
// a.name() => "Mike"
```

This approach has as **advantage** the fact that the prototype of the Rectangle permits to instantiate different objects **without rewriting the same structure**, the **downside** is that in this way **properties cannot be added** to an object a created with the Person () constructor and the methods redefinitions can be confusing, eg:

```
b=new Person("Mike","Sullivan");
a.fullName= function (){return 100};
console.log(a.fullName()); // output= 100
console.log (b.fullName()); // output = "Mike Sullivan"
```

For **each** new Person() **object** a **memory space** is **allocated**. (So basically two fullName functions will be held in memory).

## Prototypes

Every object is **associated** to another object called **prototype**. The Prototype is a separate naming space, shared by all instances of the same constructor/class.

```
function Person(n,s){
  this.name=n;
  this.surname=s;
}

Person.prototype.fullName=function(){return this.name + " " + this.surname};
Person.prototype.nick="Nickname";

let p1= new Person("James", "Sullivan");
let p2= new Person("Mike", "Wazowski");

console.log(p1.nick); // =>Nickname
console.log(p2.nick); // =>Nickname

p2.nick="Hello";

console.log(p1.nick); // =>Nickname
console.log(p2.nick); // =>Hello

Person.prototype.nick="Sara";

console.log(p1.nick); // =>Sara
console.log(p2.nick); // =>Hello
//NB: The propeties that are not found are searched in the prototype but if is already
"directly" specified this value will be selected over the prototype one.
```

## How to inspect objects

```
x = new Person("Mark", "Evans");
Person.prototype.nick="Nickname";

for (let e in x){
    console.log(e+" "+x[e]);
}

Output:
name Mark
surname Evans
fullName function(){return this.name + " "+ this.surname}
nick Sara
```

Nb: "e" is a string, if I want to consider only the properties attached to the object itself and not the prototypes `getOwnPropertyNames()` or `hasOwnProperty()` methods can be used to check:

```
x = new Person("Mark", "Evans");
Person.prototype.nick="Nickname";
Object.defineProperty(x,"age",{value: 33, enumerable: false});
//adds a non-enumerable property
y= Object.getOwnPropertyNames(x);
console.log(y);

Output:
[ 'name', 'surname', 'age' ]
NB: nick property is not present because it was "inherited" from the
prototype of Person.
```

If I want to use the enumerable:

```
x = new Person("Mark", "Evans");
Person.prototype.nick="Nickname";
Object.defineProperty(x,"age",{value: 33, enumerable:
false});
//adds a non-enumerable property
y= Object.getOwnPropertyNames(x);
for(let e of y){
    if(x.propertyIsEnumerable(e)){
        console.log(e+ " "+ x[e]);
    }
}

Output:
name Mark
surname Evans
Nb: the result is similar to the previous one but "age" is
not present because not
```

Nb: in the first example the for was using "in" because x is an object, in the last one is using "of" because y is a sequence.

## Object inheritance

- 1) How to use an object as a “**model**” for another object:

```
function Person(){
  this.name;
  this.surname;
  this.isHuman=false;
  this.printHi=function(){
    console.log("My name is "+ this.name +", Human= "+ this.isHuman);
  }
}

let pippo = new Person();
pippo.printHi(); // My name is undefined, Human= false;

let Mat= Object.create(pippo); //Pippo is used as model for another
object
Mat.printHi(); // My name is undefined, Human= false;

Mat.name="Mat";
Mat.surname="Dan";
Mat.isHuman=true;
Mat.printHi(); // My name is Mat, Human= true
```

- 2) **Prototype based inheritance:**

```
function Person(n,s){
  this.firstname=n;
  this.lastname=s;
}
Person.prototype.nickname="The Best";
Person.prototype.fullname= function(){
  return this.firstname + " " + this.lastname;
};
function Athlete(n,s, sport){
  Person.call(this, n, s);
  this.sport=sport;
}
Athlete.prototype = Object.create(Person.prototype); // use the Person
prototype
Object.defineProperty(Athlete.prototype, 'constructor', {
  value: Athlete, enumerable: false, writable:true}
}); //all the Athlete objects share the same constructor

let person1= new Athlete("Michael", "Jordan","PingPong");
let person2= new Athlete("Yohan", "Blake","Swimming");
console.log(person1.fullname()+" "+ person1.sport+ " " + person1.nickname);
console.log(person2.fullname()+" "+ person2.sport+ " " + person2.nickname);
```

### Output:

Michael Jordan PingPong The Best

Yohan Blake Swimming The Best



Nb: the highlighted code has the same structure as:

```
Person.prototype = Object.create(Object.prototype);
Object.defineProperty(Person.prototype, 'constructor',
{
  value: Person, enumerable: false, writable:true}
);
```

but this is done implicitly.

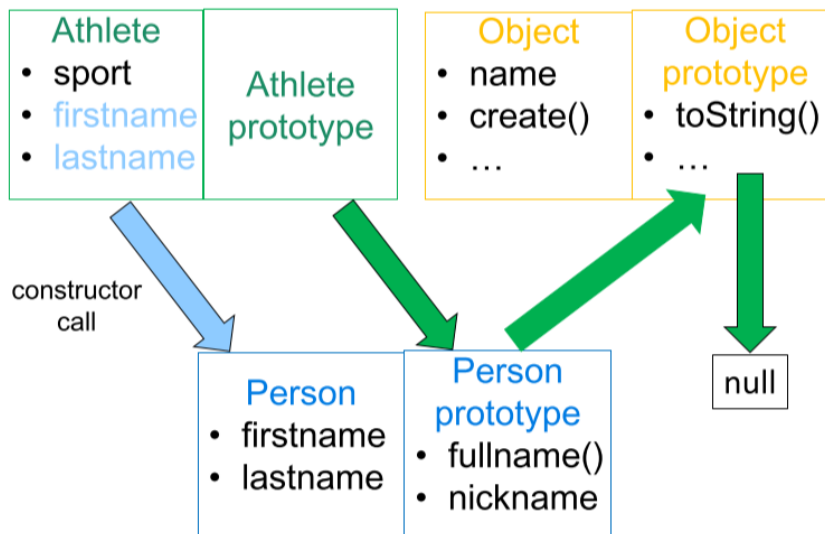
### The “Big Picture”:

In the constructor we define the instance variables of our instances.

In the prototype we define the methods, and the variables shared by all our instances.

By invoking the "superclass" constructor, we inherit its instance variables.

By associating or prototype to the "superclass", we inherit its prototype.



(Inheritance exercise: see repo)

### Predefined objects

see: [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects?](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects?)

native prototypes should never be extended

## Classes

JS classes are **syntactic sugar** of the **prototype based inheritance**. Class declarations are **not hoisted**.

Eg:

```
class Person{
  constructor(f,l) {
    this.firstName = f;
    this.lastName = l;
  }
  fullname(){
    return this.firstName+" "+this.lastName;
  }
  maxAge=300;
}
class Athlete extends Person{
  constructor(f,l,s){
    super(f,l);
    this.sport=s;
  }
}

let person1= new Athlete("Michael", "Jordan","PingPong");
let person2= new Athlete("Yohan", "Blake","Swimming");
console.log(person1.fullname()+" "+ person1.sport);
console.log(person2.fullname()+" "+ person2.sport);
```

**Output:**

Michael Jordan PingPong  
Yohan Blake Swimming

The **instance variable** are **inside** the **constructor** (not inside the class like in Java, at least for the current version of JS).

```
Person.minAge=0;
Person.prototype.maxHeight=250;
let person0=new Person("Bob", "Billie");
console.log(person0.maxAge); // 300
console.log(person0.minAge); //undefined
console.log(person0.maxHeight); // 250
console.log(Person.maxAge); //undefined
console.log(Person.minAge); // 0
console.log(Person.maxHeight); // undefined
```

If we define a **static methods** they **cannot** be **called** on the **instances**.

```
class Persons{
  constructor(f,l) {
    this.firstName = f;
    this.lastName = l;
  }
  fullname(){
    return this.firstName+" "+this.lastName;
  }
  static names(p){
    return p.firstName;
  }
}

let pers= new Persons("Fernando", "Alonso");
console.log(Persons.names(pers)); // Fernando
```

## Arrays

Can be **declared** in two ways:

- `a=[];`
- `a= new Array();`

Can be **accessed** in three ways:

- `a[number]`
- `a[string]`
- `a.value`

Arrays are **sparse**, **inhomogeneous** and **associative**.

Functions:

- add/remove an element at the end
- add/remove an element at the front
- add/remove an element by index
- remove a number of elements starting from an index
- find the index of an element
- make a copy of an array

Nb: **length has a strange behavior**: if I declare an array and then sets something like `a[0]=2`, `a[3]=9`, `a[12]=4` then `a.length()`=13 → **It adds one to the last used index**.

See [https://www.w3schools.com/js/js\\_arrays.asp](https://www.w3schools.com/js/js_arrays.asp) .