Image maps

It is possible to specify that clicking on different parts of an image has different effects. Such images are called *Image maps*. HTML 2.0 had only so-called *Server-side image maps*, but in HTML 3.2 also *Client-side image maps* were added.

The difference between server- and client-side image maps is where the decision is made what to do.

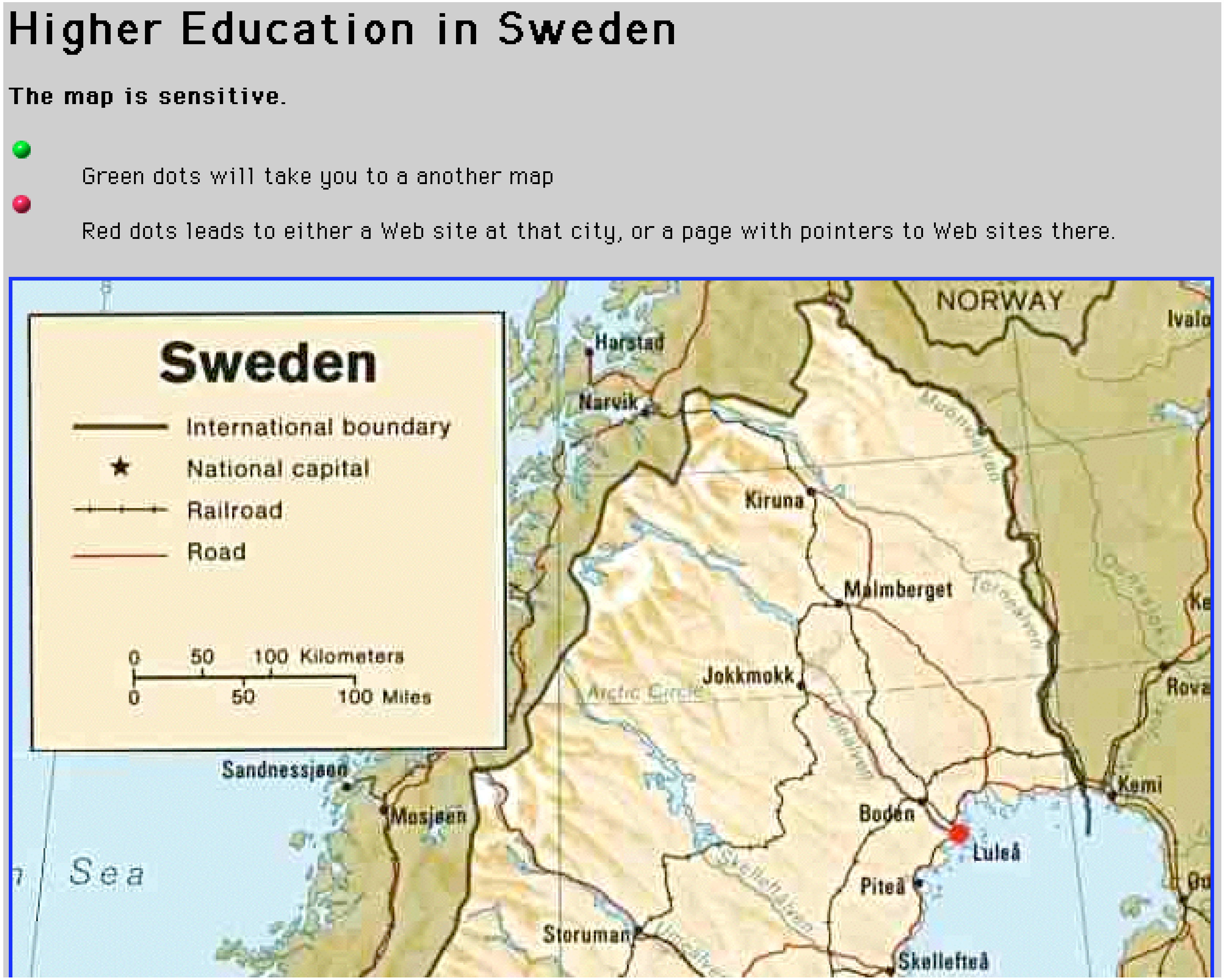
With server-side image maps, the x- and y-coordinates of the place in the image on which the user clicked are transferred to the server, and the software in the server then decides, based on these coordinates, what to return.

With client-side image maps, the HTML markup contains specifications of which areas of an image will represent different URIs, so that the user's web browser can deduce which URI to use depending on where in the map the user clicked.

This means that with client-side image maps, no special functionality is needed in the server to handle such maps. The individual page designer can thus use image maps without help from the service provider.

More information about image maps see URL
`http://www.berkana.com/class2/maps.html`

Example of a client-side image map.



Higher Education in Sweden

The map is sensitive.

●

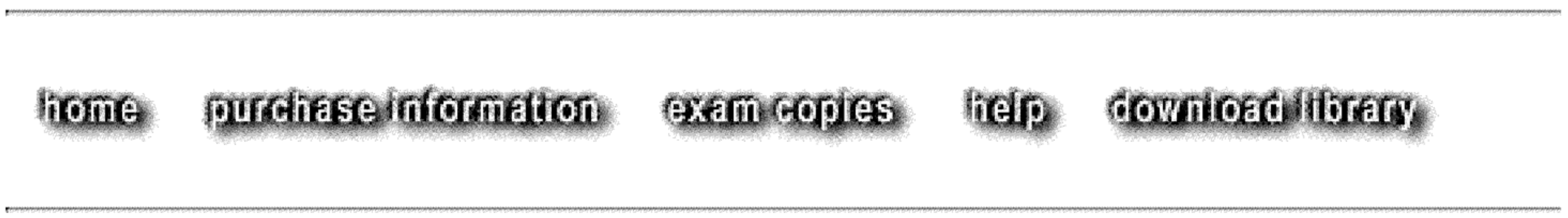
Green dots will take you to a another map

●

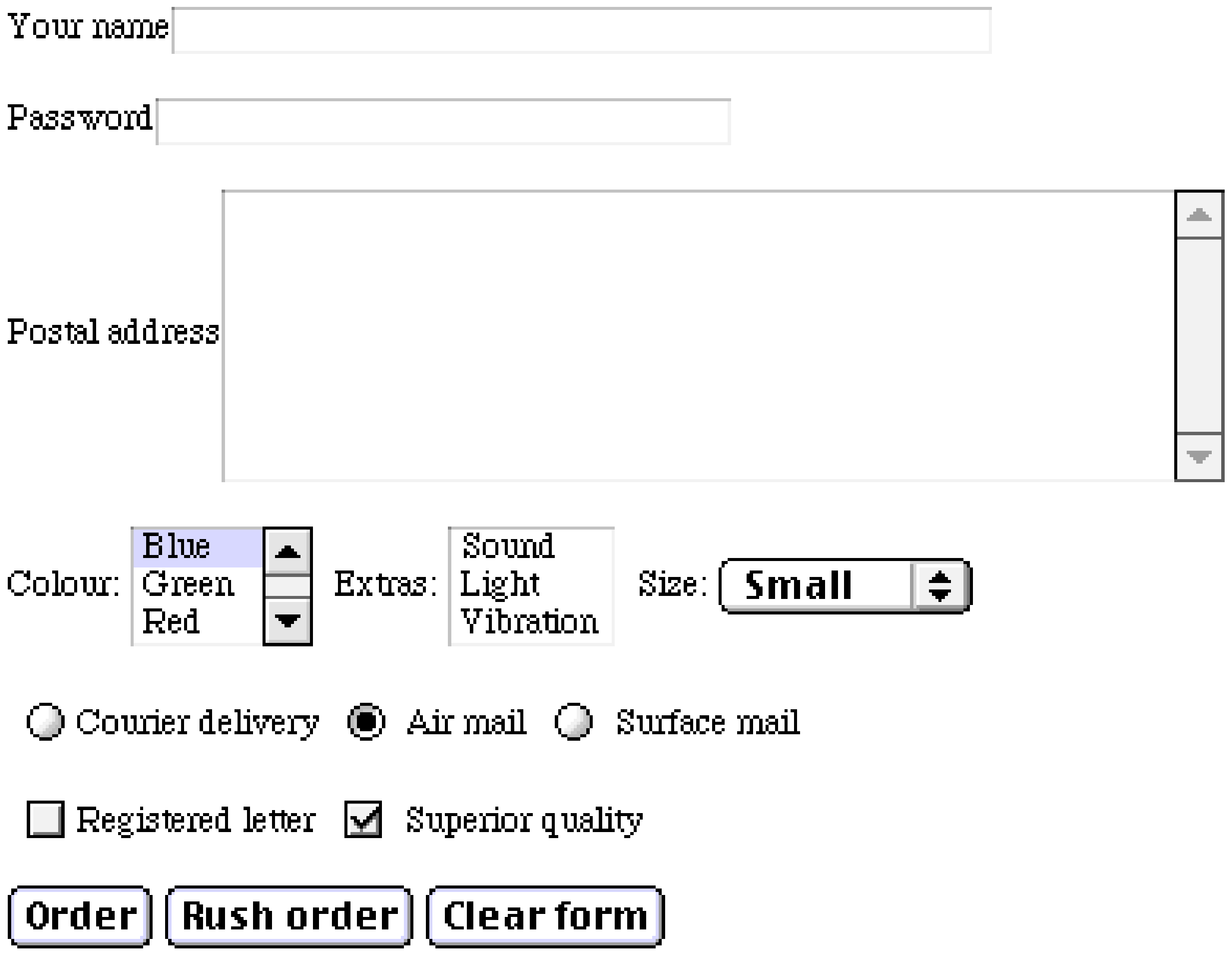
Red dots leads to either a Web site at that city, or a page with pointers to Web sites there.

Example of a server-side image map

The image map below could also have been realized by several separate graphics, one for each command, and with its own URL, or, of course, with plain textual URL-s. What are the pros and cons of these three methods of rendering the same information?



HTML forms makes HTML into a general user interface generation language



Your name

Password

Postal address

Colour: [✓ Blue] Green Red   Extras: Sound Light Vibration   Size: Small

○ Courier delivery [✓ Air mail] ○ Surface mail

☐ Registered letter ☑ Superior quality

HTML forms example; markup:

At URL: `HTTP://www.dsv.su.se/~jpalme/test/form-example-1.html`

```
<FORM ACTION=ordering-script.cgi
ENCTYPE= "application/x-www-form-urlencoded" METHOD=POST>

Your name<INPUT NAME="name" TYPE=TEXT SIZE="43"
MAXLENGTH="60"><P>

Password<INPUT NAME="PW" TYPE=PASSWORD SIZE="30"><P>

Postal address<TEXTAREA NAME="Address" ROWS="7"
COLS="50"></TEXTAREA><P>

Colour: <SELECT SIZE="3" NAME="Colour">
<OPTION SELECTED>Blue    <OPTION>Green
<OPTION>Red              <OPTION>Yellow
<OPTION>Brown            </SELECT>

Extras: <SELECT NAME="Extras" SIZE="3" MULTIPLE>
<OPTION>Sound <OPTION>Light <OPTION>Vibration </SELECT>
```

```
Size: <SELECT NAME="Size">
<OPTION SELECTED>Small <OPTION>Medium <OPTION>Large </SELECT><P>

<INPUT TYPE=RADIO VALUE="courier" NAME="delivery">
Courier delivery
<INPUT TYPE= RADIO VALUE="air-mail" NAME="delivery" CHECKED>
 Air mail
<INPUT TYPE= RADIO VALUE="surface" NAME="delivery">
Surface mail<P>

<INPUT TYPE=CHECKBOX NAME="Registered" VALUE="Yes"> Registered
letter

<INPUT TYPE=CHECKBOX NAME="Quality" VALUE="Superior" CHECKED>
Superior quality<P>

<INPUT NAME="submit" TYPE=SUBMIT VALUE="Order">
<INPUT NAME="submit" TYPE=SUBMIT VALUE="Rush order">
<INPUT NAME="reset" TYPE=RESET VALUE="Clear form"><P>

</FORM></BODY></HTML>
```

HTML form attributes

Start of HTML form

Element	Attribute	Description
FORM		Start of a FORM..
	ACTION	The action URI for the form. Default: Base URI of the document>.
	METHOD	GET with no side-effects.
		POST has side-effects.
	ENCTYPE	Media type for encoding sent data.

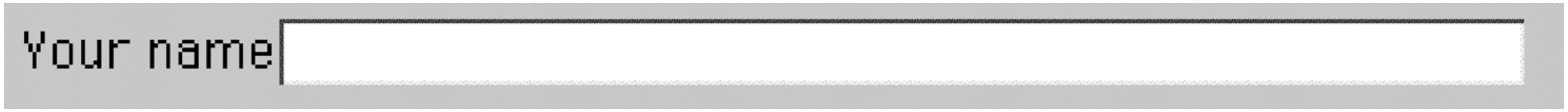
One-line text input

`INPUT`		A field for user input.
	`TYPE=TEXT`	A single line text-entry field.
	`MAXLENGTH`	Maximum number of characters.
	`SIZE`	Display space.
	`VALUE`	Initial value.
	`TYPE= PASSWORD`	Same as TEXT, but not shown on the screen.

Example:

```
Your name<INPUT NAME="name" TYPE="text" SIZE="43"
MAXLENGTH="60">
```

Rendering:



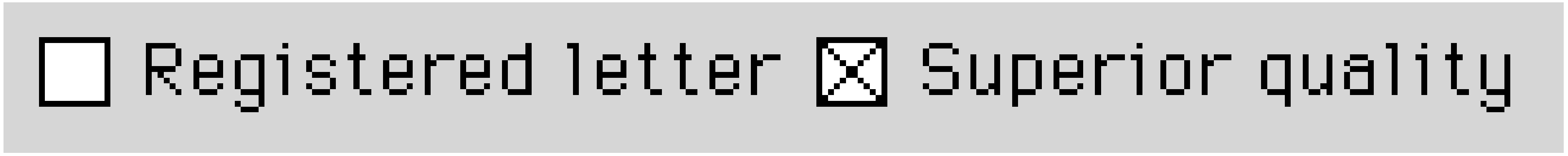
Checkbox

INPUT		A field for user input.
	TYPE= CHECKBOX	A checkbox.
	NAME	Symbolic name for group of fields
	VALUE	Portion of the value contributed by this element
	CHECKED	Initial state

Example:

```
<INPUT TYPE="checkbox" NAME="Registered" VALUE="Yes">
Registered letter

<INPUT TYPE="checkbox" NAME="Quality" VALUE="Superior"
CHECKED> Superior quality<P>
```

Rendering:


☐ Registered letter ☒ Superior quality

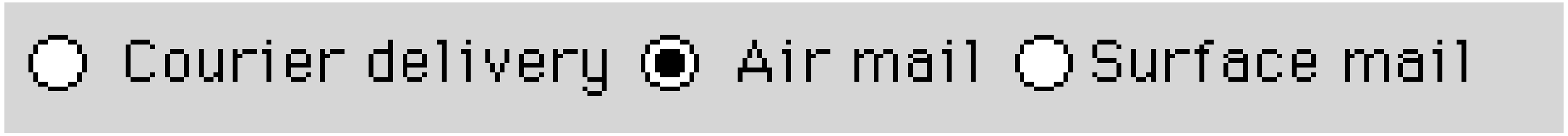
Radio button

INPUT		A field for user input.
	TYPE= RADIO	A checkbox.
	NAME	Symbolic name for group of fields.
	VALUE	Portion of the value contributed by this element.
	CHECKED	Initial state (can only be set for one in a group).

Example:

```
<INPUT TYPE=RADIO VALUE="courier" NAME="delivery">
Courier delivery
<INPUT TYPE=RADIO VALUE="air-mail" NAME="delivery" CHECKED>
 Air mail
<INPUT TYPE=RADIO VALUE="surface" NAME="delivery">
Surface mail
```

Rendering:



HIDDEN field

`INPUT`		A field for user input.
	`TYPE=HIDDEN`	Field which is not displayed to the user, but which returns value when the form is submitted.
	`NAME`	Symbolic name for group of fields.
	`VALUE`	Value submitted.

Example:

```
<INPUT TYPE=HIDDEN VALUE="xy654zd" NAME="password">
```

Rendering:

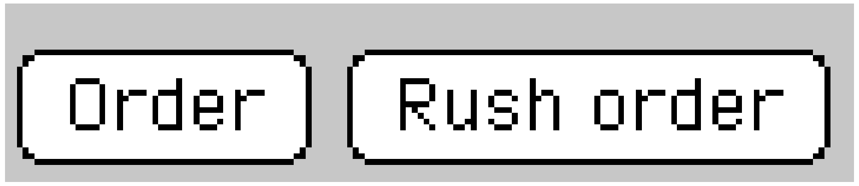
Submit

`INPUT`		A field for user input.
	`TYPE= SUBMIT`	A submit button.
	`NAME`	Symbolic name.
	`VALUE`	Value submitted (can be different for different submit buttons).

Example:

```
<INPUT NAME="submit" TYPE=SUBMIT VALUE="Order">
<INPUT NAME="submit" TYPE=SUBMIT VALUE="Rush order">
```

Rendering:



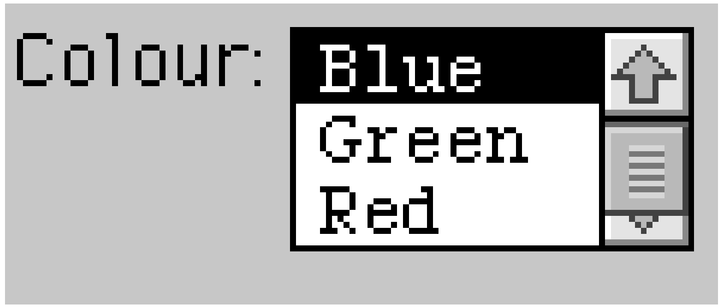
Select

SELECT		Start of a selection field.
	MULTIPLE	Allow more than one option to be chosen.
	NAME	Symbolic name.
	SIZE	Number of visible elements. SIZE=1 gives pop-down menu, No. of elements > SIZE >1 gives scrolling list, SIZE=No.of elements gives list.
OPTION		Alternative in a selection field.
	SELECTED	This option is initially selected, defaults to first.
	VALUE	Value returned, defaults to content of element.

Example:

```
Colour:
<SELECT SIZE="3" NAME="Colour">
<OPTION SELECTED>Blue <OPTION>Green
<OPTION>Red           <OPTION>Yellow
<OPTION>Brown          </SELECT>
```

Rendering:



SELECTED above was not necessary, since by default the first option is selected.

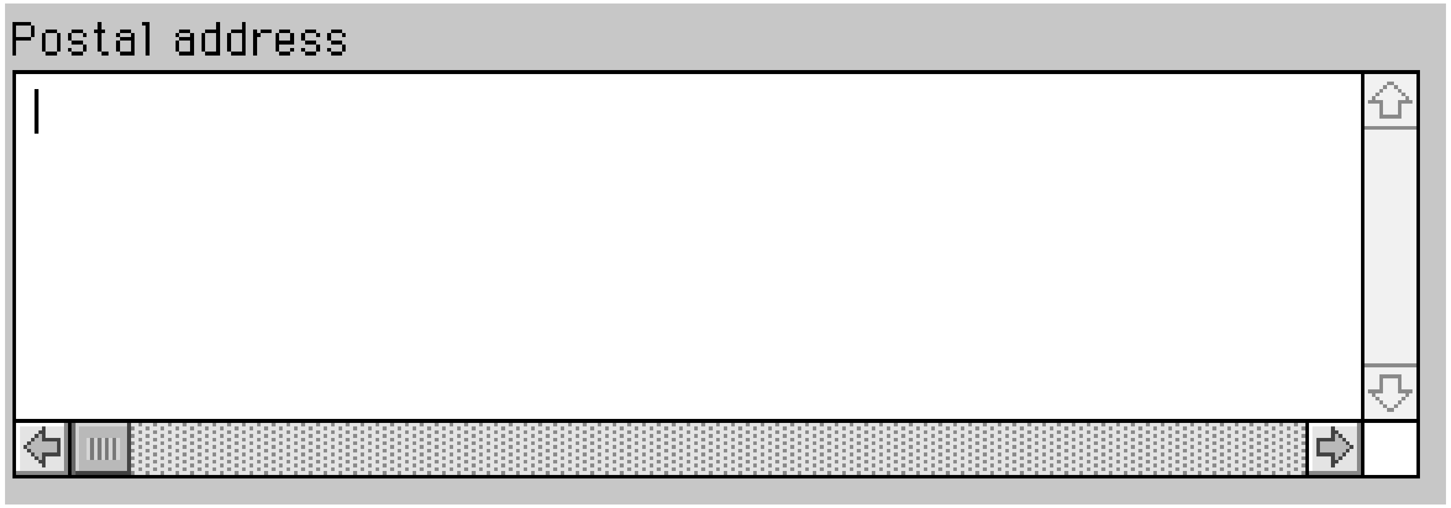
Textarea

TEXTAREA		Multi-line text field.
	COLS	Width in characters of visible field.
	NAME	Symbolic name.
	ROWS	Number of visible rows.

Example:

```
Postal address<TEXTAREA NAME="Address" ROWS="7"
COLS="50"></TEXTAREA><P>
```

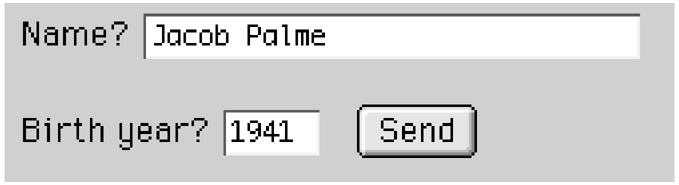
Rendering:



Submission of filled-in forms

HTTP method	MIME Content-Type	Description
get	application/x-www-form-urlencoded	Very compact single string, appended to the URL
post	multipart/form-data	Volumnious format, every field value becomes its own MIME body part

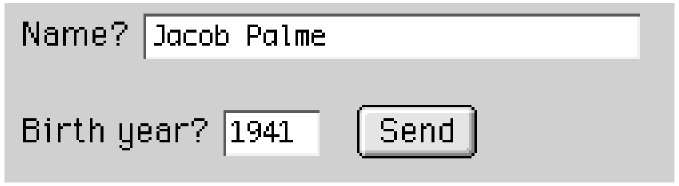
Example:



Name?	Jacob Palme
Birth year?	1941

```
<FORM action="mailto:foo@bar"
method="POST">
<P>Name? <INPUT TYPE="text" NAME="name"
VALUE="" SIZE=30 MAXLENGTH=30>
<P>Birth year? <INPUT TYPE="text"
NAME="born" VALUE="" SIZE=5
MAXLENGTH=4>&nbsp;&nbsp;<INPUT
TYPE="submit" NAME="Send" VALUE="Send">
</FORM>
```

Example:



Name? Jacob Palme

Birth year? 1941 Send

```
<FORM action="mailto:foo@bar" method="POST">
<P>Name? <INPUT TYPE="text" NAME="name" VALUE="
SIZE=30 MAXLENGTH=30>
<P>Birth year? <INPUT TYPE="text"
NAME="born" VALUE="" SIZE=5
MAXLENGTH=4>&nbsp;&nbsp;<INPUT TYPE="submit"
NAME="Send" VALUE="Send"> </FORM>
```

This example will be sent in the following format:

```
From: Jacob Palme <jpalme@dsv.su.se>
MIME-Version: 1.0
To: foo@bar
Subject: ...
Content-type: application/x-www-form-urlencoded

name=Jacob+Palme&born=1941&Send=Send
```

If the first line had been

```
<FORM action="http://www.dsv.su.se/cgi-bin/foo" method="GET">
```

it would have been sent as

```
GET /cgi-bin/foo?name=Jacob+Palme&born=1941&Send=Send HTTP/1.1
```

If the first line had been

```
<FORM action="mailto:foo@bar" method="POST" enctype="multipart/form-
data">
```

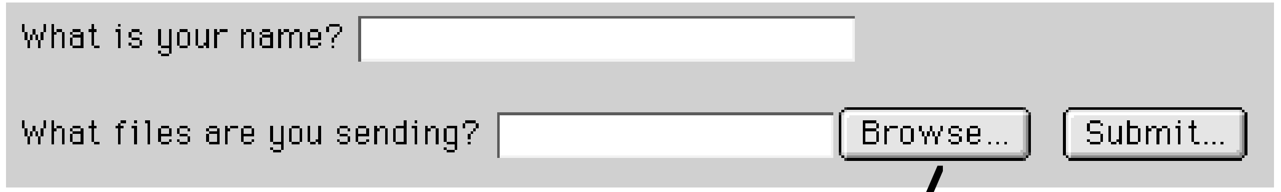
Then it would have been sent as

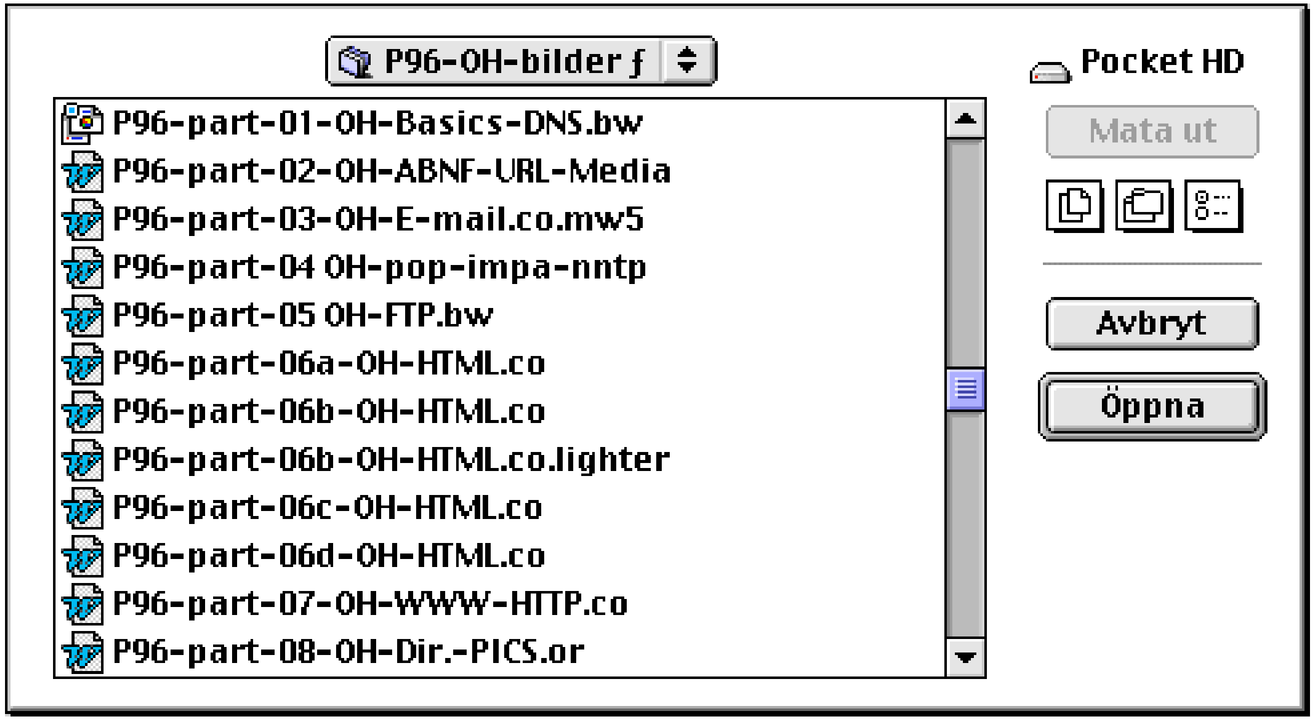
```
 From: Jacob Palme <jpalme@dsv.su.se>
 MIME-Version: 1.0
 To: foo@bar
 Subject: ...
 Content-type: multipart/form-data; boundary=++218421377

--++218421377911030
 Content-Disposition: form-data; name="name"

Jacob Palme
 --++218421377911030
 Content-Disposition: form-data; name="born"

 1941
 --++218421377911030
 Content-Disposition: form-data; name="Send"

 Send
 --++218421377911030--
```

A separate MIME body part for each field value

Form-based File Upload in HTML

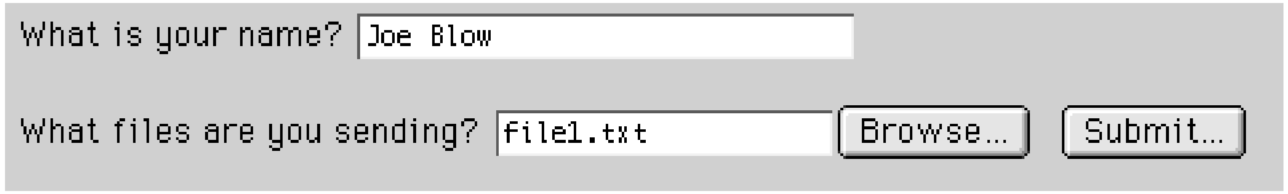
(RFC 1867 and HTML 4 chapter 17.13.4 Form Content types)





P96-part-01-OH-Basics-DNS.bw
P96-part-02-OH-ABNF-URL-Media
P96-part-03-OH-E-mail.co.mw5
P96-part-04 OH-pop-impa-nntp
P96-part-05 OH-FTP.bw
P96-part-06a-OH-HTML.co
P96-part-06b-OH-HTML.co
P96-part-06b-OH-HTML.co.lighter
P96-part-06c-OH-HTML.co
P96-part-06d-OH-HTML.co
P96-part-07-OH-WWW-HTTP.co
P96-part-08-OH-Dir.-PICS.or

Form-based File Upload in HTML



What is your name? Joe Blow

What files are you sending? file1.txt

```
<FORM ACTION="http://server.dom/cgi/handle"
      ENCTYPE="multipart/form-data"
      METHOD=POST>
What is your name? <INPUT TYPE=TEXT NAME=submitter>
What files are you sending? <INPUT TYPE=FILE NAME=pics>
</FORM>
```

`TYPE=FILE` indicates that a file is requested. `NAME` is to be used when sending the file to indicate which field in the form it applies to. The `ACCEPT` attribute can constrain which file patterns are allowed. The `SIZE` attribute can indicate a size of a field where the files the user has selected are listed. The `VALUE` attribute can be used to give a default file name.

The client might send back the following data:

```
Content-type: multipart/form-data, boundary=AaB03x

--AaB03x
content-disposition: form-data; name="field1"

Joe Blow
--AaB03x
content-disposition: form-data; name="pics"
Content-type: multipart/mixed, boundary=BbC04y

--BbC04y
Content-disposition: attachment; filename="file1.txt"
Content-Type: text/plain

... contents of file1.txt ...
--BbC04y

Content-disposition: attachment; filename="file2.gif"
Content-type: image/gif
Content-Transfer-Encoding: binary

  ...contents of file2.gif...
--BbC04y--
--AaB03x--
```

Note that each field value is sent as a separate MIME body part

Note: Binary data must be encoded

Building applications based on HTTP

Using the form facility of HTML, it is possible to build application programs based on HTML and HTTP. Such an application program uses a web browser as client, and a specially configured HTTP server as server. Almost any computer application which does not require very fast interaction with the user can be built in this way.

+ User does not have to install special client software in his personal computer/workstation.
+ HTML makes it easy to design the user interface.
+ Users may find the web browser interface easy to use because they are accustomed to it.
+ The web browser provides additional facilities automatically, in particular that any page can be printed or saved on a file, and the *Back* and *Forward* buttons in the web browser.

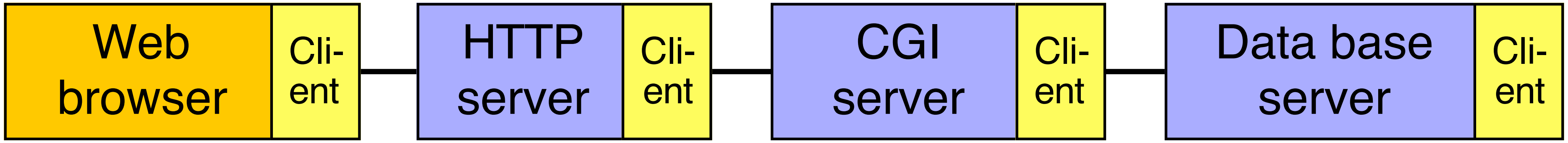
– Sometimes less neat user interface than with a custom-built client.
– Response times sometimes less good than with a custom-built client.
– Applications which require a data base in the client computer cannot be built in this way.

CGI = Common Gateway Interface

CGI is a standard for the interaction between an HTTP server and a special program. CGI allows the HTTP server to recognize special input from the user, for example filled-in-forms, and giving them to an application program. This program can then return a custom-built HTML page to be sent back to the user.

CGI is not the only possible way of doing this. HTTP servers and application can communicate using other methods also.

More about CGI will be said in a special lecture in this course, given by Fredrik Kilander.



- Long response times

HTML/HTTP applications which require server knowledge of previous interactions

Many application program requires that information is kept between interactions between a user and the server.

Examples:

- A user logs in, gives his name and password, and can then perform multiple interactions with the server without having to give his name and password again.
- A user retrieves data, and then in a later interaction wants to perform some action based on the data retrieved in the previous interaction.
- A user inputs data, and then in a later interaction wants to perform some action on the data already input (for example change words in a text, add recipients to a message, etc.)
- Suppose for example that we designed software for a user to communicate with his bank. The user might first move some money from one account to another, and then use the moved money to pay a bill.

HTTP (version 1.0) is a stateless protocol

HTTP 1.0 is a stateless protocol. There is no knowledge of previous interactions in the protocol. Every request creates a new interaction, which opens a connection, performs the interaction (for example retrieving data, och sending in a form which the user has filled in). After data has been transmitted, the HTTP connection between the client and the server is broken. Thus, HTTP 1.0 as such is not suitable for sessions of multiple interactions between user and server, unless some special trick is used. Also HTTP 1.1 is inpractice mostly used as a stateless protocol.

Here are some such special tricks:

(1) Store session information in custom-built URLs.

(2) Store session information in hidden fields in a form.

(3) Use cookies.

(1) Store session information in custom-built URLs.

When the server creates the custom-built web page to be sent to the user, the server can store session information in specially built URLs. When the user clicks on these URLs, this in formation is sent back to the server.

Example: The server creates a URL like this:

```
 HTTP:/www.dsv.su.se/exam-results?per-nils+sf14ty
```

where `per-nils` is the user account and `sf14ty` is the user password.

It is somewhat dangerous to store passwords in URL-s which other people might see and use. To reduce this risk, often a special session password is used, with more limited applicability. For example, the session password will become invalid if there is no interaction in 10 minutes. Such session passwords are often named *Magic cookie*. A *Magic cookie* is a special password which gives the user some special rights, often only at a certain time. A *Magic cookie* often also gives the server information to identify the user, so that it can replace both the user name and the user password.

(2) Store session information in hidden fields in a form.

If the interaction is made by the server sending forms to the user, and the user returning the filled in forms, then the server can store session information in the forms sent to the user. This can be done in open fields, if the user needs to see the information sent, or in hidden form fields, if the user would not want to see it. The contents of the hidden fields are sent back with the filled-in form to the server. Two common usage of such hidden fields are:

(a) To store user account names, passwords or magic cookies. This information will help the server to look up the user information.

Disadvantage: Server has to store information about each concurrent user.

(b) To store full information of what the user has achieved. For example, an application where the user interactively creates a budget, the whole budget could be sent back in each interaction. Thus, the server need not remember anything of previous interactions, all of it is provided in data sent to the server with every interaction.

Disadvantage: The amount of information sent back and forward between user and server must not get too large, or the response times will be less good. If, for example, a 28800 bps connection is used and a maximum delay of 5 seconds is acceptable, then a maximum of 28800*5/2*10 = 72 Kbytes is acceptable.

(3) Use cookies

Newer browsers have a *cookie* facility, with which a server can store a “cookie”, i.e. some kind of session-ID, in the web browser, which the server at a later time can query the value of.

(More about cookies in the HTTP lecture.)

Version 1.1 of HTTP has facilities to support *persistent connections*.

The disadvantage with these methods, is that they are not supported by all web browsers, and that some users set their browsers to not accept cookies, because they believe cookies to be an infringement of their privacy.

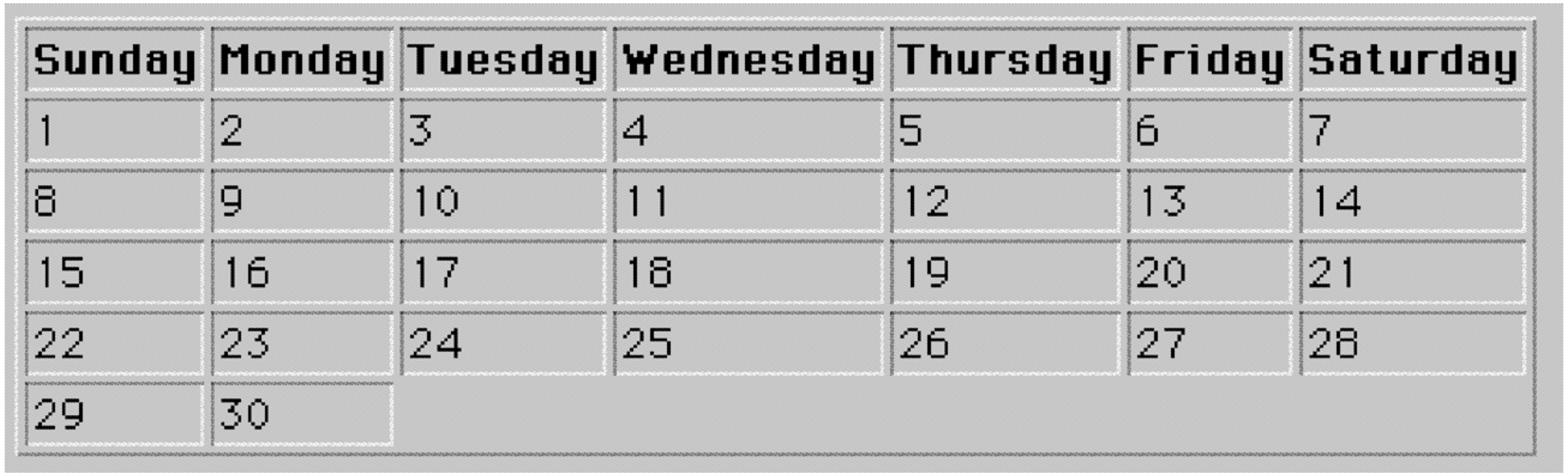
HTML tables

A feature in HTML 3.2, which is much used and supported by many browsers, is HTML tables.

Table example 1:

This example can be found at URL:

`HTTP://www.dsv.su.se/~jpalme/test/table-example-1.html`



Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30					

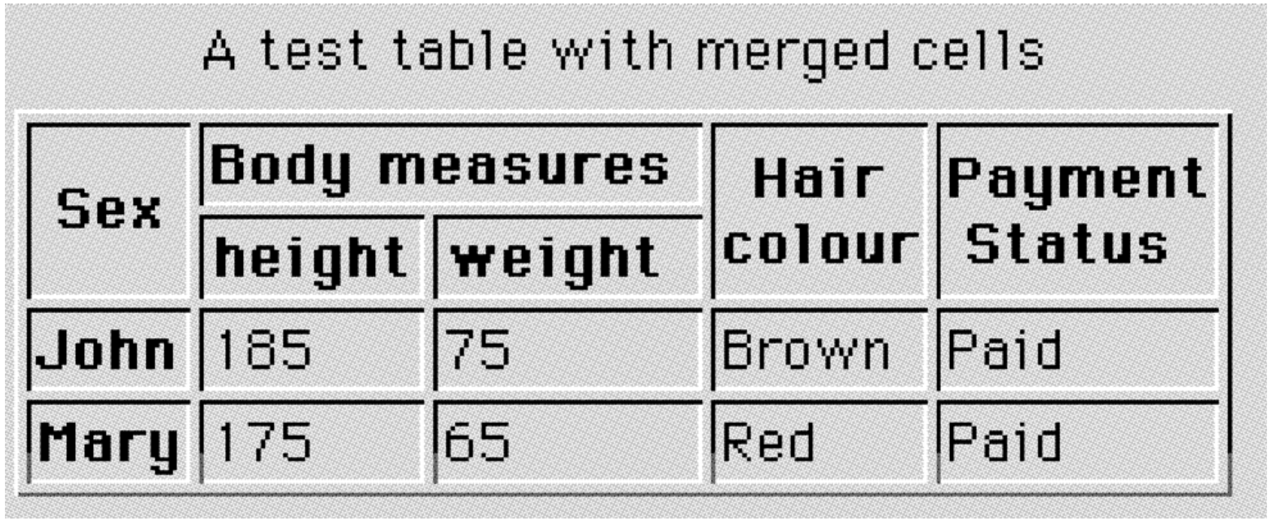
HTML code behind table example 1:

```
<TABLE>
<CAPTION>Calendar for September 1996</CAPTION>
<TR><TH>Sunday</TH><TH>Monday</TH><TH>Tuesday</TH>
<TH>Wednesday</TH><TH>Thursday</TH><TH>Friday</TH>
<TH>Saturday<BR></TH></TR>
<TR><TD>1</TD><TD>2</TD><TD>3</TD><TD>4</TD><TD>5</TD>
<TD>6</TD><TD>7<BR></TR>
<TR><TD>8</TD><TD>9</TD><TD>10</TD><TD>11</TD><TD>12</TD>
<TD>13</TD><TD>14<BR></TR>
<TR><TD>15</TD><TD>16</TD><TD>17</TD><TD>18</TD><TD>19</TD>
<TD>20</TD><TD>21<BR></TR>
<TR><TD>22</TD><TD>23</TD><TD>24</TD><TD>25</TD><TD>26</TD>
<TD>27</TD><TD>28<BR></TR>
<TR><TD>29</TD><TD>30</TD></TR>
</TABLE>
<BR CLEAR=LEFT>
```

Column width autofit is very useful for tables with lots of

Merging cells

It is possible to merge adjacent cells both horizontally and vertically into arbitrary rectangular shapes.

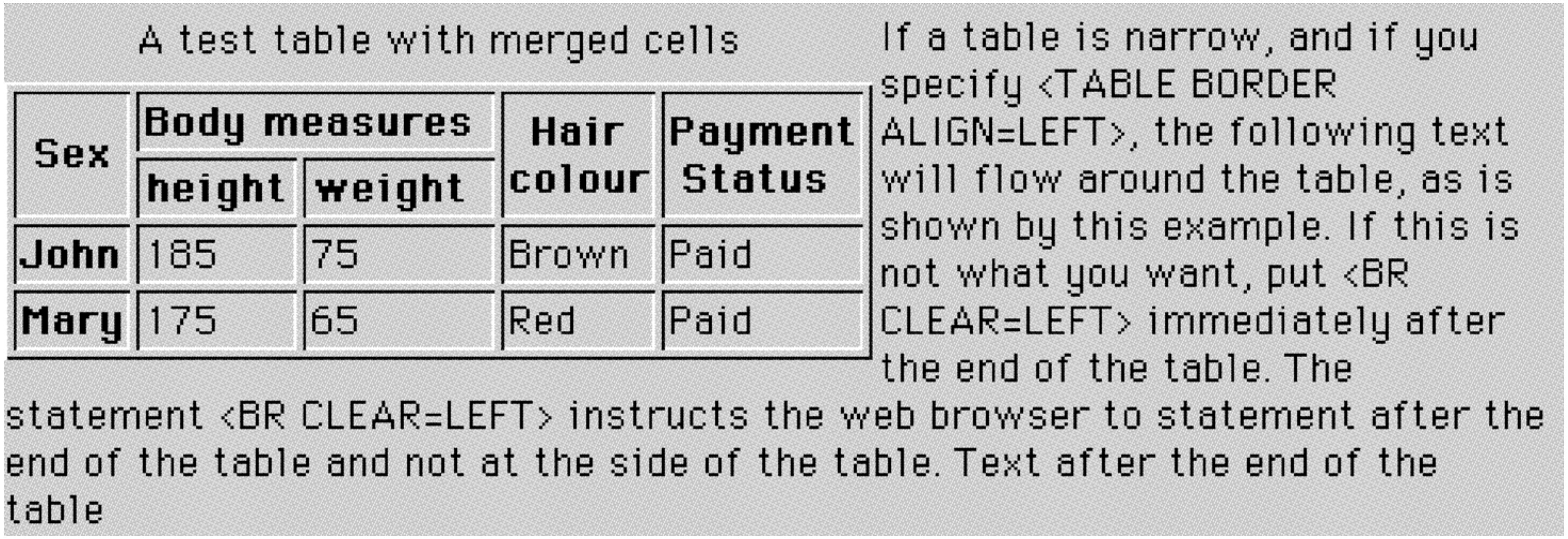


A test table with merged cells

Sex	Body measures		Hair colour	Payment Status
	height	weight		
John	185	75	Brown	Paid
Mary	175	65	Red	Paid

```
<TABLE BORDER=1>
  <CAPTION>A test table with merged cells</CAPTION>
  <TR><TH ROWSPAN=2>Sex<TH COLSPAN=2>Body measures
      <TH ROWSPAN=2>Hair<BR>colour<TH
ROWSPAN=2>Payment<BR>Status
  <TR><TH>height<TH>weight
  <TR><TH>John<TD>185<TD>75<TD>Brown<TD>Paid
  <TR><TH>Mary<TD>175<TD>65<TD>Red<TD>Paid
</TABLE>
```

Text flowing around a table



A test table with merged cells

Sex	Body measures		Hair colour	Payment Status
	height	weight		
John	185	75	Brown	Paid
Mary	175	65	Red	Paid

If a table is narrow, and if you specify <TABLE BORDER ALIGN=LEFT>, the following text will flow around the table, as is shown by this example. If this is not what you want, put <BR CLEAR=LEFT> immediately after the end of the table. The statement <BR CLEAR=LEFT> instructs the web browser to statement after the end of the table and not at the side of the table. Text after the end of the table

HTML markup for text flowing around a table

```
<TABLE BORDER=1 ALIGN=LEFT>
  <CAPTION>A test table with merged cells</CAPTION>
  <TR><TH ROWSPAN=2>Sex<TH COLSPAN=2>Body measures
      <TH ROWSPAN=2>Hair<BR>colour<TH
ROWSPAN=2>Payment<BR>Status
  <TR><TH>height<TH>weight
  <TR><TH>John<TD>185<TD>75<TD>Brown<TD>Paid
  <TR><TH>Mary<TD>175<TD>65<TD>Red<TD>Paid
</TABLE>
```

```
If a table is narrow, and if you specify &lt;TABLE BORDER
ALIGN=LEFT&gt;, the following text will flow around the table,
as is shown by this example. If this is not what you want, put
&lt;BR CLEAR=LEFT&gt; immediately after the end of the table.
The statement &lt;BR CLEAR=LEFT&gt; instructs the web browser
to statement after the end of the table and not at the side of
the table.
```

To cause text to start below the table and not flow around it, put the element **<BR CLEAR=LEFT>** before the text, or do not put **ALIGN=LEFT** in the **<TABLE>** element.

Java, Javascript (ECMAScript)

Program (s.k. applets) som laddas ner från nätet samtidigt med websidor. Kan köras så snart de laddats ner. Exempel på användningar:

- Minska svarstiden genom lokal interaktion istället för server-interaktion
- Rörliga bilder och andra saker som inte så enkelt kan göras med vanlig HTML
- Begränsade av säkerhetsproblem

Skillnaden mellan Java och Javascript

Egenskap	Java	Javascript
Lagring	I separata filer, till vilka det finns länkar i HTML-texten.	Som del av HTML-texten.
Kompilering	Kompileras till bytekod, som sedan interpreteras, eller till objetkod som exekveras.	Källkoden interpreteras direkt.
Funktioner	Generellt programmeringsspråk. observera dock säkerhetsbegränsningarna.	Mest kommandon för att styra webläsaren. Kan idag inte ändra i redan visad web-sida (däremot skriva nya sidor), detta kommer snart att ändras!