**Ministry of education and science of Ukraine**

**Ivano-Frankivsk national technical university of oil and gas**

**Computer Systems and networks department**

**M.O. Slabinoha**

**FRONTEND DEVELOPMENT AND WEB-DESIGN**

###### **LECTURE NOTES**

COMPILATION

PART 1

###### 

**Ivano-Frankivsk**

**2021**

UDC 004.514

С 47

| Reviewer: |  |
| --- | --- |
| A.R. Voronych | PhD, Docent, Docent of computer systems and networks department, Ivano-Frankivsk national technical university of oil and gas |

*Recommended by methodical university council*

*( protocol # of )*

M.O. Slabinoha

С 47. Frontend development and web-design: lecture notes (compilation, part 1)/ M.O. Slabinoha – Ivano-Frankivsk, IFNTUOG, 2021. – 106 p.

МВ 02070855- -2021

The lecture notes on the discipline "Frontend development and web design" are developed in accordance with the syllabus of the discipline and the working curriculum.

Designed to prepare bachelors in specialty 123 - "Computer Engineering". Lecture notes can be used by full-time and part-time students.

© M.O. Slabinoha

© IFNTUOG, 2021

UDC 004.514

С 47

| Reviewer: |  |
| --- | --- |
| A.R. Voronych | PhD, Docent, Docent of computer systems and networks department, Ivano-Frankivsk national technical university of oil and gas |

*Recommended by methodical university council*

*( protocol # of )*

M.O. Slabinoha

С 47. Frontend development and web-design: lecture notes (compilation, part 1)/ M.O. Slabinoha – Ivano-Frankivsk, IFNTUOG, 2021. – 106 p.

The lecture notes on the discipline "Frontend development and web design" are developed in accordance with the syllabus of the discipline and the working curriculum.

Designed to prepare bachelors in specialty 123 - "Computer Engineering". Lecture notes can be used by full-time and part-time students.

© M.O. Slabinoha

© IFNTUOG, 2021

Head of computer systems and networks department S.I. Melnychuk

Agreed:

Member of the expert review

university commission V.V. Bandura

Normocontroller H. Ya.Tomashivska

Leading librarian STL H. M. Matsyuk

CONTENTS

DISCLAIMER………………………………………………………………….5

TOPIC 1………………………………………………………………………...6

TOPIC 2………………………………………………………………………..19

TOPIC 3………………………………………………………………………..26

TOPIC 4………………………………………………………………………..33

TOPIC 5………………………………………………………………………..43

TOPIC 6………………………………………………………………………..50

TOPIC 7………………………………………………………………………..63

TOPIC 8………………………………………………………………………..74

TOPIC 9………………………………………………………………………..97

REFERENCES……………………………………………………………….106

## 

## DISCLAIMER

## These lecture notes are just compilation of different sources that explain the topic of subject in the best way the author can imagine. The author of this document is grateful for the tremendous work of creators of these materials and doesn’t mean to violate their exclusive copyright on their creation.

## This will be used with all the referencing responsibility for teaching the students and making IT education better. The author doesn’t support the plagiarism in any possible way. All the references are mentioned in the corresponding section

## TOPIC 1

HOW THE WEB WORKS

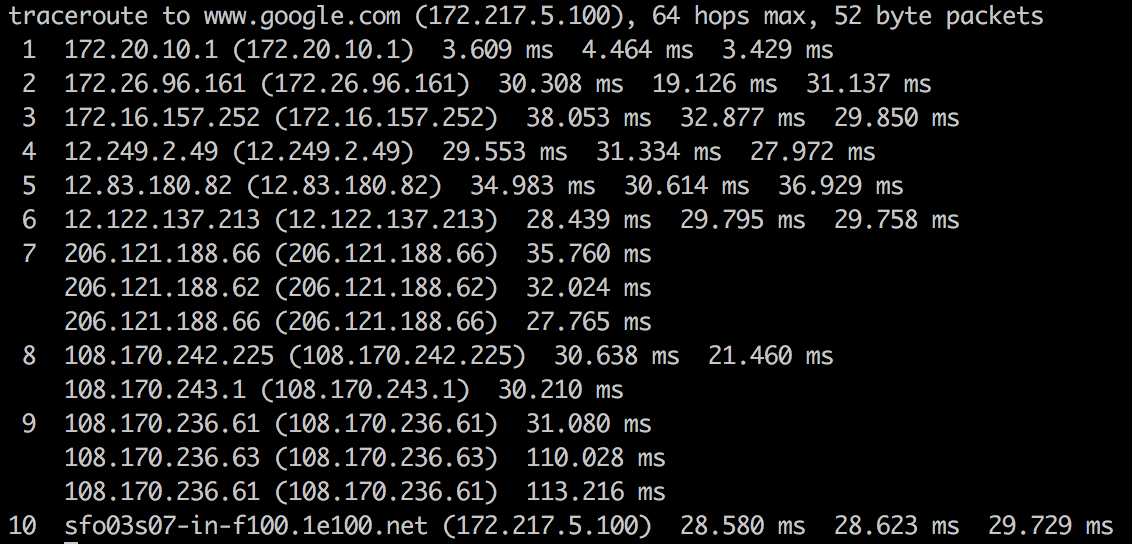
The Internet works through a packet routing network in accordance with the Internet Protocol (IP), the Transport Control Protocol (TCP) and other protocols.

A protocol is a set of rules specifying how computers should communicate with each other over a network. For example, the Transport Control Protocolhas a rule that if one computer sends data to another computer, the destination computer should let the source computer know if any data was missing so the source computer can re-send it. Or the Internet Protocolwhich specifies how computers should route information to other computers by attaching addresses onto the data it sends.

Data sent across the Internet is called a message. Before a message is sent, it is first split in many fragments called packets. These packets are sent independently of each other. The typical maximum packet size is between 1000 and 3000 characters. The Internet Protocol specifies how messages should be packetized.

Packet working network is a network that routes packets from a source computer to a destination computer. The Internet is made up of a massive network of specialized computers called routers. Each router’s job is to know how to move packetsalong from their source to their destination. A packet will have moved through multiple routers during its journey.

When a packet moves from one router to the next, it’s called a hop. You can use the command line-tool traceroute to see the list of hops packets take between you and a host.



**Fig. 1.1. Command-line utility traceroute showing all the hops between my computer and google’s servers**

The Internet Protocol specifies how network addresses should be attached to the packet’s headers, a designated space in the packet containing its meta-data. The Internet Protocol also specifies how the routers should forward the packets based on the address in the header.

These routers originated in the 1960s as ARPANET, a military project whose goal was a computer network that was decentralized so the government could access and distribute information in the case of a catastrophic event. Since then, a number of Internet Service Providers (ISP) corporations have added routers onto these ARPANET routers.

There is no single owner of these Internet routers, but rather multiple owners: The government agencies and universities associated with ARPANET in the early days and ISP corporations like AT&T and Verizon later on.

Asking who owns the Internet is like asking who owns all the telephone lines. No one entity owns them all; many different entities own parts of them.

The packets may arrive at their destination out of order. This happens when a later packet finds a quicker path to the destination than an earlier one. But packet’s header contains information about the packet’s order relative to the entire message. The Transport Control Protocol uses this info for reconstructing the message at the destination.

The Internet Protocol makes no guarantee that packets will always arrive at their destinations. When that happens, it's called a packet loss. This typically happens when a router receives more packets it can process. It has no option other than to drop some packets.

However, the Transport Control Protocol handles packet loss by performing re-transmissions. It does this by having the destination computer periodically send acknowledgement packets back to the source computer indicating how much of the message it has received and reconstructed. If the destination computer finds there are missing packets, it sends a request to the source computer asking it to resend the missing packets.

When two computers are communicating through the Transport Control Protocol, we say there is a TCP connection between them.

These addresses are called IP addresses and there are two standards.

The first address standard is called IPv4 and it looks like 212.78.1.25 . But because IPv4 supports only 2³² (about 4 billion) possible addresses, the Internet Task Force proposed a new address standard called IPv6, which look like 3ffe:1893:3452:4:345:f345:f345:42fc . IPv6 supports 2¹²⁸ possible addresses, allowing for much more networked devices, which will be plenty more than the as of 2017 current 8+ billion networked devices on the Internet.

As such, there is a one-to-one mapping between IPv4 and IPv6 addresses. Note the switch from IPv4 to IPv6 is still in progress and will take a long time. As of 2014, Google revealed their IPv6 traffic was only at 3%.

It’s because there are public and private IP addresses. Multiple devices on a local network connected to the Internet will share the same public IP address. Within the local network, these devices are differentiated from each other by private IP addresses, typically of the form 192.168.xx or 172.16.x.x or 10.x.x.x where x is a number between 1 and 255. These private IP addresses are assigned by Dynamic Host Configuration Protocol (DHCP).

For example, if a laptop and a smart phone on the same local network both make a request to www.google.com, before the packets leave the modem, it modifies the packet headers and assigns one of its ports to that packet. When the google server responds to the requests, it sends data back to the modem at this specific port, so the modem will know whether to route the packets to the laptop or the smart phone.

In this sense, IP addresses aren’t specific to a computer, but more the connection which the computer connects to the Internet with. The address that is unique to your computer is the MAC address, which never changes throughout the life of the computer.

This protocol of mapping private IP addresses to public IP addresses is called the Network Address Translation (NAT) protocol. It’s what makes it possible to support 8+ billion networked devices with only 4 billion possible IPv4addresses.

Every router does not need to know where every IP address is. It only needs to know which one of its neighbors, called an outbound link, to route each packet to. Note that IP Addresses can be broken down into two parts, a network prefix and a host identifier. For example, 129.42.13.69 can be broken down into

Network Prefix: 129.42

Host Identifier: 13.69

All networked devices that connect to the Internet through a single connection (ie. college campus, a business, or ISP in metro area) will all share the same network prefix.

Routers will send all packets of the form 129.42.\*.\* to the same location. So instead of keeping track of billions of IP addresses, routers only need to keep track of less than a million network prefix.

A new router may come with a few preconfigured routes. But if it encounters a packet it does not know how to route, it queries one of its neighboring routers. If the neighbor knows how to route the packet, it sends that info back to the requesting router. The requesting router will save this info for future use. In this way, a new router builds up its own routing table, a database of network prefixes to outbound links. If the neighboring router does not know, it queries its neighbors and so on.

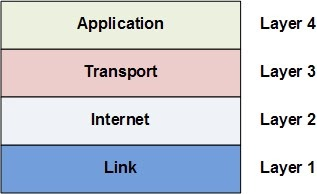
We call looking up the IP address of a human-readable domain name like www.google.com “resolving the IP address”. Computers resolve IP addresses through the Domain Name System (DNS), a decentralized database of mappings from domain names to IP addresses.

To resolve an IP address, the computer first checks its local DNS cache, which stores the IP address of web sites it has visited recently. If it can’t find the IP address there or that IP address record has expired, it queries the ISP’sDNS servers which are dedicated to resolving IP addresses. If the ISP’s DNSservers can’t resolve the IP address, they query the root name servers, which can resolve every domain name for a given top-level domain . Top-level domains are the words to the right of the right-most period in a domain name. .com .net .org are some examples of top-level domains.

Like many other complex engineering projects, the Internet is broken down into smaller independent components, which work together through well-defined interfaces. These components are called the Internet Network Layers and they consist of Link Layer, Internet Layer, Transport Layer, and Application Layer. These are called layers because they are built on top of each other; each layer uses the capabilities of the layers beneath it without worrying about its implementation details.

Internet applications work at the Application Layer and don’t need to worry about the details in the underlying layers. For example, an application connects to another application on the network via TCP using a construct called a socket, which abstracts away the gritty details of routing packets and re-assembling packets into messages.

At the lowest level is the Link Layer which is the “physical layer” of the Internet. The Link Layer is concerned with transmitting data bits through some physical medium like fiber-optic cables or wifi radio signals.



**Fig. 1.2. Internet Layers**

On top of the Link Layer is the Internet Layer. The Internet Layer is concerned with routing packets to their destinations. The Internet Protocolmentioned earlier lives in this layer (hence the same name). The Internet Protocol dynamically adjusts and reroutes packets based on network load or outages. Note it does not guarantee packets always make it to their destination, it just tries the best it can.

On top of the Internet Layer is the Transport Layer. This layer is to compensate for the fact that data can be loss in the Internet and Link layers below. The Transport Control Protocol mentioned earlier lives at this layer, and it works primarily to re-assembly packets into their original messagesand also re-transmit packets that were loss.

The Application Layer sits on top. This layer uses all the layers below to handle the complex details of moving the packets across the Internet. It lets applications easily make connections with other applications on the Internet with simple abstractions like sockets. The HTTP protocol which specifies how web browsers and web servers should interact lives in the Application Layer. The IMAP protocol which specifies how email clients should retrieve email lives in the Application Layer. The FTP protocol which specifies a file-transferring protocol between file-downloading clients and file-hosting servers lives in the Application Layer.

While clients and servers are both applications that communicate over the Internet, clients are “closer to the user” in that they are more user-facing applications like web browsers, email clients, or smart phone apps. Serversare applications running on a remote computer which the clientcommunicates over the Internet when it needs to.

A more formal definition is that the application that initiates a TCP connection is the client, while the application that receives the TCP connection is the server.

In the early days of the Internet, it was enough to ensure that the network routers and links are in physically secure locations. But as the Internet grew in size, more routers meant more points of vulnerability. Furthermore, with the advent of wireless technologies like WiFi, hackers could intercept packets in the air; it was not enough to just ensure the network hardware was physically safe. The solution to this was encryption and authenticationthrough SSL/TLS.

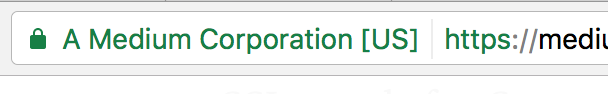
SSL stands for Secured Sockets Layer. TLS stands for Transport Layer Security. SSL was first developed by Netscape in 1994 but a later more secure version was devised and renamed TLS. We will refer to them together as SSL/TLS.

SSL/TLS is an optional layer that sits between the Transport Layer and the Application Layer. It allows secure Internet communication of sensitive information through encryption and authentication.

Encryption means the client can request that the TCP connection to the serverbe encrypted. This means all messages sent between client and server will be encrypted before breaking it into packets. If hackers intercept these packets, they would not be able to reconstruct the original message.

Authentication means the client can trust that the server is who it claims to be. This protects against man-in-the-middle attacks, which is when a malicious party intercepts the connection between client and server to eavesdrop and tamper with their communication.

We see SSL in action whenever we visit SSL-enabled websites on modern browsers. When the browser requests a web site using the https protocol instead of http, it’s telling the web server it wants an SSL encrypted connection. If the web server supports SSL, a secure encrypted connection is made and we would see a lock icon next to the address bar on the browser.



**Fig. 1.3. Example of a website with enabled SSL**

The medium.com web server is SSL-enabled. The browser can connect to it over https to ensure that communication is encrypted. The browser is also confident it is communicating with a real medium.com server, and not a man-in-the-middle.

SSL uses asymmetric encryption and SSL certificates.

Asymmetric encryption is an encryption scheme which uses a public key and a private key. These keys are basically just numbers derived from large primes. The private key is used to decrypt data and sign documents. The public key is used to encrypt data and verify signed documents. Unlike symmetric encryption, asymmetric encryption means the ability to encrypt does not automatically confer the ability to decrypt. It does this by using principles in a mathematical branch called number theory.

An SSL certificate is a digital document that consists of a public key assigned to a web server. These SSL certificates are issued to the server by certificate authorities. Operating systems, mobile devices, and browsers come with a database of some certificate authorities so it can verify SSL certificates.

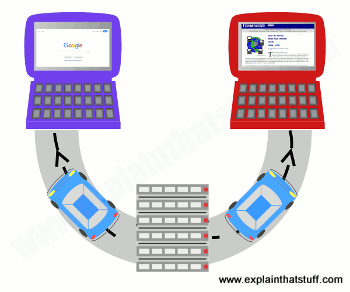
When a client requests an SSL-encrypted connection with a server, the serversends back its SSL certificate. The client checks that the SSL certificate

* is issued to this server
* is signed by a trusted certificate authority
* has not expired.

The client then uses the SSL certificate’s public key to encrypt a randomly generated temporary secret key and send it back to the server. Because the server has the corresponding private key, it can decrypt the client’s temporary secret key. Now both client and server know this temporary secret key, so they can both use it to symmetrically encrypt the messages they send to each other. They will discard this temporary secret key after their session is over.

Suppose a hacker intercepted every message sent between the client and the server. The hacker sees the SSL certificate the server sends as well as the client’s encrypted temporary secret key. But because the hacker doesn’t have the private key it can’t decrypt the temporarily secret key. And because it doesn’t have the temporary secret key, it can’t decrypt any of the messages between the client and server.

The Internet is a superhighway; the web is like traffic on that highway.

****

**Fig. 1.4. The Internet and world wide web comparison**

The Internet is like a superhighway (gray) connecting the world's computers together. The World Wide Web is a kind of traffic that can travel on that highway. In this example, two web pages (cars) are traveling from the server (the computer where they're stored) at the bottom to web browsers running on the laptops at the top. Other kinds of traffic that can travel on the Internet include email, VoIP phone calls, torrents, and files shared peer-to-peer.

Let's get one thing straight before we go any further: the Web and the Internet are two totally different things:

The Internet is a worldwide network of computers, linked mostly by telephone lines; the Web is just one of many things (called applications) that can run on the Internet. When you send an email, you're using the Internet: the Net sends the words you write over telephone lines to your friends. When you chat to someone online, you're most likely using the Internet too—because it's the Net that swaps your messages back and forth. But when you update a blog or Google for information to help you write a report, you're using the Web over the Net. You can read more in our article about how the Internet works.

The Web is the worldwide collection of text pages, digital photographs, music files, videos, and animations you can access over the Internet. What makes the Web so special (and, indeed, gives it its name) is the way all this information is connected together. The basic building blocks of the Web are pages of text, like this one—Web pages as we call them. A collection of Web pages on the same computer is called a website. Every web page (including this one) has highlighted phrases called links (or hypertext links) all over it. Clicking one of these takes you to another page on this website or another website entirely. So far, so simple.

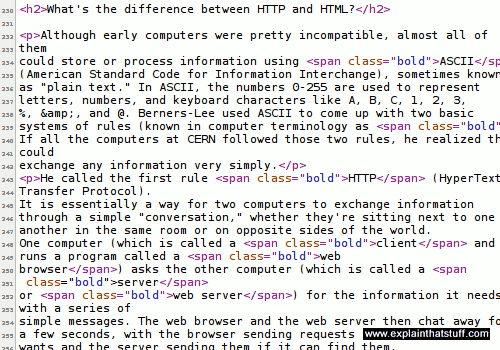
The really clever thing about the Internet is that it allows practically every computer on the planet to exchange information. That's a much bigger deal than it sounds. Back in the earlier days of computers, in the 1960s, 1970s, and 1980s, it was rare for computers to be able to exchange information at all. The machines made by one manufacturer were often totally incompatible with those made by everyone else. In the 1970s, early personal computers (which were called microcomputers) could not even run the same programs. Instead, each type of computer had to have programs written specially for it. Hooking computers up together was possible, but tricky. So most computers were used as standalone machines, like gigantic pocket calculators. Things like email and chat were all but impossible, except for a handful of scientists who knew what they were doing.

All this began to change in the 1980s. The first thing that happened was that IBM—the world's biggest computer company, famous for its "big blue" mainframes—introduced a personal computer for small businesses. Other people started to "clone" (copy) it and, pretty soon, all personal computers started to look and work the same way. Microsoft came up with a piece of software called Windows that allowed all these "IBM-compatible" computers to run the same programs. But there was a still a problem getting machines like home computers talking to giant machines in science laboratories or big mainframes in large companies. How could computers be made to talk the same language?

The person who solved that problem was English computer scientist Tim Berners-Lee (1955–). In the 1980s, he was working at CERN, the European particle physics laboratory, which is staffed mostly by people from universities around the world who come and go all the time, and where people were using all kinds of different, incompatible computers. Berners-Lee realized CERN had no "memory": every time people left, they took useful information with them. A related problem was that people who used different computers had no easy way of exchanging their research. Berners-Lee started to wonder how he could get all of CERN's computers—and people—talking together.

Although early computers were pretty incompatible, almost all of them could store or process information using ASCII (American Standard Code for Information Interchange), sometimes known as "plain text." In ASCII, the numbers 0–255 are used to represent letters, numbers, and keyboard characters like A, B, C, 1, 2, 3, %, &, and @. Berners-Lee used ASCII to come up with two basic systems of rules (known in computer terminology as protocols). If all the computers at CERN followed those two rules, he realized they could exchange any information very simply.

He called the first rule HTTP (HyperText Transfer Protocol). It is essentially a way for two computers to exchange information through a simple "conversation," whether they're sitting next to one another in the same room or on opposite sides of the world. One computer (which is called a client and runs a program called a web browser) asks the other computer (which is called a server or web server) for the information it needs with a series of simple messages. The web browser and the web server then chat away for a few seconds, with the browser sending requests for the things it wants and the server sending them if it can find them. The HTTP conversation between a web browser and and a web server is a bit like being at a dinner table when someone says: "Pass the salt, please", someone else says "Here it is", and the first person says "Thank you." HTTP is a sort of simple, polite language that all computers have learned to speak so they can swap files back and forth over the Internet.



**Fig. 1.5. An example of HTML code.**

A computer also needs to be able to understand any files it receives that have been sent by HTTP. So Berners-Lee introduced another stroke of genius. His second rule was to make all the CERN computers exchange files written in a common language called HTML (HyperText Markup Language). It was based on ASCII, so any computer could understand it. Unlike ASCII, HTML has special codes called tags to structure the text. A Web browser can read these tags and use them to display things like bold font, italics, headings, tables, or images. Incidentally, for the curious among you: you can see what the "secret" HTML behind any web page looks like by right clicking your mouse on a web page and then selecting the View source or View page source option. Try it now!

HTTP and HTML are "how the Web works": HTTP is the simple way in which one computer asks another one for Web pages; HTML is the way those pages are written so any computer can understand them and display them correctly. If you find that confusing, try thinking about libraries. HTTP is like the way we arrange and access books in libraries according to more or less the same set of rules: the fact that they have books arranged on shelves, librarians you can ask for help, catalogs where you can look up book titles, and so on. Since all libraries work roughly the same way, if you've been to one library, you know roughly what all the others are like and how to use them. HTML is like the way a book is made: with a contents at the front, an index at the back, text on pages running left to right, and so on. HTML is how we structure information so anyone can read it. Once you've seen one book, you know how they all work.

Web browsers (clients) and servers converse not in English, French, or German—but HTTP: the language of "send me a Web page", "Okay, here it is." This is a brief example of how your browser could ask to see our A-Z index page and what our server would say in response. The actual page and its information is sent separately.

What the browser asks for

GET /azindex.html HTTP/1.1

Host: www.explainthatstuff.com,

User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:44.0) Gecko/20100101 Firefox/44.0

Accept: text/html, application/xhtml+xml, application/xml; q=0.9,\*/\*;q=0.8,

Accept-Language: en-gb,en;q=0.5,

Accept-Encoding: gzip,deflate,

Accept-Charset: ISO-8859-1,utf-8;q=0.7,\*;q=0.7,

Keep-Alive: 300,

Connection: keep-alive,

What the server replies

HTTP/1.1 200 OK

Date: Thu, 18 Jan 2018 09:03:23 GMT

Server: Apache

Expires: Sun, 19 Mar 2018 09:03:23 GMT

Content-Encoding: gzip

Content-Length: 19702

Content-Type: text/html; charset=UTF-8

What does it all mean? Briefly, the browser is explaining what software it is (Firefox), what operating system I'm running (Linux Ubuntu), which character-sets (foreign fonts and so on) it can accept, which forms of compressed file it can understand (gzip, deflate), and which file it wants (azindex.html). The server (running software called Apache) is sending a compressed file (gzip), along with data about how long it is (19702 bytes) and what format it's in (text/html, using the UTF-8 character set).

Right at the start of the server's reply, you can see it says HTTP/1.1 200 OK: the 200 "status code" (sometimes called a response code) means the server has correctly located the page and is sending it to the browser. A server can send a variety of other numeric codes too: if it can't find the page, it sends a 404 "Not Found" code; if the page has moved elsewhere, the server sends a 301 "Permanently moved" code and the address of the page's new location; and if the server is down for maintenance, it can send a 503 "Service Unavailable" code, which tells browsers they should try again later.

There was one more clever thing Berners-Lee thought of—and that was a way for any computer to locate information stored on any other computer. He suggested each web page should have something like a zip code, which he called a URL(a Universal or Uniform Resource Locator). The URL is the page address you see in the long bar at the top of your Web browser.

The address or URL of page is: https:// www.explainthatstuff.com/ howthewebworks.html

What does all that gobbledygook mean? Let's take it one chunk at a time:

The http:// bit means your computer can pull this page off my computer using the standard process called HTTP. If the URL begins with https, as it does for this page, communication is encrypted as it travels between your browser and the Web server (so things like credit-card numbers, user names, passwords, and so on are kept secure from interference in transit). https pages are inherently more secure than http pages, but https alone does not make a website completely secure: it simply secures the connection between your computer and the server (or servers) you're talking to.

www.explainthatstuff.com is the address or domain name of my computer. Some websites use domain names that begin with things other than www (for example maps.google.com and mail.yahoo.com), which are called subdomains. maps.google.com, drive.google.com, and indeed www.google.com are all subdomains of the main google.com domain.

howthewebworks.html is the name of the file you're currently reading off my computer.

The .html part of the filename tells your computer it's an HTML file. Other filenames you might see include .php and .asp, which mean the pages you're looking at are "dynamic"; unlike "static" HTML pages, dynamic pages are built specifically for you, at the moment you request them, by the web server.

Taken all together, that stuff tells your computer where to find this page on my computer, how to access it, and what to do with it to display it correctly.

And that's how the Web works!

The famous American inventor and publisher Benjamin Franklin once said that two things in life are certain: death and taxes. These days, he might add something else to that list: websites—because just about everybody seems to have one! Businesses promote themselves with websites, television soaps have spinoff sites devoted to their characters, newlyweds set up sites for their wedding photographs, and most kids have profiles (statements about themselves and what they like) on "social-networking sites" such as Facebook. If you feel like you're getting left behind, maybe it's time to set up a site yourself? How do you go about it?

The basic idea of the Web is that you can read information that anyone else has stored on a publicly accessible space called their website. If you're familiar with using computers for wordprocessing, you'll know that when you create a document (such as a letter or a CV/resumé), it exists on your computer as a file, which you store in a place called a folder (or directory). A website is simply a collection of interlinked documents, usually stored in the same directory on a publicly accessible computer known as a server. Apart from the main documents (text pages), a website generally also contains images or graphic files (photographs, typically stored as JPG files, and artworks, usually stored as GIF or PNG files). So the basic idea of creating a website involves writing all these text pages and assembling the various graphic files you need, then putting them all together in a folder where other people can access them

Theoretically, you could turn your own computer into a server and allow anyone else on the planet to access it to browse your website. All you have to do is configure your computer in a certain way so that it accepts incoming traffic from the Internet and also register your computer with all the other servers on the Internet so they know where to find it. There are three main reasons why this is not generally a good idea. First, you won't be able to use your computer for anything else because it will be spending all its time serving requests for information from other people. (But if you have more than one computer, that's not such a problem.) Second, you'd have to make sure that your computer was switched on and available 24 hours a day—and you might not want to do that. Third, making your computer available to the Internet in this way is something of a security risk. A determined hacker might be able to access all the other folders on your machine and either steal your information or do other kinds of malicious damage.

So, in practice, people rent web space on a large computer operated by an Internet service provider (ISP). This is known as getting someone to host your website for you. Generally, if you want to set up a website, you will need a hosting package (a basic contract with an ISP to give you so much disk space and bandwidth (the maximum amount of information that your website can transfer out to other people each month). The web space you get is simply a folder (directory) on the ISPs server and it will have a fairly obscure and unmemorable name such as: www.example.com/ABC54321/ That's not exactly the sort of thing you want to paint on the side of your truck, if you're in business. So you'll need a more memorable name for your website—also called a domain name. The domain name is simply a friendly address that you give to your website so that other people can find it more easily. The domain address is set up to point to the real address of your site at your ISPs server (www.example.com/ ABC54321/ ), so when people type your domain name into their Web browser, they are automatically redirected to the correct address without actually having to worry about what it is.

Some ISPs offer a user-friendly system where you simply purchase a domain name and hosting package for a single annual payment (generally, it will be less than about $60 or £30 per annum). With other ISPs, you have to buy the domain name and the hosting package separately and that works out better if you are hosting several different domains with the same ISP. Buying a domain name makes you its legal owner and you'll find that you are immediately registered on a central database known as WHOIS, so that other people can't use the same name as well.

Setting up a domain name and Web hosting package takes all of five minutes; creating a website can take an awful lot longer because it means writing all the information you need, coming up with a nice page layout, finding your photographs, and all the rest of it. Generally, there are three ways to create web pages.

The most basic way of creating web pages is to use a text editor such as notepad or WordPad on Windows and build up your pages from raw HTML web page coding as you go. Generally, this gives you a much better understanding of how web pages work, but it's a bit harder for novices to get the hang of it—and unless you're a geek you may not want to bother. Instead of creating pages from scratch, you can use ready-made ones called templates. They're bare-bones, pre-designed HTML files into which you simply insert your own content. Just change the bits you need and you have an instant website! The main drawback of templates is that you can end up with a me-too site that looks the same as everyone else's.

Another approach is to use an editing program that does all the hidden Web-page coding (known as HTML) for you. This is called a WYSIWYG (what you see is what you get) editor because you lay out your pages on the screen broadly as you want them to appear to everyone who browses your site. Popular programs such as Dreamweaver work in this way. Most word processors, including Microsoft Word and OpenOffice, let you convert existing documents into web pages ("export HTML files") with a couple of mouse clicks.

The final method is to use what's called a content-management system (CMS), which handles all the technical side of creating a website automatically. You simply set up a basic page template, style its visual appearance with what's called a "theme," create your various interlinked pages based on the template, and then upload them. CMS systems like Wordpress, Drupal, and Joomla (and less sophisticated ones such as Weebly and Wix) work this way. You can add various extra functions to them using what are called plugins.

Once you've created your web pages and you have your domain name and web space, you simply need to upload the pages onto your web space using a method called FTP (file transfer protocol). It's very easy: just like copying files from one folder of your computer to another. When you've uploaded your files, your website should be publicly accessible within seconds (assuming that your domain name has already been registered for at least a couple of days first). Updating your web pages is then simply a matter of updating them on your local computer, as often as you wish, and copying the changes onto your web space as necessary. Generally it's best to do all your updating on copies of your pages on your own computer rather than editing live pages on the server itself. You avoid embarrassing mistakes that way, but you also have a useful backup copy of the entire site on your computer in case the server crashes and loses all your files.

You want lots of other people to find your website, so you'll need to encourage other websites to make links to yours. You'll also need to register your site with search engines such as Google, Bing, and all the dozens of others. Sooner or later, search engines like Google will pick up your site if it's linked by other sites that they're already indexing, because they're constantly "crawling" the web looking for new content.

And that's pretty much all there is to it. The best way to learn about websites is to build one for yourself. So, off you go and do it! You can learn all about building basic web pages by playing with HTML files on your computer. Once you're confident about what you're doing, it's easy to take the next step and make a world-wide website for the whole wide world!

**Questions**

1. What is and IP?
2. What is included into server response?
3. What does HTML stands for?

## 

## TOPIC 2

INTRODUCTION TO HTML TAGS,ATTRIBUTES AND ELEMENTS. TITLES, PARAGRAPHS, HEADINGS, LISTS AND LINKS

Most of the stuff on the web is no different than the stuff on your computer — it’s just a whole load of files sorted into a whole load of directories. HTML files are nothing more than simple text files, so to start writing in HTML, you need nothing more than a simple text editor.

Notepad is a common text editor on Windows-based computers (usually found under the Programs > Accessories menu) and Mac OSX computers come bundled with TextEdit but any program that lets you fiddle with text will do.

Type this in to your text editor:

This is my first web page

Now create a folder called “html” wherever you like to save files on your computer and save the file as “myfirstpage.html”.

To look at HTML files, they don’t even need to be on the web. Open a web browser such as Chrome, Firefox, Safari or Internet Explorer and in the address bar, where you usually type web addresses, type in the location of the file you just saved (for example, “c:\html\myfirstpage.html”) and hit return. Alternatively, go to the File menu of the browser, select Open, and browse for the file.

Pow. There it is. Your first web page. How exciting. And all it took was a few typed words.

We’ve said here to use a basic text-editor, such as Notepad, but you may be tempted to use a dedicated software program such as Adobe Dreamweaver.

You should be very careful when using these programs, especially if you are a beginner, because they often throw in unnecessary or non-standard code to “help” you.

If you’re serious about learning HTML, you should read through a tutorial such as this first, so that you at least have a basic understanding of what is going on.

Software programs such as these will never give you the same control over a web page as coding by hand.

If you do decide to use specialized code-editing software, we recommend one in which you are still coding by hand. They can, in fact, be helpful, especially the more advanced you become, in terms of code syntax highlighting and file management.

Although the basics of HTML is plain text, we need a bit more to make it a nice and shiny HTML document.

The basic structure of an HTML document includes tags, which surround content and apply meaning to it.

Change your document so that it looks like this:

<!DOCTYPE html>

<html>

<body>

This is my first web page

</body>

</html>

Now save the document again, go back to the web browser and reload the page.

The appearance of the page will not have changed at all, but the purpose of HTML is to apply meaning, not presentation, and this example has now defined some fundamental elements of a web page.

The first line on the top, <!DOCTYPE html>, is a document type declaration and it lets the browser know which flavor of HTML you’re using (HTML5, in this case). It’s very important to stick this in - If you don’t, browsers will assume you don’t really know what you’re doing and act in a very peculiar way.

To get back to the point, <html> is the opening tag that kicks things off and tells the browser that everything between that and the </html> closing tag is an HTML document. The stuff between <body> and </body> is the main content of the document that will appear in the browser window.

The </body> and </html> put a close to their respective elements (more on elements in a moment).

Not all tags have closing tags like this (<html></html>) some tags, which do not wrap around content will close themselves. The line-break tag for example, looks like this : <br> - a line break doesn’t hold any content so the tag merrily sits by its lonely self. We will come across these examples later. All you need to remember is that all tags with content between them should be closed, in the format of opening tag → content → closing tag. It isn’t, strictly speaking, always a requirement, but it’s a convention we’re using in these tutorials because it’s good practice that results in cleaner, easier to understand code.

You might come across “self-closing” tags, whereby a br tag, for example, will look like “<br />” instead of simply “<br>”. This is a remnant of XHTML, a form of HTML based on another markup language called XML. HTML5 is much more chilled out than XHTML and will be happy with either format. Some developers prefer one way, some prefer the other. We tossed a coin and decided to stick with the simpler version.

Tags can also have attributes, which are extra bits of information. Attributes appear inside the opening tag and their values sit inside quotation marks. They look something like <tag attribute="value">Margarine</tag>. We will come across tags with attributes later.

Once again, quotation marks aren’t always essential but it is a good-practice convention HTML Dog uses for consistency and clarity. We suggest you do the same.

Tags tend not to do much more than mark the beginning and end of an element. Elements are the bits that make up web pages. You would say, for example, that everything that is in between (and includes) the <body> and </body> tags is the body element. As another example, whereas “<title>” and “</title>” are tags, “<title>Rumple Stiltskin</title>” is a title element.

All HTML pages should have a page title.

To add a title to your page, change your code so that it looks like this:

<!DOCTYPE html>

<html>

<head>

<title>My first web page</title>

</head>

<body>

This is my first web page

</body>

</html>

We have added two new elements here, that start with the head tag and the title tag (and see how both of these close).

The head element (that which starts with the <head> opening tag and ends with the </head> closing tag) appears before the body element (starting with <body> and ending with </body>) and contains information about the page. The information in the head element does not appear in the browser window.

We will see later on that other elements can appear inside the head element, but the most important of them is the title element.

If you look at this document in the browser (save and reload as before), you will see that “My first web page” will appear on a tab or the title bar of the window (not the actual canvas area). The text that you put in between the title tags has become the title of the document (surprise!). If you were to add this page to your “favorites” (or “bookmarks”, depending on your browser), you would see that the title is also used there.

Now that you have the basic structure of an HTML document, you can mess around with the content a bit.

Go back to your text editor and add another line to your page:

<!DOCTYPE html>

<html>

<head>

<title>My first web page</title>

</head>

<body>

This is my first web page

How exciting

</body>

</html>

Look at the document in your browser.

You might have expected your document to appear as you typed it, on two lines, but instead you should see something like this:

This is my first web page How exciting.

This is because web browsers don’t usually take any notice of what line your code is on. It also doesn’t take any notice of spaces (you would get the same result if you typed “This is my first web page How exciting”).

If you want text to appear on different lines or, rather, if you intend there to be two distinct blocks of text (because, remember, HTML is about meaning, not presentation), you need to explicitly state that.

Change your two lines of content so that they look like this:

<p>This is my first web page</p>

<p>How exciting</p>

The p tag is used for paragraphs.

Look at the results of this. The two lines will now appear on two lines because the browser recognizes them as separate paragraphs.

Think of the HTML content as if it were a book - with paragraphs where appropriate.

You can emphasize text in a paragraph using em (emphasis) and strong (strong importance).

<p>Yes, that really <em>is</em> exciting. <strong>Warning:</strong> level of excitement may cause head to explode.</p>

Traditionally, browsers will display em in italics and strong in bold by default but they are not the same as i and b tags which (although they have been tenuously redefined in HTML5) have their origins in italic and bold - remember - HTML isn’t for presentation. If you want to emphasize something visually (making something italic, for example), you almost certainly want to give it general emphasis. You can’t speak in italics.

The line-break tag can also be used to separate lines like this:

This is my first web page<br>

How exciting

There’s no content involved in breaking lines so there is no closing tag.

It could be tempting to over-use line breaks and br shouldn’t be used if two blocks of text are intended to be separate from one another (because if that’s what you want to do you probably want the p tag).

The p tag is just the start of text formatting.

If you have documents with genuine headings, then there are HTML tags specifically designed just for them.

They are h1, h2, h3, h4, h5 and h6, h1 being the almighty emperor of headings and h6 being the lowest pleb.

Change your code to the following:

<!DOCTYPE html>

<html>

<head>

<title>My first web page</title>

</head>

<body>

<h1>My first web page</h1>

<h2>What this is</h2>

<p>A simple page put together using HTML</p>

<h2>Why this is</h2>

<p>To learn HTML</p>

</body>

</html>

Note that the h1 tag is only used once, as the main heading of the page. h2 to h6, however, can be used as often as desired, but they should always be used in order, as they were intended. For example, an h4 should be a sub-heading of an h3, which should be a sub-heading of an h2.

There are three types of list; unordered lists, ordered lists and definition lists. We will look at the first two here, and description lists in the next lectures.

Unordered lists and ordered lists work the same way, except that the former is used for non-sequential lists with list items usually preceded by bullets and the latter is for sequential lists, which are normally represented by incremental numbers.

The ul tag is used to define unordered lists and the ol tag is used to define ordered lists. Inside the lists, the li tag is used to define each list item.

Change your code to the following:

<!DOCTYPE html>

<html>

<head>

<title>My first web page</title>

</head>

<body>

<h1>My first web page</h1>

<h2>What this is</h2>

<p>A simple page put together using HTML</p>

<h2>Why this is</h2>

<ul>

<li>To learn HTML</li>

<li>To show off</li>

<li>Because I've fallen in love with my computer and want to give her some HTML loving.</li>

</ul>

</body>

</html>

If you look at this in your browser, you will see a bulleted list. Simply change the [ul](https://www.htmldog.com/references/html/tags/ul/) tags to [ol](https://www.htmldog.com/references/html/tags/ol/) and you will see that the list will become numbered.

Lists can also be included in lists to form a structured hierarchy of items.

Replace the above list code with the following:

<ul>

<li>To learn HTML</li>

<li>

To show off

<ol>

<li>To my boss</li>

<li>To my friends</li>

<li>To my cat</li>

<li>To the little talking duck in my brain</li>

</ol>

</li>

<li>Because I've fallen in love with my computer and want to give her some HTML loving.</li>

</ul>

Et voilà. A list within a list. And you could put another list within that. And another within that. And so on and so forth.

So far you’ve been making a stand-alone web page, which is all very well and nice, but what makes the Internet so special is that it all links together.

The “H” and “T” in “HTML” stand for “hypertext”, which basically means a system of linked text.

An anchor tag (a) is used to define a link, but you also need to add something to the anchor tag — the destination of the link.

Add this to your document:

<!DOCTYPE html>

<html>

<head>

<title>My first web page</title>

</head>

<body>

<h1>My first web page</h1>

<h2>What this is</h2>

<p>A simple page put together using HTML</p>

<h2>Why this is</h2>

<p>To learn HTML</p>

<h2>Where to find the tutorial</h2>

<p><a href="http://www.htmldog.com">HTML Dog</a></p>

</body>

</html>

The destination of the link is defined in the href attribute of the tag. The link can be absolute, such as “http://www.htmldog.com”, or it can be relative to the current page.

So if, for example, you had another file called “flyingmoss.html” in the same directory then the line of code would simply be <a href="flyingmoss.html">The miracle of moss in flight</a> or something like this.

A link does not have to link to another HTML file, it can link to any file anywhere on the web.

A link can also send a user to another part of the same page they are on. You can add an id attribute to just about any tag, for example <h2 id="moss">Moss</h2>, and then link to it by using something like this: <a href="#moss">Go to moss</a>. Selecting this link will scroll the page straight to the element with that ID.

**Questions**

1. Name the examples of tags that don’t require closing.
2. How many Headings types are there in HTML?
3. Explain the structure of link tag.

## 

## TOPIC 3

IMAGES, TABLES AND FORMS IN HTML

Things might seem a little bland and boring with all of this text formatting. Of course, the web is not just about text, it is a multi-media extravaganza and the most common form of sparkle is the image.

The img tag is used to put an image in an HTML document and it looks like this:

<img src="http://www.htmldog.com/badge1.gif" width="120" height="90" alt="HTML Dog">

The src attribute tells the browser where to find the image. Like the [a](https://www.htmldog.com/references/html/tags/a/) tag, this can be absolute, as the above example demonstrates, but is usually relative. For example, if you create your own image and save it as “alienpie.jpg” in a directory called “images” then the code would be <img src="images/alienpie.jpg"...

The width and height attributes are necessary because if they are excluded, the browser will tend to calculate the size as the image loads, instead of when the page loads, which means that the layout of the document may jump around while the page is loading.

The alt attribute is the alternative description. This is an accessibility consideration, providing meaningful information for users who are unable to see the image (if they are visually impaired, for example).

Note that, like the br tag, because the img element does not enclose any content, no closing tag is required.

The construction of images for the web is a little outside of the remit of this website, but it is worth noting a few things…

The most commonly used file formats used for images are JPEGs, GIFs, and PNGs. They are compressed formats, and have very different uses.

A JPEG (pronounced “jay-peg”) uses a mathematical algorithm to compress the image and will distort the original slightly. The lower the compression, the higher the file size, but the clearer the image.

JPEGs are typically used for images such as photographs.

A GIF (pronounced “jif”) can have no more than 256 colors, but they maintain the colors of the original image. The lower the number of colors you have in the image, the lower the file size will be. GIFs also allow any pixel in the image to be transparent.

GIFs are typically used for images with solid colors, such as icons or logos.

A PNG (pronounced “ping”) replicates colors, much like a GIF, but allows 16 million colors as well as alpha transparency (that is, an area could be 50% transparent).

PNGs are typically used for versatile images in more complex designs BUT they are not fully supported by some older browsers.

The web is forever getting faster and faster but you obviously want your web pages to download as quickly as possible. Using super-high resolution images isn’t doing your or your user’s bandwidth (or patience) any favors. Image compression is a great tool and you need to strike a balance between image quality and image size. Most modern image manipulation programs allow you to compress images and the best way to figure out what is best suited for yourself is trial and error.

HTML tables are still best known for being used and abused to lay out pages. The correct use for tables is to do exactly what you would expect a table to do — to structure tabular data.

There are a number of tags used in tables, and to fully get to grips with how they work is probably the most difficult area of this HTML Beginner Tutorial.

Copy the following code into the body of your document and then we will go through what each tag is doing:

<table>

<tr>

<td>Row 1, cell 1</td>

<td>Row 1, cell 2</td>

<td>Row 1, cell 3</td>

</tr>

<tr>

<td>Row 2, cell 1</td>

<td>Row 2, cell 2</td>

<td>Row 2, cell 3</td>

</tr>

<tr>

<td>Row 3, cell 1</td>

<td>Row 3, cell 2</td>

<td>Row 3, cell 3</td>

</tr>

<tr>

<td>Row 4, cell 1</td>

<td>Row 4, cell 2</td>

<td>Row 4, cell 3</td>

</tr>

</table>

The table element defines the table.

The tr element defines a table row.

The td element defines a data cell. These must be enclosed in tr tags, as shown above.

If you imagine a 3x4 table, which is 12 cells, there should be four [tr](https://www.htmldog.com/references/html/tags/tr/) elements to define the rows and three td elements within each of the rows, making a total of 12 [td](https://www.htmldog.com/references/html/tags/td/) elements.

It can be quite difficult to visualize a two-dimensional grid from one-dimensional lines of code.

Well, it gets trickier. All thanks to the rowspan and colspan attributes. Those bastards.

The following code is similar to that in before:

<table>

<tr>

<th>Column 1 heading</th>

<th>Column 2 heading</th>

<th>Column 3 heading</th>

</tr>

<tr>

<td>Row 2, cell 1</td>

<td colspan="2">Row 2, cell 2, also spanning Row 2, cell 3</td>

</tr>

<tr>

<td rowspan="2">Row 3, cell 1, also spanning Row 4, cell 1</td>

<td>Row 3, cell 2</td>

<td>Row 3, cell 3</td>

</tr>

<tr>

<td>Row 4, cell 2</td>

<td>Row 4, cell 3</td>

</tr>

</table>

The first difference is that the td tags of the first row have been replaced with [th](https://www.htmldog.com/references/html/tags/th/) tags. Whereas a td is a standard data cell, th is a header cell. As with td elements, these must be enclosed inside [tr](https://www.htmldog.com/references/html/tags/tr/) elements.

colspan and rowspan attributes have also been used in some of the td tags. If you look at this code in a browser, you will see that on the second row there are now two cells instead of three, the second cell spanning the second and third column. The colspan attribute, which means “column span” will span the cell over the number of cells that is specified. This means, in this example, that there is no need for a third td element — two cells are merged into one.

The rowspan attribute is similar to colspan, except, obviously, it will span across rows rather than columns. Again, the cells that it spans should be removed. In this example, because it spans over the fourth row, there are only two cells in that row.

As with the simpler 3x4, 12-cell table, the numbers on these tables with merged cells should also add up. Although there are only 10 cells in this example, there are 2 spans.

So you think you know how to make a table. Sure, you know the table, tr, td and th tags, you’ve even got the rowspan and colspan attributes in your pocket. You can make a really cute little plywood coffee table, but don’t you want to know how to make one of those polished solid wood, glass top dining tables that can take the weight of an oversized elephant?

You do? Oh joy.

Table rows tend to make table columns look rather stupid. They do all the work, as the table is built row by row, leaving the columns feeling quite rejected.

Luckily for those eager columns though, the colgroup and col tags have come to their rescue.

These tags allow you to define the table columns and style them as desired, which is particularly useful if you want certain columns aligned or colored differently, as, without this, you would need to target individual cells.

<table>

<colgroup>

<col>

<col class="alternative">

<col>

</colgroup>

<tr>

<td>This</td>

<td>That</td>

<td>The other</td>

</tr>

<tr>

<td>Ladybird</td>

<td>Locust</td>

<td>Lunch</td>

</tr>

</table>

In this example the styles of the CSS class “alternative” will be applied to the second column, or the second cell in every row.

You can also use the span attribute in a similar way to rowspan and colspan. Using it with the colgroup tag will define the number of rows that the column group will belong to, for example <colgroup span="2"></colgroup> would group the first two columns. Using span in the col tag is usually more useful, and could, for example, be applied to the above example like this:

<table>

<colgroup>

<col>

<col span="2" class="alternative">

</colgroup>

<!-- and so on -->

This would apply “alternative” to the last two columns.

When span is used in colgroup, you shouldn’t then use col tags.

A brief and easy accessibility consideration is to apply a caption to the table. The caption element defines the caption and should be used straight after the opening table tag.

<table>

<caption>Locust mating habits</caption>

<!-- etc. -->

A caption will appear above the table by default, although using the CSS caption-side: bottom will, well, do what you would expect.

The mightier figcaption would be preferable to caption if you are marking up a table as the sole content of a figure element. See the Sectioning Chapter for more.

thead, tfoot and tbody allow you to separate the table into header, footer and body, which can be handy when dealing with larger tables.

Whereas thead needs to come first, tfoot can, in fact come before a tbody (and you can have more than one tbody, if it takes your fancy) although browsers will render the tfoot element at the bottom of the table.

<table>

<thead>

<tr>

<td>Header 1</td>

<td>Header 2</td>

<td>Header 3</td>

</tr>

</thead>

<tfoot>

<tr>

<td>Footer 1</td>

<td>Footer 2</td>

<td>Footer 3</td>

</tr>

</tfoot>

<tbody>

<tr>

<td>Cell 1</td>

<td>Cell 2</td>

<td>Cell 3</td>

</tr>

<!-- etc. -->

</tbody>

</table>

Forms are used to collect data inputted by a user. They can be used as an interface for a web application, for example, or to send data across the web.

On their own, forms aren’t usually especially helpful. They tend to be used in conjunction with a programming language to process the information inputted by the user. These scripts take all manner of guises that are largely outside of the remit of this website because they require languages other than HTML and CSS.

The basic tags used in the actual HTML of forms are form, input, textarea, select and option.

form defines the form and within this tag, if you are using a form for a user to submit information (which we are assuming at this level), an action attribute is needed to tell the form where its contents will be sent to.

The method attribute tells the form how the data in it is going to be sent and it can have the value get, which is default, and latches the form information onto a web address, or post, which (essentially) invisibly sends the form’s information.

get is used for shorter chunks of non-sensitive information - you might see the information you have submitted in a web site’s search to appear in the web address of its search results page, for example. post is used for lengthier, more secure submissions, such as in contact forms.

So a form element will look something like this:

<form action="processingscript.php" method="post">

</form>

The input tag is the daddy of the form world. It can take a multitude of guises, the most common of which are outlined below (see the input reference page for the whole crazy family):

* <input type="text"> or simply <input> is a standard textbox. This can also have a value attribute, which sets the initial text in the textbox.
* <input type="password"> is similar to the textbox, but the characters typed in by the user will be hidden.
* <input type="checkbox"> is a checkbox, which can be toggled on and off by the user. This can also have a checked attribute (<input type="checkbox" checked> - the attribute doesn’t require a value), and makes the initial state of the check box to be switched on, as it were.
* <input type="radio"> is similar to a checkbox, but the user can only select one radio button in a group. This can also have a checked attribute.
* <input type="submit"> is a button that when selected will submit the form. You can control the text that appears on the submit button with the value attribute, for example <input type="submit" value="Ooo. Look. Text on a button. Wow">.

Note that, like img and br tags, the input tag, which doesn’t surround any content, doesn’t require a closing tag.

textarea is, basically, a large, multi-line textbox. The anticipated number of rows and columns can be defined with rows and cols attributes, although you can manipulate the size to your heart’s content using CSS.

<textarea rows="5" cols="20">A big load of text</textarea>

Any text you choose to place between the opening and closing tags (in this case “a big load of text”) will form the initial value of the text area.

The select tag works with the option tag to make drop-down select boxes.

<select>

<option>Option 1</option>

<option>Option 2</option>

<option value="third option">Option 3</option>

</select>

When the form is submitted, the value of the selected option will be sent. This value will be the text between the selected opening and closing option tag unless an explicit value is specified with the value attribute, in which case this will be sent instead. So, in the above example, if the first item is selected, “Option 1” will be sent, if the third item is selected, “third option” will be sent.

Similar to the checked attribute of checkboxes and radio buttons, an option tag can also have a selected attribute, to start off with one of the items already being selected, eg. <option selected>Rodent</option> would pre-select “Rodent” from the items.

All of the tags mentioned above will look very nice presented on the page but if you hook up your form to a form-handling script, they will all be ignored. This is because the form fields need names. So to all of the fields, the attribute name needs to be added, for example <input type="text" name="talkingsponge">.

A contact form for Noah’s Ark, for example, might look something like the one below. (Note: this form will not work unless there is a “contactus.php” file, which is stated in the action attribute of the form tag, to handle the submitted data).

<form action="contactus.php" method="post">

<p>Name:</p>

<p><input type="text" name="name" value="Your name"></p>

<p>Species:</p>

<p><input name="species"></p>

<!-- remember: 'type="text"' isn't actually necessary -->

<p>Comments: </p>

<p><textarea name="comments" rows="5" cols="20">Your comments</textarea></p>

<p>Are you:</p>

<p><input type="radio" name="areyou" value="male"> Male</p>

<p><input type="radio" name="areyou" value="female"> Female</p>

<p><input type="submit"></p>

</form>

Phew.

**Questions**

1. Which tags are used to format table header and footer?
2. What is the purpose of “name” attribute in form?
3. What is the difference between using the image from the website subfolder and external image?

## 

## TOPIC 4

INTRODUCTION TO CSS

There are three ways to apply CSS to HTML: Inline, internal, and external.

Inline styles are plonked straight into the HTML tags using the style attribute.

They look something like this:

<p style="color: red">text</p>

This will make that specific paragraph red.

But, if you remember, the best-practice approach is that the HTML should be a stand-alone, presentation free document, and so in-line styles should be avoided wherever possible.

Embedded, or internal, styles are used for the whole page. Inside the head element, the style tags surround all of the styles for the page.

<!DOCTYPE html>

<html>

<head>

<title>CSS Example</title>

<style>

p {

color: red;

}

a {

color: blue;

}

</style>

...

This will make all of the paragraphs in the page red and all of the links blue.

Although preferable to soiling our HTML with inline presentation, it is similarly usually preferable to keep the HTML and the CSS files separate, and so we are left with our savior…

External styles are used for the whole, multiple-page website. There is a separate CSS file, which will simply look something like:

p {

color: red;

}

a {

color: blue;

}

If this file is saved as “style.css” in the same directory as your HTML page then it can be linked to in the HTML like this:

<!DOCTYPE html>

<html>

<head>

<title>CSS Example</title>

<link rel="stylesheet" href="style.css">

…

To get the most from this guide, it would be a good idea to try out the code as we go along, so start a fresh new file with your text-editor and save the blank document as “style.css” in the same directory as your HTML file.

Now change your HTML file so that it starts something like this:

<!DOCTYPE html>

<html>

<head>

<title>My first web page</title>

<link rel="stylesheet" href="style.css">

</head>

...

Save the HTML file. This now links to the CSS file, which is empty at the moment, so won’t change a thing. As you work your way through the CSS Beginner Tutorial, you will be able to add to and change the CSS file and see the results by simply refreshing the browser window that has the HTML file in it, as we did before.

Whereas HTML has tags, CSS has selectors. Selectors are the names given to styles in internal and external style sheets. In this CSS Beginner Tutorial we will be concentrating on HTML selectors, which are simply the names of HTML tags and are used to change the style of a specific type of element.

For each selector there are “properties” inside curly brackets, which simply take the form of words such as color, font-weight or background-color.

A value is given to the property following a colon (NOT an “equals” sign). Semi-colons are used to separate the properties.

body {

font-size: 14px;

color: navy;

}

This will apply the given values to the font-size and color properties to the body selector.

So basically, when this is applied to an HTML document, text between the body tags (which is the content of the whole window) will be 14 pixels in size and navy in color.

There are many property-specific units for values used in CSS, but there are some general units that are used by a number of properties and it is worth familiarizing yourself with these before continuing.

* px (such as font-size: 12px) is the unit for pixels.
* em (such as font-size: 2em) is the unit for the calculated size of a font. So “2em”, for example, is two times the current font size.
* pt (such as font-size: 12pt) is the unit for points, for measurements typically in printed media.
* % (such as width: 80%) is the unit for… wait for it… percentages.

Other units include pc (picas), cm (centimeters), mm (millimeters) and in (inches).

When a value is zero, you do not need to state a unit. For example, if you wanted to specify no border, it would be border: 0.

“px” in this case, doesn’t actually necessarily mean pixels - the little squares that make up a computer’s display - all of the time. Modern browsers allow users to zoom in and out of a page so that, even if you specify font-size: 12px, or height: 200px, for example, although these will be the genuine specified size on a non-zoomed browser, they will still increase and decrease in size with the user’s preference. It’s a good thing. Trust me.

CSS brings 16,777,216 colors to your disposal. They can take the form of a name, an RGB (red/green/blue) value or a hex code.

The following values, to specify full-on as red-as-red-can-be, all produce the same result:

* red
* rgb(255,0,0)
* rgb(100%,0%,0%)
* #ff0000
* #f00

Predefined color names include aqua, black, blue, fuchsia, gray, green, lime, maroon, navy, olive, purple, red, silver, teal, white, and yellow. transparent is also a valid value.

With the possible exception of black and white, color names have limited use in a modern, well-designed web sites because they are so specific and limiting.

The three values in the RGB value are from 0 to 255, 0 being the lowest level (no red, for example), 255 being the highest level (full red, for example). These values can also be a percentage.

Hexadecimal (previously and more accurately known as “sexadecimal”) is a base-16 number system. We are generally used to the decimal number system (base-10, from 0 to 9), but hexadecimal has 16 digits, from 0 to f.

The hex number is prefixed with a hash character (#) and can be three or six digits in length. Basically, the three-digit version is a compressed version of the six-digit (#ff0000 becomes #f00, #cc9966 becomes #c96, etc.). The three-digit version is easier to decipher (the first digit, like the first value in RGB, is red, the second green and the third blue) but the six-digit version gives you more control over the exact color.

Colors can be applied by using color and background-color (note that this must be the American English “color” and not “colour”).

A blue background and yellow text could look like this:

h1 {

color: yellow;

background-color: blue;

}

These colors might be a little too harsh, so you could change the code of your CSS file for slightly different shades:

body {

font-size: 14px;

color: navy;

}

h1 {

color: #ffc;

background-color: #009;

}

Save the CSS file and refresh your browser. You will see the colors of the first heading (the h1 element) have changed to yellow and blue.

You can apply the color and background-color properties to most HTML elements, including body, which will change the colors of the page and everything in it.

You can alter the size and shape of the text on a web page with a range of properties.

This is the font itself, such as Times New Roman, Arial, or Verdana.

The user’s browser has to be able to find the font you specify, which, in most cases, means it needs to be on their computer so there is little point in using obscure fonts that are only sitting on your computer. There are a select few “safe” fonts (the most commonly used are Arial, Verdana and Times New Roman), but you can specify more than one font, separated by commas. The purpose of this is that if the user does not have the first font you specify, the browser will go through the list until it finds one it does have. This is useful because different computers sometimes have different fonts installed. So font-family: arial, helvetica, serif, will look for the Arial font first and, if the browser can’t find it, it will search for Helvetica, and then a common serif font.

Note: if the name of a font is more than one word, it should be put in quotation marks, such as font-family: "Times New Roman".

You can use a wider selection than the “safe” fonts using several methods outlined in the CSS Advanced Tutorial but if you’re just getting to grips with CSS, we suggest sticking with this basic standard approach for the moment.

font-size sets the size of the font. Be careful with this — text such as headings should not just be an HTML paragraph ([p](https://www.htmldog.com/references/html/tags/p/)) in a large font - you should still use headings (h1, h2 etc.) even though, in practice, you could make the font-size of a paragraph larger than that of a heading (not recommended for sensible people).

font-weight states whether the text is bold or not. Most commonly this is used as font-weight: bold or font-weight: normal but other values are bolder, lighter, 100, 200, 300, 400 (same as normal), 500, 600, 700 (same as bold), 800 or 900.



**Fig. 4.1. Different text modifications**

Play around with these font-weight values if you want see their effect but, keep in mind, that some older browsers become a little confused with anything other than bold and normal so we suggest sticking to those unless you’re a typography ninja.

font-style states whether the text is italic or not. It can be font-style: italic or font-style: normal.

text-decoration states whether the text has got a line running under, over, or through it.

* text-decoration: underline, does what you would expect.
* text-decoration: overline places a line above the text.
* text-decoration: line-through puts a line through the text (“strike-through”).

This property is usually used to decorate links and you can specify no underline with text-decoration: none.

Underlines should only really be used for links. They are a commonly understood web convention that has lead users to generally expect underlined text to be a link.

text-transform will change the case of the text.

* text-transform: capitalize turns the first letter of every word into uppercase.
* text-transform: uppercase turns everything into uppercase.
* text-transform: lowercase turns everything into lowercase.
* text-transform: none I’ll leave for you to work out.

So, a few of these things used together might look like this:

body {

font-family: arial, helvetica, sans-serif;

font-size: 14px;

}

h1 {

font-size: 2em;

}

h2 {

font-size: 1.5em;

}

a {

text-decoration: none;

}

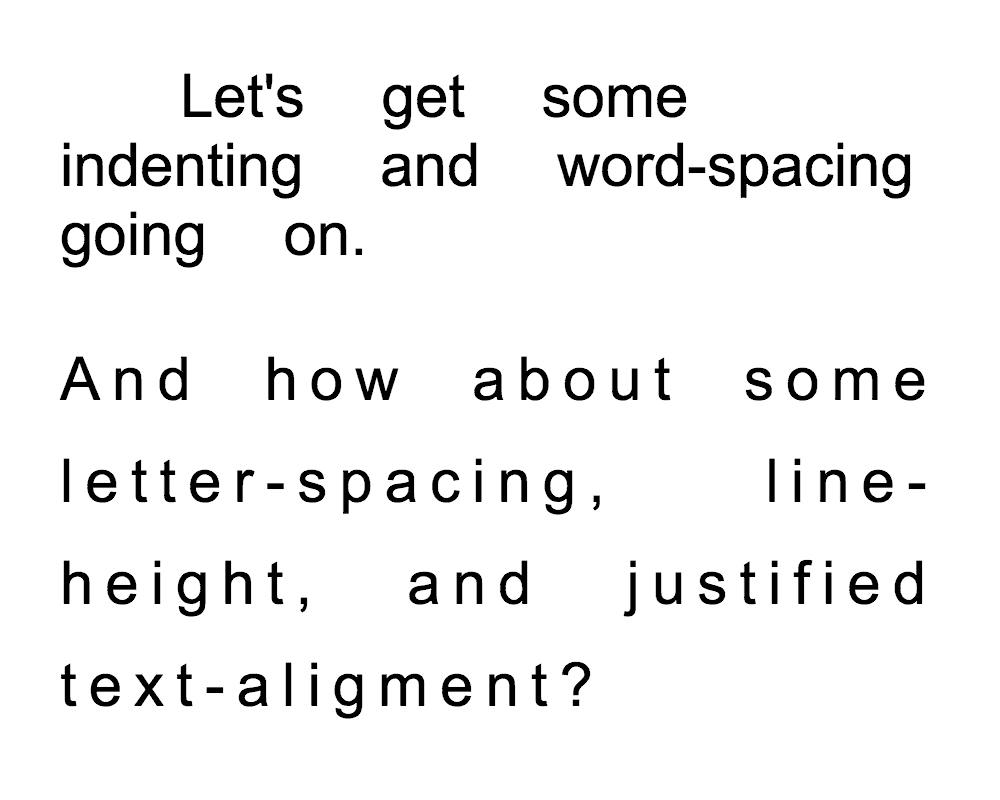
strong {

font-style: italic;

text-transform: uppercase;

}

Before we move on from this introduction to styling text, a quick look at how to space out the text on a page.



**Fig. 4.2. Spacing out text**

The letter-spacing and word-spacing properties are for spacing between letters or words. The value can be a length or normal.

The line-height property sets the height of the lines in an element, such as a paragraph, without adjusting the size of the font. It can be a number (which specifies a multiple of the font size, so “2” will be two times the font size, for example), a length, a percentage, or normal.

The text-align property will align the text inside an element to left, right, center, or justify.

The text-indent property will indent the first line of a paragraph, for example, to a given length or percentage. This is a style traditionally used in print, but rarely in digital media such as the web.

p {

letter-spacing: 0.5em;

word-spacing: 2em;

line-height: 1.5;

text-align: center;

}

margin and padding are the two most commonly used properties for spacing-out elements. A margin is the space outside something, whereas padding is the space inside something.

Change the CSS code for h2 to the following:

h2 {

font-size: 1.5em;

background-color: #ccc;

margin: 20px;

padding: 40px;

}

This leaves a 20-pixel width space around the secondary header and the header itself is fat from the 40-pixel width padding.

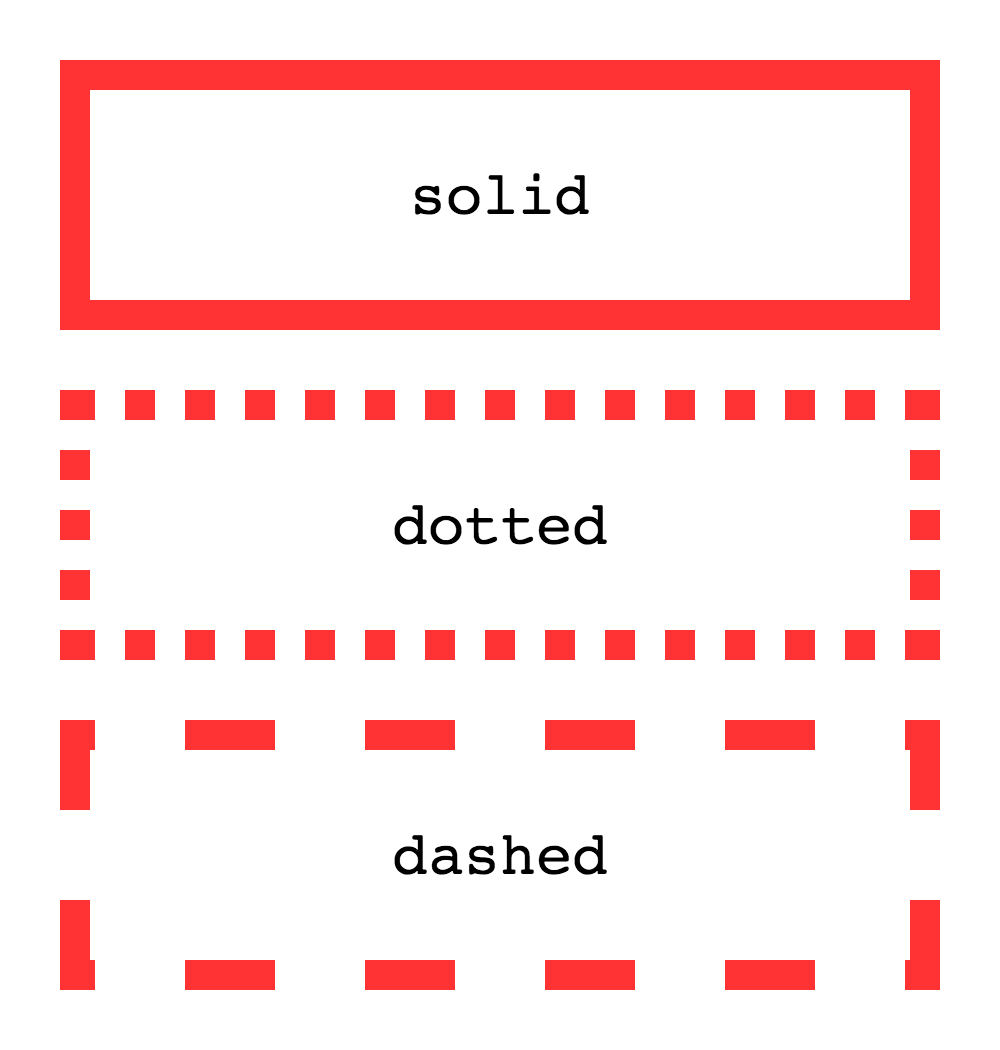
The four sides of an element can also be set individually. margin-top, [ma](https://www.htmldog.com/references/css/properties/margin-right/)rgin-right, margin-bottom, margin-left, padding-top, padding-right, padding-bottom and padding-left are the self-explanatory properties you can use.

Margins, padding and border are all part of what’s known as the Box Model. The Box Model works like this: in the middle you have the content area (let’s say an image), surrounding that you have the padding, surrounding that you have the border and surrounding that you have the margin. It can be visually represented like it’s shown on Fig. 4.3. You don’t have to use all of these, but it can be helpful to remember that the box model can be applied to every element on the page, and that’s a powerful thing!

To make a border around an element, all you need is border-style. The values can be solid, dotted, dashed, double, groove, ridge, inset and outset.



**Fig. 4.3. Box model**



**Fig. 4.4. Border styles**

border-width sets the width of the border, most commonly using pixels as a value. There are also properties for border-top-width, border-right-width, border-bottom-width and border-left-width.

Finally, border-color sets the color.

Add the following code to the CSS file:

h2 {

border-style: dashed;

border-width: 3px;

border-left-width: 10px;

border-right-width: 10px;

border-color: red;

}

This will make a red dashed border around all HTML secondary headers (the h2 element) that is 3 pixels wide on the top and bottom and 10 pixels wide on the left and right (these having over-ridden the 3 pixel wide width of the entire border).

The code below covers all of the CSS methods in this section. If you save this as your CSS file and look at the HTML file then you should now understand what each CSS property does and how to apply them. The best way to fully understand all of this is to mess around with the HTML and the CSS files and see what happens when you change things.

body {

font-family: arial, helvetica, sans-serif;

font-size: 14px;

color: black;

background-color: #ffc;

margin: 20px;

padding: 0;

}

/\* This is a comment, by the way \*/

p {

line-height: 21px;

}

h1 {

color: #ffc;

background-color: #900;

font-size: 2em;

margin: 0;

margin-bottom: 7px;

padding: 4px;

font-style: italic;

text-align: center;

letter-spacing: 0.5em;

border-bottom-style: solid;

border-bottom-width: 0.5em;

border-bottom-color: #c00;

}

h2 {

color: white;

background-color: #090;

font-size: 1.5em;

margin: 0;

padding: 2px;

padding-left: 14px;

}

h3 {

color: #999;

font-size: 1.25em;

}

img {

border-style: dashed;

border-width: 2px;

border-color: #ccc;

}

a {

text-decoration: none;

}

strong {

font-style: italic;

text-transform: uppercase;

}

li {

color: #900;

font-style: italic;

}

table {

background-color: #ccc;

}

**Questions**

1. Explain the ways to transform the text using CSS
2. What is the box model?
3. What attributes are applied to the box border?

## 

## TOPIC 5

DIV, SPAN, AND SEMANTIC STRUCTURE OF THE DOCUMENTS. META TAGS. ADVANCED TEXT FORMATTING

HTML is all about applying meaning to content. Whereas most HTML tags apply meaning (p makes a paragraph, h1 makes a heading etc.), the span and div tags apply no meaning at all. This might sound about as useful as a foam hammer but they are actually used quite extensively in conjunction with CSS.

They are used to group together a chunk of HTML and hook some information onto that chunk, most commonly with the attributes class and id to associate the element with a class or id CSS selector.

The difference between span and div is that a span element is in-line and usually used for a small chunk of HTML inside a line (such as inside a paragraph) whereas a div (division) element is block-line (which is basically equivalent to having a line-break before and after it) and used to group larger chunks of code.

<div id="scissors">

<p>This is <span class="paper">crazy</span></p>

</div>

div, and especially span, shouldn’t actually be used that often. Whenever there is a sensible alternative that should be used instead. For example, if you want to emphasize the word “crazy” and the class “paper” is adding italics for visual emphasis, then, for richer, more meaningful content, the code might look like this:

<div id="scissors">

<p>This is <em class="paper">crazy</em></p>

</div>

If you haven’t got a clue about classes and ID’s yet, don’t worry, they’re all explained in the next chapter. All you need to remember is that span and div are “meaningless” tags.

While they are not replacement for the div tag, HTML 5 introduces a number of tags used for grouping blocks of code together and adding meaning at the same time. For more information on article, header, footer, and more, see the chapter about sectioning.The basics of defining text using paragraphs (as well as emphasis and line breaks) and headings was covered in the HTML Beginner Tutorial. And for the same reason we use p and h1, h2, etc, there are a number of other tags we should also use to specifically represent other text-types, such as abbreviations, quotations, and computer code.

abbr is used to markup an abbreviation, a shortened form of a word or phrase.

The expanded phrase that the abbreviation represents can be defined in the value of the title attribute. This is optional but recommended.

<p>This web site is about <abbr title="HyperText Markup Language">HTML</abbr> and <abbr title="Cascading Style Sheets">CSS</abbr>.</p>

blockquote and q are used for quotations. blockquote is generally used for standalone often multi-line quotations whereas q is used for shorter, in-line quotations.

If the source of the quotation can be found on the Web, the cite attribute can be used to point to its origin.

<p>So I asked Bob about quotations on the Web and he said <q>I know as much about quotations as I do about pigeon fancying</q>. Luckily, I found HTML Dog and it said:</p>

<blockquote cite="http://www.htmldog.com/guides/html/intermediate/text/">

<p>blockquote and q are used for quotations. blockquote is generally used for standalone often multi-line quotations whereas q is used for shorter, in-line quotations.</p>

</blockquote>

Blockquotes work very nicely with the HTML5 elements figure and figcaption, enabling a nice, semantic way to group a quotation with a citation:

<figure>

<blockquote>[Big old quotation about evolution]</blockquote>

<figcaption>Charles Darwin</figcaption>

</figure>

But let’s not get carried away with that here - see the Sectioning for more.

Just to make things nice and confusing, as well as a cite attribute, there is also a cite tag. This can be used to define the title of a work, such as a book.

<p>According to <cite>the Bible</cite>, after six days God said <q>screw this for a lark, I'm having a nap</q>.</p>

code is used to represent any form of computer code. Further, var can be used for variables (which could be a variable in anything, not just in code - it could be a variable in an equation, for example), samp for sample output, and kbd (keyboard) for user input.

<p>If you add the line <code><var>givevaderachuckle</var> = true;</code> to the <code>destroy\_planet</code> subroutine and then type <kbd>ilovejabba</kbd> into the console, the big bad green Death Star laser will etch <samp>Slug Lover!</samp> on the planet's surface.</p>

pre is preformatted text and is unusual in HTML tags that it takes notice of every character in it, including the white space (whereas other elements will ignore a consecutive space or a line-break, for example). It is most commonly used for blocks of code, where spacing, such as indentations, can be relevant.

<pre><code>

&lt;div id="intro"&gt;

&lt;h1&gt;Some heading&lt;/h1&gt;

&lt;p&gt;Some paragraph paragraph thing thing thingy.&lt;/p&gt;

&lt;/div&gt;

</code></pre>

As an example, pre and code are used extensively throughout this site. The code blocks, such as the one above, are code elements inside pre elements. In-line references to tags and properties are simply code elements, often inside [a](https://www.htmldog.com/references/html/tags/a/) elements to link to the reference section.

Once upon a time, many eons ago, when the Internet was just a small number of cardboard boxes attached to each other with string, meta tags were the town criers of the Internet… erm… town.

Meta tags don’t do anything to the content that is presented in the browser window, but they are used by the likes of search engines to catalogue information about the web page.

There is one meta tag which can be used as many times as you desire inside a head element and they can contain the attributes charset, name, http-equiv, and content.

When the name or http-equiv attribute is used, the content attribute is then used in conjunction with them to apply meta data itself.

The name attribute can be anything you like. The most commonly used names are author, description, and keywords. author is used to explicitly state one of the HTML page’s authors and description is often used by search engines (such as Google) to display a description of a web page in its search results.

The reason why meta tags used to be so important was because they were relied on by search engines to build a profile of a web page. The keywords meta data was used (and abused) extensively, for example. Nowadays, however, most search engines use the actual content of the page itself.

The http-equiv attribute can be used instead of the name attribute and will make an HTTP header, which is information sent to the server where the web page is held. The attribute should rarely be used (although see charset note, below) but the value can be:

* content-type, an encoding declaration, defining what character set is being used,
* default-style, the preferred stylesheet from a selection of link or style elements,
* or refresh, which defines how often (in seconds) a page automatically refreshes or if it should automatically redirect to another page. Not great for accessibility.

The charset attribute can be used as a shorthand method to define an HTML document’s character set, which is always a good thing to do. <meta charset="utf-8"> is the same as <meta http-equiv="content-type" content="text/html; charset=utf-8">.

<head>

<title>Air conditioners? YEAH conditioners!</title>

<meta charset="utf-8">

<meta http-equiv="refresh" content="3"><!--not recommended for sane people-->

<meta name="description" content="This is my really, really, REALLY exciting web page about air conditioners">

…

Previously, we looked at unordered lists and ordered lists, but, much like Peter Cushing’s Doctor Who, description lists are quite often forgotten. This is maybe because they are much more specific than ordered and unordered lists and therefore less useful, generally, but where there is a list of terms and descriptions (such as a glossary), a description list is your go-to-element.

dl gets the ball rolling, similar to the ul and ol elements, establishing the list. Rather than containing li elements, though, description lists have dt elements, which are the terms, followed by dd elements, which are the descriptions associated to the dt elements.

There doesn’t have to be one dt followed by one dd, there can be any number of either. For example, if there are a number of words that have the same meaning, there might be a number of dt’s followed by one dd. If you have one word that means various different things, there might be one dt followed by several dd’s.

<h1>Some random glossary thing</h1>

<dl>

<dt>HTML</dt>

<dd>Abbreviation for HyperText Markup Language - a language used to make web pages.</dd>

<dt>Dog</dt>

<dd>Any carnivorous animal belonging to the family Canidae.</dd>

<dd>The domesticated sub-species of the family Canidae, Canis lupus familiaris.</dd>

<dt>Moo juice</dt>

<dt>Cat beer</dt>

<dt>Milk</dt>

<dd>A white liquid produced by cows and used for human consumption.</dd>

</dl>

OK, so we should all be know by now that there are lots of tags for text. We’ve done paragraphs, we’ve done headings, we’ve even done abbreviations, quotations, and code. And there are several other, more obscure tags that can be used. Obscure because you won’t find them plastered around the Web, not because they are unhelpful. If you find you have text that fits these elements’ descriptions, you will have a nicer, richer, more meaningful HTML page if you use them.

address should be used specifically for the contact details relating either to the entire web page (and so only used once) or to an article element (see Sectioning). It’s isn’t, as it might at first appear, for marking up any old address willy-nilly.

<h3>Author contact details</h3>

<address>

<ul>

<li>0123-456-7890</li>

<li>author\_dude@noplaceinteresting.com</li>

<li>http://www.noplaceinteresting.com/contactme/</li>

</ul>

</address>

dfn is a definition term and is used to highlight the first use of a term. Like abbr, the title attribute can be used to describe the term.

<p>Bob's <dfn title="Dog">canine</dfn> mother and <dfn title="Horse">equine</dfn> father sat him down and carefully explained that he was an <dfn title="A mutation that combines two or more sets of chromosomes from different species">allopolyploid</dfn> organism.</p>

bdo can be used to reverse the direction of the text, and can be used to display languages that read right to left. The value of the required attribute dir can be ltr (left to right) or rtl (right to left).

<bdo dir="rtl">god lmth</bdo>

ins and del are used to display editorial insertions and deletions respectively. Strictly speaking, they aren’t limited to text and can be used over whole swathes of content but, typically, they are used in moderation just like “Track Changes” feature in word processors tend to be.

They can have the attributes datetime to indicate when the edit was made and cite, to point to a description as to why the edit has been made.

<p>I have decided to <del datetime="2013-01-26">decrease</del> <ins cite="http://www.icecreamforall.com/changeofpolicy/">increase</ins> the amount of free ice cream that the State will provide for its citizens.</p>

As with traditional word processor-based editing, the ins element is typically shown underlined and the del element is usually displayed with a strikethrough.

And there’s yet more… HTML 5 brings with it even more text-related elements. See the final chapter on text, complete with a look at the reclassification of the dreaded “presentational” elements.

While headings do a grand, perfectly valid, job in defining the start of a new section or sub-section in a document, there are a number of elements that can be exploited to establish a cleaner, clearer semantic structure.

Whereas div elements can be used to contain sections, used primarily as scaffolding on which to hang CSS, they don’t hold a great deal of meaning. Sectioning involves a handful of tags that can be used to define specific parts of a page, such as articles, headers, footers, and navigation.

An article element can be used to mark up a standalone section of content. This could be used just once, if you think of a blog post as an article, for example, or a number of times, if you imagine replicating a traditional newspaper page with numerous articles.

A section element represents a more general section and could be used to split up an article, or to represent chapters, for example.

<article>

<section id="intro">

<p>[An introduction]</p>

</section>

<section id="main\_content">

<p>[Content]</p>

</section>

<section id="related">

<ul>

<li><a href="that.html">That related article</a></li>

<li><a href="this.html">This related article</a></li>

</ul>

</section>

</article>

While divs could be used to make these separations (or even if you didn’t need these separations for styling), this provides a much richer, more meaningful document.

The HTML5 specifications suggest that you can use h1 elements at the start of each section, which would become a sub-heading of anything preceding that section (so, in the example above, if you had an h1 immediately following the opening article tag, an h1 immediately after an opening section tag would be a sub-heading of that initial h1). This screws backwards compatibility, however, and any user agents (including screen readers) that don’t understand this won’t apply properly structured heading levels. We suggest sticking to the headings levels you would use if you didn’t use sections (so h1, followed by h2, etc, regardless of the sectioning). This doesn’t break anything or detract from the meaning or semantics.

header and footer provide further specialized, self-descriptive, sections:

<body>

<article>

<header>

<h1>Alternatively&hellip;</h1>

<p>[An introduction]</p>

</header>

<section id="main\_content">

<p>[Content]</p>

</section>

<footer>

<p>[End notes]</p>

</footer>

</article>

<footer>

<p>[Copyright bumf]</p>

</footer>

</body>

The footer is the footer of the section in which it is contained. So, in the above example, the first footer is the footer of the article and the second footer is the footer of the page.

An aside can be used to represent content that is related the content surrounding it. Think of pull-quotes or snippets of related information in an article:

<section id="main\_content">

<h1>Tixall</h1>

<p>[All about Tixall]</p>

<aside>

<h2>Tixall Obelisk</h2>

<p>[A short note about Tixall Obelisk]</p>

</aside>

<p>[Maybe a bit more about Tixall]</p>

</section>

While we’re at this structure-love, if you want to include figures, there happens to be two tags for doing just that:

<figure>

<img src="obelisk.jpg">

<figcaption>Tixall Obelisk</figcaption>

</figure>

Note that the img element doesn’t need an alt attribute IF the figcaption (that’s “figure caption”, in case you need it spelling out) does that job.

Finally, there’s nav, used to mark up a group of navigation links:

<nav id="main\_nav">

<ul>

<li><a href="/tutorials/">Tutorials</a></li>

<li><a href="/techniques/">Techniques</a></li>

<li><a href="/examples/">Examples</a></li>

<li><a href="/references/">References</a></li>

</ul>

</nav>

This could also be used for in-page navigation (<a href="#overthere">... etc.) but it isn’t necessary for every group of links - it is designed for major groupings. A copyright, author, and contact link could sit happily by themselves in a footer element, for example.

If you want to style these new HTML 5 elements (and you probably do, right? It’s much nicer than using <div id="content">..., etc) you will experience a problem in older browsers, most notably Internet Explorer versions 8 and earlier, that don’t understand these tags.

HTML5Shiv can come to your rescue, however; a small piece of JavaScript, slotted in to your head element, that teaches the remedial browsers and holds their hands so that you can use new HTML 5 tags and style them up to your heart’s content with CSS.

It’s your call if you want to use a JavaScript fudge or stick with the safe old (and, again, perfectly valid) but semantically poorer headings and divs approach. This site, along with many others, chooses the former. Because HTML 5’s loveliness just about outweighs a hack’s ugliness.

**Questions**

1. Explain the purpose of div and span tags.
2. Name the “meaningless” tags that are used for sectioning
3. What are the meta tags? Name some examples.

TOPIC 6

CLASS AND ID SELECTORS. GROUPING, PSEUDOCLASSES AND PAGE LAYOUT. PSEUDOELEMENTS

You can also define your own selectors in the form of class and ID selectors.

The benefit of this is that you can have the same HTML element, but present it differently depending on its class or ID.

In the CSS, a class selector is a name preceded by a full stop (“.”) and an ID selector is a name preceded by a hash character (“#”).

So the CSS might look something like:

#top {

background-color: #ccc;

padding: 20px

}

.intro {

color: red;

font-weight: bold;

}

The HTML refers to the CSS by using the attributes id and class. It could look something like this:

<div id="top">

<h1>Chocolate curry</h1>

<p class="intro">This is my recipe for making curry purely with chocolate</p>

<p class="intro">Mmm mm mmmmm</p>

</div>

The difference between an ID and a class is that an ID can be used to identify one element, whereas a class can be used to identify more than one.

You can also apply a selector to a specific HTML element by simply stating the HTML selector first, so p.jam { /\* whatever \*/ } will only be applied to paragraph elements that have the class “jam”.

Two ways that you can simplify your code — both HTML and CSS — and make it easier to manage.

You can give the same properties to a number of selectors without having to repeat them.

For example, if you have something like:

h2 {

color: red;

}

.thisOtherClass {

color: red;

}

.yetAnotherClass {

color: red;

}

You can simply separate selectors with commas in one line and apply the same properties to them all so, making the above:

h2, .thisOtherClass, .yetAnotherClass {

color: red;

}

If the CSS is structured well, there shouldn’t be a need to use many class or ID selectors. This is because you can specify properties to selectors within other selectors.

For example:

#top {

background-color: #ccc;

padding: 1em

}

#top h1 {

color: #ff0;

}

#top p {

color: red;

font-weight: bold;

}

This removes the need for classes or IDs on the [p](https://www.htmldog.com/references/html/tags/p/) and [h1](https://www.htmldog.com/references/html/tags/h1h2h3h4h5h6/) tags if it is applied to HTML that looks something like this:

<div id="top">

<h1>Chocolate curry</h1>

<p>This is my recipe for making curry purely with chocolate</p>

<p>Mmm mm mmmmm</p>

</div>

This is because, by separating selectors with spaces, we are saying “h1 inside ID top is colour #ff0” and “p inside ID top is red and bold”.

This can get quite complicated (because it can go for more than two levels, such as this inside this inside this inside this etc.) and may take a bit of practice.

Pseudo classes are bolted on to selectors to specify a state or relation to the selector. They take the form of selector:pseudo\_class { property: value; }, simply with a colon in between the selector and the pseudo class.

link, targeting unvisited links, and visited, targeting, you guessed it, visited links, are the most basic pseudo classes.

The following will apply colors to all links in a page depending on whether the user has visited that page before or not:

a:link {

color: blue;

}

a:visited {

color: purple;

}

Also commonly used for links, the dynamic pseudo classes can be used to apply styles when something happens to something.



**Fig. 6.1. Changing the styles of a box when the cursor moves over it.**

* active is for when something activated by the user, such as when a link is clicked on.
* hover is for a when something is passed over by an input from the user, such as when a cursor moves over a link.
* focus is for when something gains focus, that is when it is selected by, or is ready for, keyboard input.

focus is most often used on form elements but can be used for links. Although most users will navigate around and between pages using a pointing device such as a mouse those who choose note to, or are unable to do so, such as those with motor disabilities, may navigate using a keyboard or similar device. Links can be jumped between using a tab key and they will gain focus one at a time.

a:active {

color: red;

}

a:hover {

text-decoration: none;

color: blue;

background-color: yellow;

}

input:focus, textarea:focus {

background: #eee;

}

Finally (for this article, at least), there is the first-child pseudo class. This will target something only if it is the very first descendant of an element. So, in the following HTML…

<body>

<p>I’m the body’s first paragraph child. I rule. Obey me.</p>

<p>I resent you.</p>

...

…if you only want to style the first paragraph, you could use the following CSS:

p:first-child {

font-weight: bold;

font-size: 40px;

}

CSS3 has also delivered a whole new set of pseudo classes: last-child, target, first-of-type, and more.

margin allows you to amalgamate margin-top, margin-right, margin-bottom, and margin-left in the form of property: top right bottom left;

So:

p {

margin-top: 1px;

margin-right: 5px;

margin-bottom: 10px;

margin-left: 20px;

}

Can be summed up as:

p {

margin: 1px 5px 10px 20px;

}

padding can be used in exactly the same way.

By stating just two values (such as padding: 1em 10em;), the first value will be the top and bottom and the second value will be the right and left.

border-width can be used in the same way as margin and padding, too. You can also combine border-width, border-color, and border-style with the border property. So:

p {

border-width: 1px;

border-color: red;

border-style: solid;

}

Can be simplified to be:

p {

border: 1px red solid;

}

The width/color/style combination can also be applied to border-top, border-right etc.

Font-related properties can also be gathered together with the font property:

p {

font: italic bold 12px/2 courier;

}

This combines font-style, font-weight, font-size, line-height, and font-family.

So, to put it all together, try this code:

p {

font: 14px/1.5 "Times New Roman", times, serif;

padding: 30px 10px;

border: 1px black solid;

border-width: 1px 5px 5px 1px;

border-color: red green blue yellow;

margin: 10px 50px;

}

Lovely.

Used in a very different way to the img HTML element, CSS background images are a powerful way to add detailed presentation to a page.

To jump in at the deep end, the shorthand property [background](https://www.htmldog.com/references/css/properties/background/) can deal with all of the basic background image manipulation aspects.

body {

background: white url(http://www.htmldog.com/images/bg.gif) no-repeat top right;

}

This amalgamates these properties:

* background-color, which we have come across before.
* background-image, which is the location of the image itself.
* background-repeat, which is how the image repeats itself. Its value can be:
  + repeat, the equivalent of a “tile” effect across the whole background,
  + repeat-y, repeating on the y-axis, above and below,
  + repeat-x (repeating on the x-axis, side-by-side), or
  + no-repeat (which shows just one instance of the image).
* background-position, which can be top, center, bottom, left, right, a length, or a percentage, or any sensible combination, such as top right.



**Fig. 6.2. A top-right positioned, non-repeating background.**

It can actually be used to specify a few other background features too, notably attachment, clip, origin, and size (see the background property reference for the nitty-gritty), but let’s not get carried away just yet — color, image, repeat, and position are by far the most fundamental aspects that you would want to manipulate most often.

Background-images can be used in most HTML elements - not just for the whole page (body) and can be used for simple but effective results. As an example, background images are used on this web site as the bullets in lists, as the magnifying glass in the search box, and as the icons in the top left corner of some notes, such as this one.

If you have two (or more) conflicting CSS rules that point to the same element, there are some basic rules that a browser follows to determine which one is most specific and therefore wins out.

It may not seem like something that important, and in most cases you won’t come across any conflicts at all, but the larger and more complex your CSS files become, or the more CSS files you start to juggle with, the greater likelihood there is of conflicts arising.

If the selectors are the same then the last one will always take precedence. For example, if you had:

p { color: red }

p { color: blue }

The text in the box of p elements would be colored blue because that rule came last.

However, you won’t usually have identical selectors with conflicting declarations on purpose (because there’s not much point). Conflicts quite legitimately come up, though, when you have nested selectors.

div p { color: red }

p { color: blue }

In this example it might seem that a p within a div would be colored blue, seeing as a rule to color p boxes blue comes last, but they would actually be colored red due to the specificity of the first selector. Basically, the more specific a selector, the more preference it will be given when it comes to conflicting styles.

The actual specificity of a group of nested selectors takes some calculating. You can give every ID selector (“#whatever”) a value of 100, every class selector (“.whatever”) a value of 10 and every HTML selector (“whatever”) a value of 1. When you add them all up, hey presto, you have a specificity value.

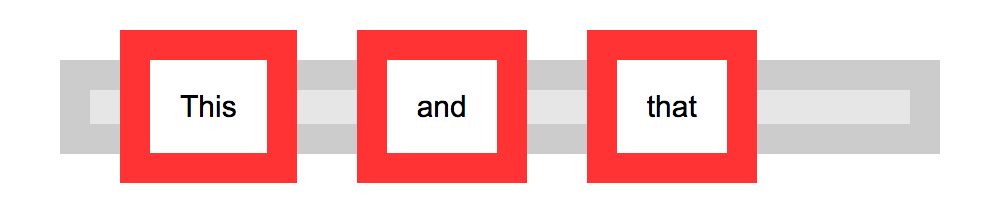
* p has a specificity of 1 (1 HTML selector)
* div p has a specificity of 2 (2 HTML selectors, 1+1)
* .tree has a specificity of 10 (1 class selector)
* div p.tree has a specificity of 12 (2 HTML selectors + a class selector, 1+1+10)
* #baobab has a specificity of 100 (1 id selector)
* body #content .alternative p has a specificity of 112 (HTML selector + id selector + class selector + HTML selector, 1+100+10+1)

So if all of these examples were used, div p.tree (with a specificity of 12) would win out over div p (with a specificity of 2) and body #content .alternative p would win out over all of them, regardless of the order.

A key trick to the manipulation of HTML elements is understanding that there’s nothing at all special about how most of them work. Most pages could be made up from just a few tags that can be styled any which way you choose. The browser’s default visual representation of most HTML elements consist of varying font styles, margins, padding and, essentially, display types.

The most fundamental types of display are inline, block and none and they can be manipulated with the display property and the shockingly surprising values inline, block and none.

inline does just what it says — boxes that are displayed inline follow the flow of a line. Anchor (links) and emphasis are examples of elements that are displayed inline by default.



**Fig. 6.3. Inline boxes.**

The following code, for example, will cause all list items in a list to appear next to each other in one continuous line rather than each one having its own line:

li { display: inline }

block makes a box standalone, fitting the entire width of its containing box, with an effective line break before and after it. Unlike inline boxes, block boxes allow greater manipulation of height, margins, and padding. Heading and paragraph elements are examples of elements that are displayed this way by default in browsers.

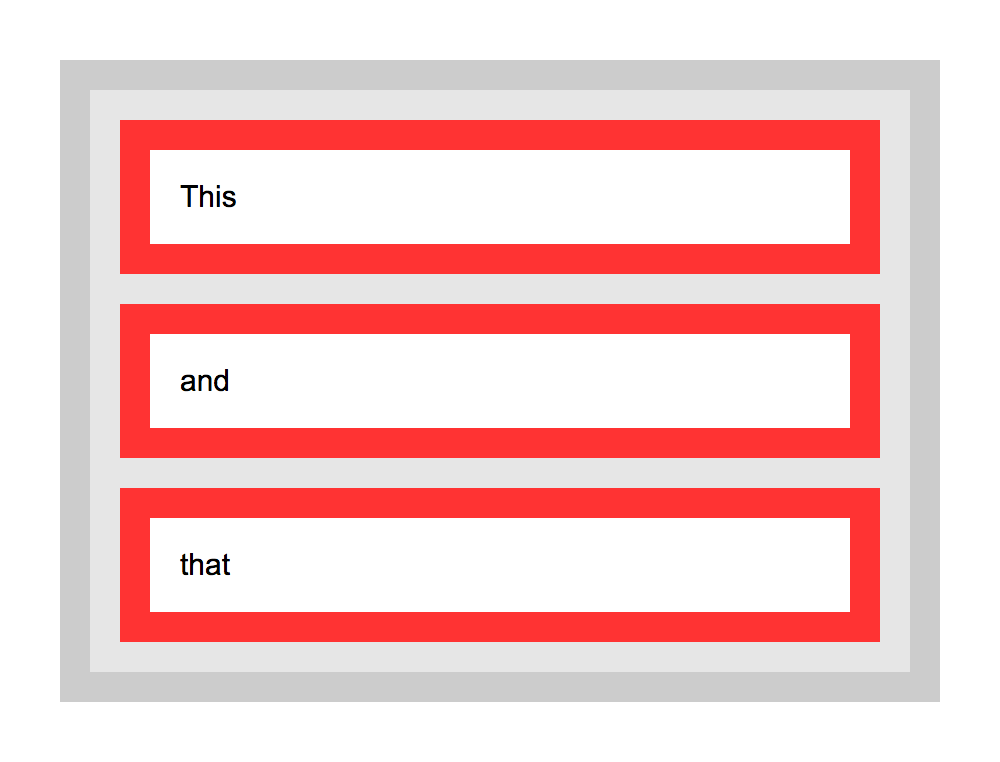


Fig. 6.4. Block boxes.

The next example will make all links in “nav” large clickable blocks:

#navigation a {

display: block;

padding: 20px 10px;

}

display: inline-block will keep a box inline but lend the greater formatting flexibility of block boxes, allowing margin to the right and left of the box, for example.

none, well, doesn’t display a box at all, which may sound pretty useless but can be used to good effect with dynamic effects, such as switching extended information on and off at the click of a link, or in alternative stylesheets.

The following code, for example, could be used in a print stylesheet to basically “turn off” the display of elements such as navigation that would be useless in that situation:

#navigation, #related\_links { display: none }

display: none and visibility: hidden vary in that display: none takes the element’s box completely out of play, whereas visibility: hidden keeps the box and its flow in place without visually representing its contents. For example, if the second paragraph of 3 were set to display: none, the first paragraph would run straight into the third whereas if it were set to visibility: hidden, there would be a gap where the paragraph should be.

OK. So that was the basics. Now for something a little more advanced and rarely used…

Perhaps the best way to understand the table-related display property values is to think of HTML tables. table is the initial display and you can mimic the tr and td elements with the table-row and table-cell property values respectively.

The display property goes further by offering table-column, table-row-group, table-column-group, table-header-group, table-footer-group and table-caption as values, which are all quite self-descriptive. The immediately obvious benefit of these values is that you can construct a table by columns, rather than the row-biased method used in HTML.

Finally, the value inline-table basically sets the table without line breaks before and after it.

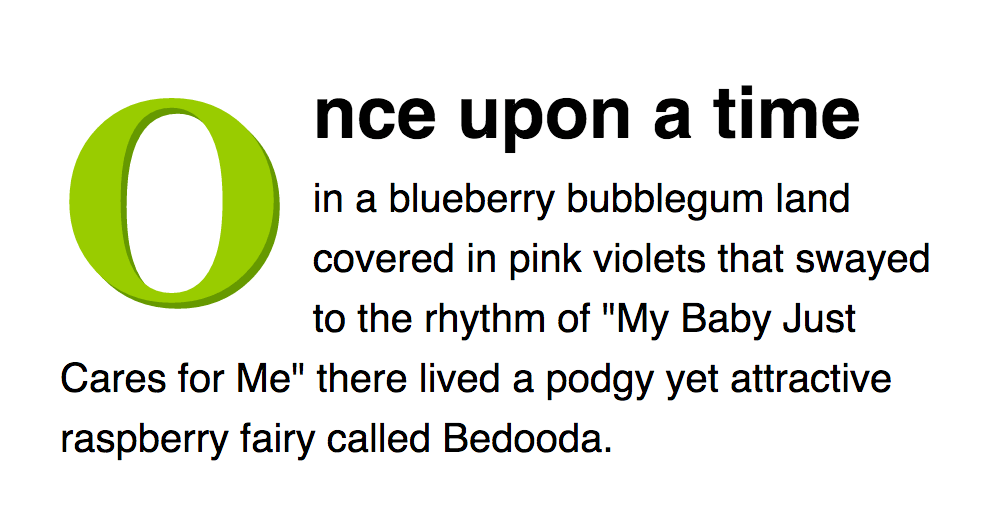
Be careful when using these values. Older browsers struggle with them and getting carried away with CSS tables can seriously damage your accessibility. HTML should be used to convey meaning, so if you have tabular data it should be arranged in HTML tables. Using CSS tables exclusively could result in a mash of data that is completely unreadable without the CSS. Bad. And not in a Michael Jackson way.

list-item displays a box in the way that you would usually expect an [li](https://www.htmldog.com/references/html/tags/li/) HTML element to be displayed. To work properly then, elements displayed this way should be nested in a [ul](https://www.htmldog.com/references/html/tags/ul/) or [ol](https://www.htmldog.com/references/html/tags/ol/) element.

run-in makes a box either in-line or block depending on the display of its parent.

Pseudo elements suck on to selectors much like pseudo classes, taking the form of selector:pseudoelement { property: value; }. There are four of the suckers.

The first-letter pseudo element applies to the first letter inside a box and first-line to the top-most displayed line in a box.



**Fig. 6.5. Targeting the first letter and first line of a paragraph with pseudo elements.**

As an example, you could create drop caps and a bold first-line for paragraphs with something like this:

p {

font-size: 12px;

}

p:first-letter {

font-size: 24px;

float: left;

}

p:first-line {

font-weight: bold;

}

The CSS 3 specs suggest pseudo elements should include two colons, so p::first-line as opposed to p:first-line. This is to differentiate them with pseudo classes. Aiming for backwards-compatibility (whereby older web pages will still work in new browsers), browsers will still behave if they come across the single colon approach and this remains the best approach in most circumstances due to some older browsers not recognizing the double colon.

The before and after pseudo elements are used in conjunction with the content property to place content either side of a box without touching the HTML.

What?! Content in my CSS?! But I thought HTML was for content!

Well, it is. So use sparingly. Look at it like this: You are borrowing content to use solely as presentation, such as using “!” because it looks pretty. Not because you actually want to exclaim anything.

The value of the content property can be open-quote, close-quote, any string enclosed in quotation marks, or any image, using url(imagename).

blockquote:before {

content: open-quote;

}

blockquote:after {

content: close-quote;

}

li:before {

content: "POW! ";

}

p:before {

content: url(images/jam.jpg);

}

The content property effectively creates another box to play with so you can also add styles to the “presentational content”:

li:before {

content: "POW! ";

background: red;

color: #fc0;

}

In the olden days, pre-hominid apes used HTML tables to layout web pages. Hilarious, right?! But CSS, that 2001: A Space Odyssey monolith, soon came along and changed all of that.

Layout with CSS is easy. You just take a chunk of your page and shove it wherever you choose. You can place these chunks absolutely or relative to another chunk.

The position property is used to define whether a box is absolute, relative, static or fixed:

* static is the default value and renders a box in the normal order of things, as they appear in the HTML.
* relative is much like static but the box can be offset from its original position with the properties top, right, bottom and left.
* absolute pulls a box out of the normal flow of the HTML and delivers it to a world all of its own. In this crazy little world, the absolute box can be placed anywhere on the page using top, right, bottom and left.
* fixed behaves like absolute, but it will absolutely position a box in reference to the browser window as opposed to the web page, so fixed boxes should stay exactly where they are on the screen even when the page is scrolled.

When we talk of absolutely positioned boxes being placed anywhere on the page, they’re actually still relatively positioned from the edges of the page. And to add another backtrack, the page doesn’t actually have to be the container — a box will be “absolutely” positioned in relation to any non-static positioned containing box. Just ignore that for now, though…

You can create a traditional two-column layout with absolute positioning if you have something like the following HTML:

<div id="navigation">

<ul>

<li><a href="this.html">This</a></li>

<li><a href="that.html">That</a></li>

<li><a href="theOther.html">The Other</a></li>

</ul>

</div>

<div id="content">

<h1>Ra ra banjo banjo</h1>

<p>Welcome to the Ra ra banjo banjo page. Ra ra banjo banjo. Ra ra banjo banjo. Ra ra banjo banjo.</p>

<p>(Ra ra banjo banjo)</p>

</div>

We’re being old-skool and using div elements so as not to introduce too many new things, but Sectioning is better.

And if you apply the following CSS:

#navigation {

position: absolute;

top: 0;

left: 0;

width: 200px;

}

#content {

margin-left: 200px;

}

You will see that this will set the navigation bar to the left and set the width to 200 pixels. Because the navigation is absolutely positioned, it has nothing to do with the flow of the rest of the page so all that is needed is to set the left margin of the content area to be equal to the width of the navigation bar.

How stupidly easy! And you aren’t limited to this two-column approach. With clever positioning, you can arrange as many blocks as you like. If you wanted to add a third column, for example, you could add a “navigation2” chunk to the HTML and change the CSS to:

#navigation {

position: absolute;

top: 0;

left: 0;

width: 200px;

}

#navigation2 {

position: absolute;

top: 0;

right: 0;

width: 200px;

}

#content {

margin: 0 200px; /\* setting top and bottom margin to 0 and right and left margin to 200px \*/

}

The only downside to absolutely positioned boxes is that because they live in a world of their own, there is no way of accurately determining where they end. If you were to use the examples above and all of your pages had small navigation bars and large content areas, you would be okay, but, especially when using relative values for widths and sizes, you often have to abandon any hope of placing anything, such as a footer, below these boxes. If you wanted to do such a thing, one way would be to float your chunks, rather than absolutely positioning them.

Floating a box will shift it to the right or left of a line, with surrounding content flowing around it.

Floating is normally used to shift around smaller chunks within a page, such as pushing a navigation link to the right of a container, but it can also be used with bigger chunks, such as navigation columns.

Using float, you can float: left or float: right.

Working with the same HTML, you could apply the following CSS:

#navigation {

float: left;

width: 200px;

}

#navigation2 {

float: right;

width: 200px;

}

#content {

margin: 0 200px;

}

Then, if you do not want the next box to wrap around the floating objects, you can apply the clear property:

* clear: left will clear left floated boxes
* clear: right will clear right floated boxes
* clear: both will clear both left and right floated boxes.

So if, for example, you wanted a footer in your page, you could add a chunk of HTML…

<div id="footer">

<p>Footer! Hoorah!</p>

</div>

…and then add the following CSS:

#footer {

clear: both;

}

And there you have it. A footer that will appear underneath all columns, regardless of the length of any of them.

This has been a general introduction to positioning and floating, with emphasis on the larger “chunks” of a page, but remember, these methods can be applied to any box within those boxes, too. With a combination of positioning, floating, margins, padding and borders, you should be able to represent any web design your mischievous little imagination can conjure. The best way to learn? Tinker! Play! Go!

**Questions**

1. What CSS selectors are used for ids and classes?
2. Explain the usage of background images.
3. Explain the work of different “display” CSS options.

## 

## LECTURE 7

HTML5 POSSIBILITIES AND EMBEDDING THE CONTENT

HTML5 adds and amends a handful of tags relating to text. Many of the minor amendments, such as differing attributes in existing tags, have already been covered, but this page looks at two new tags — time and mark — as well as the re-definition of presentational tags.

time is by far the chocolate ice cream sexiest sweet sugar lovely of these tags and is used to make dates and times super-semantically rich and mmm.

The text sandwiched in the middle of the opening and closing tag can be any format of date of time - the whole precise lot, or just one part, such as a day. It is made more helpful, however, by the datetime attribute, the value of which should be a machine-readable date and/or time.

<p>Written by Doctor Who on <time datetime="2052-11-21">Thursday 21st November 2052</time>.</p>

Valid datetime values can take the format of a year-month-day date (as above), of as a “fuzzy” date, such as “2052-11”, of a time, such as “09:30” (always using a 24-hour clock) or a combination, such as “2052-11-21 09:30”. This can also accommodate time zones and durations.

If the textual content of the time element is already machine readable, you don’t need the datetime attribute but it is required if it isn’t.

Text can be highlighted, as if with a marker pen, using mark:

<blockquote>

<p>He wants to play with his <mark>Legos</mark></p>

</blockquote>

<p>The person being quoted is clearly American because, for some odd reason, they pluralise "Lego".</p>

Yes, this is a form of emphasis, literally speaking, but it won’t always be considered emphasis in the original meaning (for example, the person being quoted above isn’t emphasizing “Legos”, the commenter is), hence its inclusion.

One of the slightly more The single most revolting “advance” in HTML 5 is its attempt to redefine the archaic presentational tags. Once popular, many moons ago, they just won’t go away no matter how over the hill they are. These tags are also known as “Cliff Richard tags”.

Some of the newly defined blighters are helpful, some are questionable, and some have new definitions crowbarred in that are, to say the least, tenuous.

* hr, no longer “horizontal rule”, is a thematic break, between paragraphs, for example, like those found in many a chapter of many a book.
* small, used for small print. Arguably a fair point, “small print” has taken on a meaning beyond “print that is small”.
* s, no longer “strikethrough”, is for text that is no longer correct (eg, this is <s>presentational, not</s> meaningful). Hmm. OK. Maybe. del still seems fine to most normals, though.
* u, no longer “underline”, is for text that is unarticulated. It’s also “useless” but bonus point for the abbreviation remaining intact.
* i, no longer “italic”, is for text in an alternate voice or representing a different quality of text. So, like, differently emphasized, then (see note below).
* b, no longer “bold”, stands for “text to which attention is being drawn without conveying importance or suggesting an alternative voice” (and even that’s paraphrasing). b also stands for “bollocks.”
* sub and sup are still subscript and superscript and yet, at the same time, they’re somehow not presentational anymore.

The specs contradict themselves at times: “The sup element represents a superscript,” it says. Superscript is, by definition, not to mention its actual inherent semantic meaning, presentational. And yet the specification goes on to say that it “must be used only to mark up typographical conventions with specific meanings, not for typographical presentation”.

The fundamental problem with many of these new definitions, in fact, is that they say, in essence “this is emphasis, but it isn’t emphasis.”

As an example, perhaps the best argument for an [i](https://www.htmldog.com/references/html/tags/i/) element is using it to represent taxonomic genus and species names which, traditionally, are represented in italics. But if something is in italics when text surrounding it isn’t in italics then it’s emphasized visually. If it’s (literally) in an alternate voice from preceding and proceeding content, it’s emphasized aurally.

And, on a practical level, given it has taken so long for software that interprets HTML, such as screen readers, to take any notice of basic web standards, what are the chances of them taking any notice of tags for vague notions of “alternative voices” and “drawing attention without conveying importance?” (And is there a tag for rhetorical questions?)

Adding more specific meaning is welcome but even when helpful, these tags still aren’t ideal - they’re ugly. Messy. While we’re loving semantics, we’re supposed to be happy with the likes of hr when “H.R.” is a misnomer? Tempered happiness, maybe.

In case you haven’t picked up on the subtleties, we recommend you avoid these tags whenever possible. They serve to pollute and confuse more than clarify and em does the job perfectly more often than not.

Some well-respected professionals (well-respected by HTML Dog as well) are happy to use these tags. You can too, if you really want. Go ahead - swallow the W3Cs drunken folly - not many fairies will die because of it. But here at the kennels we think that keeping a puritanical mindset of separating meaningful content from presentation reaps benefits in itself and that these tags should have been put down long ago.

So, you’re looking for a filthy hack that allows you to target HTML solely at versions of Microsoft’s Internet Explorer browser, are you? How convenient.

Older versions of Internet Explorer are frequently either inept or naughty. Or both. But they are still popular so we don’t want to ignore them.

Conditional comments, which are nothing more than simple HTML comments that IE (up to version 9) happens to take a peep at, are used to throw a chunk of HTML at these browsers and only these browsers. Other well behaved, top-of-the-class browsers will simply see them as unremarkable comments and move along.

They have become a commonly used method of latching extra CSS to a document to plaster over holes in these browsers’ display capabilities. So, for example, you might add something like this inside your head element:

<link href="everything.css" rel="stylesheet">

<!--[if IE]><link href="stupidie.css" rel="stylesheet"><![endif]-->

Everything between <!--[if IE]> and <![endif]--> will be picked up by Internet Explorer. So this will bolt on a CSS file as normal, and then, only if the browser is Internet Explorer (in practice, this will be Internet Explorer version 9 and below), it will also apply an extra CSS file patch.

You can also target specific versions of Internet Explorer:

* <!--[if IE 6>…
* <!--[if IE 7>…
* <!--[if IE 8>…
* <!--[if IE 9>…

You can also target all versions that are greater or less than a certain number:

* eg. <!--[if IE gt 6]>… for IE versions greater than 6
* eg. <!--[if IE gte 8]>… for IE versions greater than or equal to than 8
* eg. <!--[if IE lt 7]>… for IE versions less than 7
* eg. <!--[if IE lte 7]>… for IE versions less than or equal to 7

There are actually more options than this which are largely totally unnecessary. Take a look at Microsoft’s own take on it if you really feel compelled to find out more.

Conditional comments really are horrible. Necessary, often, at the moment, but horrible. Like all hacks, it is best to avoid them wherever possible. They’re not really there for whacking completely different stylesheets in different browsers, for example. It’s more for small fallbacks so that you can use the scrumptious likes of CSS 3 without compromise. And don’t assume you have to accommodate every stone-age version of IE, either - try and figure out what is sensible for you to support. Are your web site visitors likely to be using IE6? Probably not.

There are a number of ways links — the absolutely fundamentally most important interactive element of web sites — can be made more accessible to those people with disabilities.

Users who do not or cannot use pointing devices can tab through links and, as such, links should be in a logical tabbing order. The tabindex attribute allows you to define this order although if the HTML is linear, as it should be, a logical tabbing order should automatically fall into place.

<ul>

<li><a href="here.html" tabindex="1">Here</a></li>

<li><a href="there.html" tabindex="3">There</a></li>

<li><a href="limbo.html" tabindex="2">Limbo</a></li>

</ul>

In this example (which is used purely as a demonstration - it would be quite dumb, practically speaking), tabbing would jump from “Here” to “Limbo” to “There”.

The good bit of link-accessibility advice is to write good link text. Have the words the [a](https://www.htmldog.com/references/html/tags/a/) tags wrap around explain where the link will take the user. If someone is using a screen reader, having the links read out to them as they tab between them, the user will then know what they’re letting themselves in for if they select a link. “Click here” or random words aren’t especially helpful…Link titles

If you have a link that isn’t self-descriptive, or the link destination could benefit from being explained in more detail, you can add information to a link using the title attribute.

<p>I'm really bad at writing link text. <a href="inept.html" title="Why I'm rubbish at writing link text: An explanation and an apology.">Click here</a> to find out more.</p>

Another tip: Don’t have links open something in a new window or tab. It’s precious to think your page is important enough to stay alive while a user visits elsewhere anyway. We all know how to use the back button. We know how to close windows and tabs, too, but if you can’t actually see that that’s what has happened…

If you insist on doing this, at least mention it in a link title.

Access keys allow easier navigation by assigning a keyboard shortcut to a link (which will usually gain focus when the user presses “Alt” or “Ctrl” + the access key).

<a href="somepage.html" accesskey="s">Some page</a>

For users who do not use pointing devices, this can be a quick way to navigate. The trouble is that there the user may not know what they are unless they are explicitly stated although some assistive software will tell the user what these are.

To aid tabbing, you can supply links that allow users to jump over chunks of your web page. You might want to allow someone to jump over a plethora of navigation links, for example, so they can just read a page’s main content rather than cycle through all of the links:

<header>

<h1>The Heading</h1>

<a href="#content">Skip to content</a>

</header>

<nav>

<!--loads of navigation stuff -->

</nav>

<section id="content">

<!--lovely content -->

</section>

You probably won’t want this link to be displayed visually - it’s a peculiar link to see for abled-bodied users and screen reader users won’t need to see it (it will be read out). So you can use CSS to render it invisible but it could also be beneficial to those with motor disabilities so you might also want to consider making it visible when it has focus from being tabbed to using the :focus CSS pseudo class.

Forms aren’t the easiest of things to use for people with disabilities. Navigating around a page with written content is one thing, hopping between form fields and inputting information is another. Because of this, it is a good idea to add a number of elements to the form.

Each form field should have its own explicit label. The label tag sorts this out, with a for attribute that associates it to a form element:

<form>

<label for="yourName">Your Name</label>

<input name="yourName" id="yourName">

<!-- etc. -->

Labels have the added bonus of visual browsers rendering the labels themselves clickable, putting the focus on the associated form field.

Both name and id attributs are typically required; the name for the form to handle that field and the id for the label to associate it to.

You can group fields, for example name (first, last, middle, title etc.) or address (line 1, line 2, county, country, postal code, country etc.) using the fieldset tag.

Within the field set, you can set a caption with the legend tag.

<form action="somescript.php" >

<fieldset>

<legend>Name</legend>

<p>First name <input name="firstName"></p>

<p>Last name <input name="lastName"></p>

</fieldset>

<fieldset>

<legend>Address</legend>

<p>Address <textarea name="address"></textarea></p>

<p>Postal code <input name="postcode"></p>

</fieldset>

<!-- etc. -->

Most browsers tend to represent field sets with a border surrounding them and the legend caption breaking the left of the top border by default. You can, of course, change this with CSS if you wish.

The optgroup element groups options in a select box. It requires a label attribute, the value of which is displayed as a non-selectable pseudo-heading preceding that group in the drop-down list of visual browsers.

<select name="country">

<optgroup label="Africa">

<option value="gam">Gambia</option>

<option value="mad">Madagascar</option>

<option value="nam">Namibia</option>

</optgroup>

<optgroup label="Europe">

<option value="fra">France</option>

<option value="rus">Russia</option>

<option value="uk">UK</option>

</optgroup>

<optgroup label="North America">

<option value="can">Canada</option>

<option value="mex">Mexico</option>

<option value="usa">USA</option>

</optgroup>

</select>

## 

Like links, form fields (and field sets) need to be navigated to without the use of a pointing device, such as a mouse. The same methods used in links to make this task easier can be used on form elements — tab stops and access keys.

The accesskey and tabindex attributes can be added to the individual form tags such as input and also to legend tags.

<input name="firstName" accesskey="f" tabindex="1">

For more about this, see the Accessible Links page.

HTML5 greatly advances form controls, with numerous additional input types, several new attributes, and a handful of extra elements.

Getting this warning in early, in case you quite understandably decide that it would be a waste of time reading the rest of this page, a vast majority of this new gubbins will not yet work fully on all major browsers. Unsurprisingly, Internet Explorer is the main dunce, with next to none of this working in anything lower than IE 10 and even that version fails to support some input types. All is not fruitless, though — see the note about usage suggestions at the end of HTML5 Forms Pt. 2.

Basic form fields created using the input element include text, password, checkbox, radio, and submit, which have already been covered in the previous lectures. These types have been extended in HTML5 to accommodate more specific fields:

Used for a search query text box, this performs exactly as a standard text input should.

<input type="search" name="search">

The main intention of the inclusion of this input type in the HTML5 specification is one of style. As well as making your HTML clearer, you can also target this element with a CSS attribute selector:

input[type=search] { background: url(magnifyingglass.png) right no-repeat) }

Other “special” text input types include tel, for telephone numbers, url, for web addresses, and email, for email addresses.

<input type="tel" name="telephone\_number">

<input type="url" name="web\_address">

<input type="email" name="email\_address">

You can use the :valid and :invalid CSS3 pseudo classes to style these fields depending on whether their content is considered valid.

input[type=email]:valid { background: green }

input[type=email]:invalid { background: red }

This example will paint an email field’s background green if the entered text is recognized as an email address (such as “sausage@htmldog.com”) or red if it isn’t (if the user were to type “sausages?”, for example).

A simple text box that also allows a user to directly type in a number, or cycle through numbers (usually using an up and down arrow to the side of the field), can be achieved with type="number". A further step attribute can be added to specify how much is added or subtracted from the number with each increment.

If you also want the number to have a minimum or maximum value, you can further use the min and max attributes.

<input type="number" name="quantity" step="2" min="20" max="30">

Once again, if this is supported, the user will be able either to type directly into the field or, if using the arrows, cycle between 20 and 30, two units at a time.

You can use the :valid and :invalid pseudo classes in relation to this, too. If the user were to type “12”, for example, that would be invalid, because it isn’t between 20 and 30. If they typed “23” that would also be invalid because it isn’t a multiple of 2.

An alternative to the digits-in-a-text-box approach can be achieved using type="range". By default, this should be displayed as a horizontal bar with a slider in the middle of it. The user can then adjust the slider left and right, the far left resulting in a value of “0” and the far right a value of “100”. This range can be adjusted using the min and max attributes.

<input type="range" name="temperature" min="15" max="25" step="0.5" value="18.5">

There are several input types for dates and times:

* type="datetime"
* type="date"
* type="month"
* type="week"
* type="time"
* type="datetime-local"

If supported (they aren’t, widely, and they are also inconsistent between browsers), these will prompt the user to enter a date or time in a specific format, either by directly typing it in, cycling through one week/day/hour/minute/etc. at a time, or by selecting from a dropdown calendar.

step, min, and max attributes can be used with dates and times, too, as can the CSS pseudo classes to style according to validity.

Finally, type="color" is designed to allow a user to select a color, sending a six-digit hex code as its value.

<input name="color" type="color" value="#ff8800">

Continuing from HTML5 Forms Pt. 1, in addition to the multitude of fresh new input types, there are also additional form-specific attributes at your disposal as well as data lists, a sort of text/select hybrid.

As well as those attributes mentioned, both here and in earlier guides, there is a handful of additional attributes:

The placeholder attribute can be used with text input fields (and their text-like cousins, such as type="email" and type="number") as well as textarea elements. It is intended as a hint, rather than a label, for which a label element should still be used.

<label for="email\_address">Email address</label>

<input type="email" placeholder="you@somewhere.com" name="email\_address" id="email\_address">

You might want focus to land on a form field when a page loads. If you think of a search engine, for example, when you land on its home page you don’t normally need to click on the search box to start typing - you can do so straight away because it already has focus. The autofocus attribute is a quick way to achieve this effect.

<input name="query" autofocus>

A data list takes the form of a list of suggestions that accompanies a text field:

<input name="country" list="country\_name">

<datalist id="country\_name">

<option value="Afghanistan">

<option value="Albania">

<option value="Algeria">

<option value="Andorra">

<option value="Armenia">

<option value="Australia">

<option value="Austria">

<option value="Azerbaijan">

<!-- etc. -->

</datalist>

The value of the list attribute in the input element (which could be any of the text-like input types) binds it to a datalist element with the corresponding ID (“country\_name”, in this example). As a user types in the text field, if what they type matches the start of anything in the data list, those matches will be shown underneath the text field as suggestions. So, here, if “A” is typed, the 8 countries beginning with “a” are displayed. If “L” is typed after “A”, the list of suggestions will reduce to just “Albania” and “Algeria”, and so on. The data sent when the form is submitted will be whatever is in the text field - it could be something selected from the list or it could be something completely different, inputted by the user.

The good news is that many of the features outlined in these two HTML 5 Forms pages degrade gracefully. Those browsers that don’t support data lists still maintain the text box; unrecognised input types revert to text inputs, so you can use the likes of search, tel, and url as long as you aren’t relying on their special features; placeholder text simply won’t appear so as long as it isn’t essential, go for it.

HTML5 introduces a swathe of new tags to accommodate the increasingly interactive and multimedia nature of the Web. While there have been numerous ways to embed video, audio, and dynamic imagery in the past, the new web standard attempts to make this easier, more consistent, and more reliable.

The simplest embedded (foreign) content is an image, applied to a web page with the img element. In the olden days, object, along with various plugins and proprietary devil dust, was used to bash and smash video and audio into submission. Although not without its (compatibility) problems, there is now a much better method for using various types of media in web pages.

<video src="kitties.mp4" controls></video>

Bam. There you go. Just like that. Simple.

This will embed a video, complete with controls, in browsers that support the HTML5 video tag and the video content type.

While HTML5 is pushing for a standard framework to play media, the media itself is not standardised across browsers. In practice, this means that it is unlikely all visitors will be able to experience your video (or audio) file. Some support WebM video, for example, while others support MPEG. Don’t lose too much sleep over this, though — see “Alternative content”, below.

The controls attribute is optional but if you don’t want it - if you really want to take control away from the user - you can just slap in an autoplay attribute:

<video src="kitties.mp4" autoplay></video>

This will play the video on page load, won’t display any controls, and will most likely annoy the hell out of your visitors. Of course you could, if you were kind, put in both the controls and autoplay attributes.

Other basic attributes at your disposal include width, height, loop, and muted.

<video src="kitties.mp4" width="300" height="200" loop muted autoplay controls></video>

If you insist on using autoplay, bringing muted (and controls) to the party as well is a nice gesture and is a convention that many sites employ. If you have a video in an aside, for example, it can begin playing but the user can then opt to follow it by de-muting the video via the controls, if they choose, decreasing the likelihood of irritation.

You can specify a placeholder image, which will be displayed before the video is played, with the poster attribute.

<video src="kitties.mp4" poster="fluffy.jpg" controls></video>

The specified image will stretch or shrink to fit the dimensions of the video, regardless of the original size of the image.

So, yes, there is an opening and closing tag. Whatever could go in between them? Why, fall-back content: content that is displayed if the browser doesn’t understand the video element. That could be a few words, a chunk of HTML, or a “really funny” and “highly original” Lolcats image.

<video src="kitties.mp4" controls>

<img src="hahahaha.jpg" alt="Hilarious cat and caption saying 'soz'.">

</video>

As already noted, it’s not only compatibility with the tag we need to worry about, but also compatibility with the source video itself. Luckily, more than one video source file can be offered up with the source element along with indications of the requirements of the file in the value of the type attribute. The browser will then take the first one it’s happy with.

<video controls>

<source src="kitties.mp4" type="video/mp4; codecs='avc1, mp4a'">

<source src="kitties.webm" type="video/webm; codecs='vp8.0, vorbis'">

<p>Browser no likey HTML 5.</p>

</video>

Here, a browser should figure out if it can handle the “video/mp4” MIME type and if it has the stated codec to decipher it. If it doesn’t, it should move on to the next and try again with the details set out in the second source element.

Applying audio is just like applying video. Using the audio tag, the structure is the same as using video and the attributes src, controls, autoplay and loop can all be used in the same way.

<audio src="meow\_mix.mp3" controls>

Your stupid browser doesn't support HTML 5 audio.

</audio>

Alternative content can also be defined in exactly the same way as with the video and source tags.

Much greater control can be exercised over video and audio using JavaScript, with the ability to manipulate aspects of playback and user controls in more detail.

A major addition to HTML5 is the canvas element. It is designed to provide a canvas onto which JavaScript can be used to paint all manner of dynamic images such as graphs, animated sprites, or daft cat pictures.

<canvas id="wittykitty" width="800" height="450">

<!-- Fall-back content here, just like with video and audio -->

</canvas>

That’s it. That’s the extent of the actual HTML, at least — the power is in the scripting.

**Questions**

1. How the media files are embedded into HTML?
2. What input types appeared in HTML5 standard?
3. Explain the concept of canvas.

## 

## LECTURE 8

ADVANCED CSS. MEDIA QUERIES

Rounded corners used to be the stuff of constricting solid background images or, for flexible boxes, numerous background images, one per-corner, slapped on multiple nested div elements. Argh, ugly. Well, not any longer. Now, with simple CSS, you can lavish your designs with more curves than Marilyn Monroe.

Yeah. Border radius. Fear not, though — you don’t have to have borders. The border-radius property can be used to add a corner to each corner of a box.

#marilyn {

background: #fff;

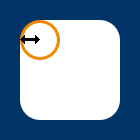
width: 100px;

height: 100px;

border-radius: 20px;

}

And there you have it. Rounded corners, see? Well, you will if you’ve got a sensible browser (see note below).



**Fig. 8.1. Corners are cut around the corresponding quarters of a circle (or ellipse) with the specified radius.**

Of course, if you do want a border…

#ok\_a\_border\_then {

border: 5px solid #8b2;

width: 100px;

height: 100px;

border-radius: 5px;

}

Woop.

border-top-left-radius, border-top-right-radius, border-bottom-right-radius, and border-bottom-left-radius can also be used to target specific corners.

Ever so slightly less horribly wordy, you can also define all corner radii (what a great word) individually with a space-separated list of values, working clockwise from top-left, just like other shorthand properties:

#monroe {

background: #fff;

width: 100px;

height: 100px;

border-radius: 6px 12px 18px 24px;

}



**Fig. 8.2. Multiple-value border-radius.**

Curvy.

By using two values instead of four you target top-left and bottom-right with the first length (or percentage) and top-right + bottom-left with the second. Three values? Top-left, then top-right + bottom-left, then bottom-right. Smashing.

Gah! There just had to be browser issues, didn’t there? Damn you, browsers.

Internet Explorer versions 8 and below don’t support border-radius. The only way you can deal with this is either to make do with a design in those browsers that doesn’t have rounded corners (most people can live with that), or revert to the old background images.

You might also stumble across similar proprietary properties, such as -webkit-border-radius and -moz-border-radius which are for older, barely-used versions of Safari and Firefox respectively. Our carefully worded professional advice? Screw ‘em.

Are circles a bit too square for you? You can specify different horizontal and vertical radiiii by splitting values with a “/”.

#nanoo {

background: #fff;

width: 100px;

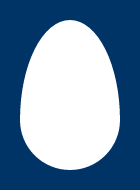
height: 150px;

border-radius: 50px/100px;

border-bottom-left-radius: 50px;

border-bottom-right-radius: 50px;

}



**Fig. 8.3. Big values of border radius**

Look! It’s like someone’s shining a torch over my web page!

You can give parts of your page “pop” by applying shadows both to boxes and to text.

box-shadow is the standard CSS property to get you going and it can have a value comprising several parts:

box-shadow: 5px 5px 3px 1px #999

* The first value is the horizontal offset — how far the shadow is nudged to the right (or left if it’s negative)
* The second value is the vertical offset — how far the shadow is nudged downwards (or upwards if it’s negative)
* The third value is the blur radius — the higher the value the less sharp the shadow. (“0” being absolutely sharp). This is optional — omitting it is equivalent of setting “0”.
* The fourth value is the spread distance — the higher the value, the larger the shadow (“0” being the inherited size of the box). This is also optional — omitting it is equivalent of setting “0”.
* The fifth value is a color. That’s optional, too.

You can also apply shadows to the inside of a box by adding “inset” to the list:

box-shadow: inset 0 0 7px 5px #ddd;

Splendid!

You might come across browser-specific versions of box-shadow, such as -moz-box-shadow and -webkit-box-shadow. Ignore ‘em. They’re old and stupid. The majority of modern browsers understand box-shadow, including Internet Explorer versions 9 and above.

box-shadow, as its name implies, only has eyes for boxes. Fickle beast. But you can also apply shadows to the outline of text with (surprise!) text-shadow:

text-shadow: -2px 2px 2px #999;

Similarly to box-shadow:

The first value is the horizontal offset

The second value is the vertical offset

The third value is the blur radius (optional)

The fourth value is the color (optional, although omitting this will make the shadow the same color as the text itself)

Note that there is no spread distance or inset option for text-shadow.

text-shadow has taken a little bit longer for browsers to figure out. Internet Explorer 9 and below won’t understand it so we suggest only using it in non-critical situations.

In earlier tutorials, we have covered HTML selectors, Class and ID selectors, and how to combine selectors to target specific element boxes. With the use of three itty-bitty characters, you can further pinpoint an element, reducing the need to bloat your HTML with class and ID attributes.

Using an asterisk (“ \* ”), you can target everything under the sun. You can use it by itself to set global styles for a page, or as a descendant of a selector to set styles of everything within something.

The following, for example, will set the margin and padding on everything in a page to zero and everything within an element with the ID “contact” to be displayed as a block:

\* {

margin: 0;

padding: 0;

}

#contact \* {

display: block;

}

Using a standalone universal selector is commonly used to “reset” many of a browser’s default styles. Setting a margin to zero, for example, will kill all spacing around the likes of paragraphs, headings and blockquotes.

A greater-than symbol (“>”) can be used to specify something that is a child of something else, that is, something immediately nested within something.

So, with this HTML…

<ul id="genus\_examples">

<li>Cats

<ul>

<li>Panthera</li>

<li>Felis</li>

<li>Neofelis</li>

</ul>

</li>

<li>Apes

<ul>

<li>Pongo</li>

<li>Pan</li>

<li>Homo</li>

</ul>

</li>

</ul>

…and the following CSS…

#genus\_examples > li { border: 1px solid red }

…a red border would be drawn around “Cats” and “Apes” only, rather than around every single list item (which would be the case with #genus\_examples li { border: 1px solid red }). This is because the likes of “Panthera” and “Felis” are grandchildren of “genus\_examples”, not children.

A plus sign (“+”) is used to target an adjacent sibling of an element, essentially, something immediately following something.

With the following HTML:

<h1>Clouded leopards</h1>

<p>Clouded leopards are cats that belong to the genus Neofelis.</p>

<p>There are two extant species: Neofelis nebulosa and Neofelis diardi.</p>

…and CSS…

h1 + p { font-weight: bold }

Only the first paragraph, that following the heading, will be made bold.

A further, CSS 3, “general sibling” selector uses a tilde (“~”) and will match an element following another regardless of its immediacy. So, in the above example, h1 ~ p { font-weight: bold } will style all paragraphs after the top-level heading but if there were any ps preceding the h1, these would not be affected.

We already know that colors can be defined by name, RGB, or hex values, but CSS 3 also allows you to paint away with HSL — hue, saturation, and lightness — as well as stipulating transparency.

There are no super special properties at play here — HSL and RGBa (the “a” standing for “alpha”, as in “alpha transparency”) can be applied to any property that has a color value, such as color, background-color, border-color or box-shadow, to name a mere handful.

RGBa opens up an exciting new dimension to web design, allowing you to set the transparency of a box or text. If you wanted a smidgen of a snazzy background image to peep through a heading, for example, you might use something like this:

h1 {

padding: 50px;

background-image: url(snazzy.jpg);

color: rgba(0,0,0,0.8);

}

A standard value of rgb(0,0,0) would set the heading to pure black but that fourth value, in rgba, sets the level of transparency, “1” being completely opaque, “0” being completely transparent. So rgba(0,0,0,0.8) is saying red=“0”, green=“0”, blue=“0”, alpha=“0.8”, which, all together, makes it 80% black.

This doesn’t only apply to text, of course, you could apply a transparent background color to an entire box, a transparent box shadow… anywhere where you can use rgb, you can used rgba.

Color names aside, web colors have always been red-green-blue biased, be that through hex codes or explicit RBG (or RGBa). Although mildly less straightforward (especially if your brain is trained to break down colors into red, green and blue), HSL can actually be more intuitive because it gives you direct control over the aspects of a color’s shade rather than its logical ingredients.

It is used in a similar way to rgb:

#smut { color: hsl(36, 100%, 50%) }

Rather than each sub-value being a part of the color spectrum, however, they are:

* Hue (“36” in the above example): Any angle, from 0 to 360, taken from a typical color wheel, where “0” (and “360”) is red, “120” is green and “240” is blue.
* Saturation (“100%” in the example): How saturated you want the color to be, from 0% (none, so a level of grey depending on the lightness) to 100% (the whole whack, please).
* Lightness (“50%” in the example): From 0% (black) to 100% (white), 50% being “normal”.

So the example used here will produce an orange (36°) that is rich (100% saturation) and vibrant (50% lightness). It is the equivalent of #ff9900, #f90, and rgb(255, 153, 0).

Hey, man, this funky fresh transparency and HSL can be combined?! You’d better believe it. Here’s HSLa:

#rabbit { background: hsla(0, 75%, 75%, 0.5) }

You can figure out what that does, right?

hsl and hsla are supported by most modern browsers, including IE versions 9 and above.

At-rules are clever, powerful little huggers that encapsulate a bunch of CSS rules and apply them to something specific. They can be used to import other CSS files, apply CSS to a particular media, or embed funkysexy uncommon fonts.

Each at-rule starts with an apetail, or an “at sign”, if you want to be boring about it (“@”).

Let’s start with the simple @import rule. This is used to bolt another stylesheet onto your existing one.

@import url(morestyles.css);

This can be used if a site requires long, complex stylesheets that might be easier to manage if they are broken down into smaller files.

@import rules must be placed at the top of a stylesheet, before any other rules.

@media can be used to apply styles to a specific media, such as print.

@media print {

body {

font-size: 10pt;

font-family: times, serif;

}

#navigation {

display: none;

}

}

Values that follow “@media” can include screen, print, projection, handheld, and all, or a comma-separated list of more than one, such as:

@media screen, projection {

/\* ... \*/

}

It doesn’t stop there, oh no. CSS 3 allows you to target not only specific media but also variables relating to that media, such as screen size (particularly helpful in targeting phones). Have a gander at the Media Queries page for more.

@font-face has been around for a long time but was nigh-on useless for much of its life. CSS 3 has polished it up and it is now widely used as a technique for embedding fonts in a web page so that a typeface can be used even if it isn’t sitting on the user’s computer. So you no longer need to rely on “web safe” fonts such as Arial or Verdana. Exciting times.

Jumping in at the deep end…

@font-face {

font-family: "font of all knowledge";

src: url(fontofallknowledge.woff);

}

What this does is create a font named “font of all knowledge” using the font-family descriptor and links the font file “fontofallknowledge.woff” to that name using the src descriptor. “font of all knowledge” can then be used in a standard font rule, such as:

p { font-family: "font of all knowledge", arial, sans-serif; }

The font will be downloaded (in this case from the same directory as the CSS file) and applied to paragraphs. If the browser is too decrepit to deal with sparkly new font-faces, it will simply revert to the next, standard, font in the list. Magic!

You can also look for a number of fonts to apply to the rule with a comma-separated list. Checking to see if a font is already present on a user’s computer, removing the need to download it, can also be accomplished by replacing “url” in the src value with “local”.

And because a font file might contain a whole host of weights or styles, you might also want to specify which one you’re interested in. So the @font-face rule could end up looking something like this:

@font-face {

font-family: "font of all knowledge";

src: local("font of all knowledge"), local(fontofallknowledge), url(fontofallknowledge.woff);

font-weight: 400;

font-style: normal;

}

Legally speaking, you can’t just throw any old font you feel like up on the Interweb because there are copyright and usage terms to consider, not to mention compatibility and optimization issues.

There are free web fonts out there that you can find, download, upload, and use, though. Hell, you could even create one yourself if you’re mad-scientist clever. There are also web font providers, such as Adobe’s Typekit, that have a large selection of fonts to choose from at a price.

Google Web Fonts has a more limited selection but it’s free to use and makes things super, super (super, super, super) easy. All you need to do is link to one of their external CSS files, which is nothing more than a @font-face rule, and whammo — you’ve got a darling new font to play with.

What? More selectors? Damn straight. Because someone somewhere really wants you to keep your HTML as tidy as possible. So before you add a class attribute to a tag see if you can simply target it by an attribute that might already be there.

Attribute selectors allow you to style an element’s box based on the presence of an HTML attribute or of an attribute’s value.

Appending an attribute name enclosed in square brackets, to style something that simply contains a certain attribute, you can do something like this:

abbr[title] { border-bottom: 1px dotted #ccc }

This basically says “shove a dotted line underneath all abbreviations with a title attribute”.

You can further specify that you want to style something with a specific attribute value.

input[type=text] { width: 200px; }

This example will apply a width of 200 pixels only to input elements that are specified as being text boxes (<input type="text">).

This approach can certainly be useful in forms where input elements can take on many guises using the type attribute.

You can also target more than one attribute at a time in a way similar to the following:

input[type=text][disabled] { border: 1px solid #ccc }

This will only apply a gray border to text inputs that are disabled.

CSS 3 further allows you to match an attribute without being exact:

* [attribute^=something] will match a the value of an attribute that begins with something.
* [attribute$=something] will match a the value of an attribute that ends with something.
* [attribute\*=something] will match a the value of an attribute that contains something, be it in the beginning, middle, or end.

So, as an example, you could style external links (eg. “http://www.htmldog.com”) differently to internal links (eg. “overhere.html”) in the following way:

a[href^=http] {

padding-right: 10px;

background: url(arrow.png) right no-repeat;

}

Transitions allow you to easily animate parts of your design without the need for the likes of JavaScript. How dangerous.

At the most simplistic level, think of a traditional :hover state, where you might change the appearance of a link when a cursor fondles it:

a:link {

color: hsl(36,50%,50%);

}

a:hover {

color: hsl(36,100%,50%);

}

This creates a binary animation; a link switches from a subdued orange to a rich orange when it is hovered over.

The transition property, however, is much more powerful, allowing smooth animation (rather than a jump from one state to another). It is a shorthand property that combines the following properties (which can be used individually if you so choose):

* transition-property: which property (or properties) will transition.
* transition-duration: how long the transition takes.
* transition-timing-function: if the transition takes place at a constant speed or if it accelerates and decelerates.
* transition-delay: how long to wait until the transition takes place.

Returning to the shorthand property, if a transition is applied like so…

a:link {

transition: all .5s linear 0;

color: hsl(36,50%,50%);

}

a:hover {

color: hsl(36,100%,50%);

}

…when the link is hovered over, the color will gradually change rather than suddenly switch. The transition property here is saying it wants all properties transitioned over half a second in a linear fashion with no delay.

For a transition to take place, only transition-duration is required, the rest defaulting to transition-property: all; transition-timing-function: ease; transition-delay: 0;. So you could, for example, simply declare transition: .5s.

While “all” can be used in transition-property (or transition), you can tell a browser only to transition certain properties, by simply plonking in the property name you want to change. So the previous example could actually include transition: color .5s ease 0, given only the color changes.

If you want to target more than one property (without using “all”), you can offer up a comma-separated list with transition-property:

a:link {

transition: .5s;

transition-property: color, font-size;

...

Or you can offer up a slew of rules for transitioning each property like so:

a:link {

transition: color .5s, font-size 2s;

…

OK, so transition-timing-function (catchy!) is the least obvious fella. It effectively states how the browser should deal with the intermediate states of the transition.

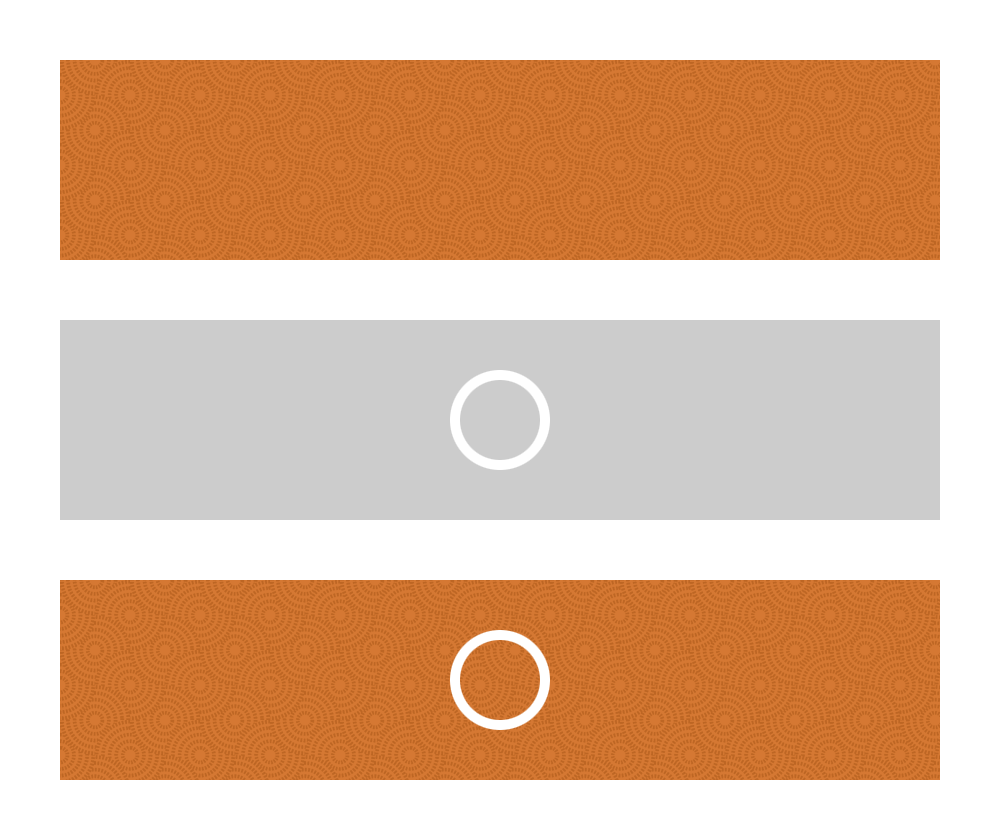
It follows a cubic Bézier curve. Yeah. Obviously, we know all about them from infant school, but, to get down to it, at the most basic level you can use ease or linear.

ease produces a gradual acceleration at the start of the transition and a gradual deceleration at the end. linear maintains a constant speed throughout. Other values include ease-in and ease-out.

CSS transitions won’t work in Internet Explorer versions 9 and below. Booo. But that won’t matter in cases (which will be most cases) where transitions aren’t essential for a design to work well. It’s just a little something to keep in mind.

As well as plastering a single or tiling background image across parts of your page, you can also apply multiple backgrounds, adjust the size of background images, and define the origin of a background based on levels of the box model.

This is the boogie web designers have been crying out for since Bing Crosbie was topping the charts.



**Fig. 8.4. A repeating background image, a single-instance background image, and combining them together in the same box.**

CSS3 allows you to apply multiple background images to a single box by simply putting image locations in a comma-separated list:

background-image: url(this.jpg), url(that.gif), url(theother.png);

More usefully, you can also define all of the other delightful background aspects. If you wanted to style a chunky button-like link with separate background, bullet, and arrow, for example, you could use something like:

background: url(bg.png), url(bullet.png) 0 50% no-repeat, url(arrow.png) right no-repeat;

Easy, right? It’s just the same as using background-image, background-position, background-repeat, background-attachment, and background, except you can specify more than one background with the aid of that helpful little comma.

The background-size property allows you to stretch or compress a background image.



**Fig. 8.5. background-size: contain and background-size: cover**

The values can be:

* auto, which maintains the background image’s original size and width/height ratio.
* lengths, a width and a height, such as 100px 50px (100px wide, 50px high). Specifying a single length, such as 100px will result the equivalent of 100px auto.
* percentages, a width and a height, such as 50% 25% (50% of the width of the background area, 25% of the height of the background area). Specifying a single percentage, such as 50% will result the equivalent of 50% auto.
* A combination of lengths, percentages, and auto, such as 80px auto (80px wide, automatic height to maintain the image’s original ratio)
* contain, which maintains the background image’s original ratio and makes it as large as possible whilst fitting entirely within the box’s background area.
* cover, which maintains the background image’s original ratio and makes it large enough to fill the entire background area, which may result in cropping of either the height or width.

background-origin is the remarkably dull kid of the bunch. Not unintelligent, just dull. The kid that the other kids point and laugh at. The property defines where the background area of a box actually starts. If you think of the box model, when you set a background it should, by default, start from the upper-left corner of the padding box. So if you had…

#burrito {

width: 400px;

height: 200px;

border: 10px solid rgba(0,255,0,.5);

padding: 20px;

background: url(chilli.png) 0 0 no-repeat;

}

…the background image should appear in the top left corner, in the padding box, immediately after the inner edges of a green border. You can change this behavior, however, with background-origin. Its self-descriptive values can be padding-box (default, as described above), border-box, and content-box.

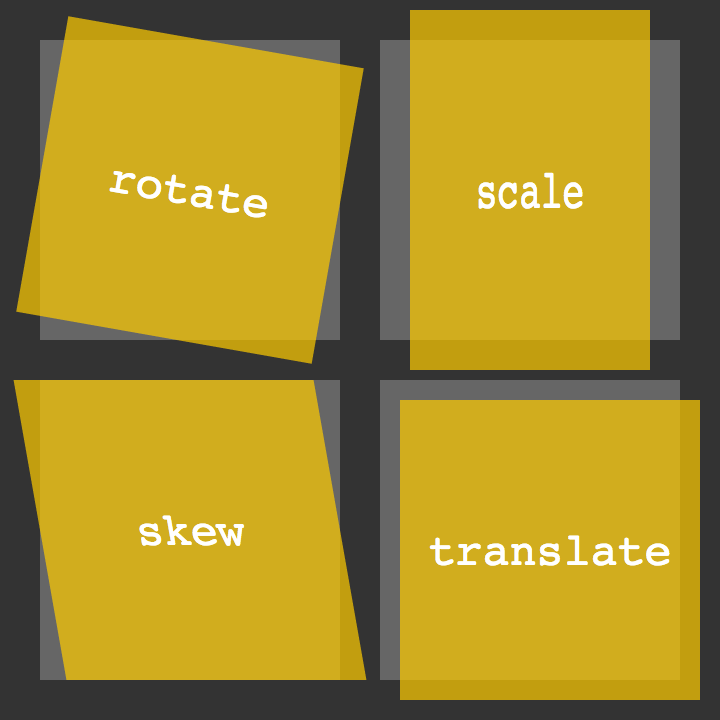
So adding background-origin: border-box to the previous example will cause the origin of the background image to be tucked up inside the border.

A woeful mega-budget Michael Bay movie about CSS?! Nay, the powerful manipulation of the shape, size, and position of a box and its contents using the transform property.

By default, CSS boxes, those visual representations of HTML elements, are so square. Rectangular, at least; level, with four straight sides and four boring right angles. But with the use of transform you can stretch and mold those boxes into all manner of shapes.

This page will only mention the transform and transform-origin properties but, in practice, you will probably want to duplicate these with -webkit-transform and -webkit-transform-origin to achieve the same results in the likes of Safari and Chrome as well as -ms-transform and -ms-transform-origin for Internet Explorer 9, which is the earliest version of IE that will support transforms.

Manipulating a box over two dimensions, transform can be used to rotate, skew, scale and translate a box and its contents.



**Fig. 8.6. The four basic 2D transform functions — rotate, skew, scale, and translate.**

The following would result in a square orange box with all of its contents — text, images, whatever — obediently standing to attention:

.note {

width: 300px;

height: 300px;

background: hsl(36,100%,50%);

}

But you can spin those soldiers around by adding a further declaration:

transform: rotate(-10deg);

This will turn the box and, importantly, everything in it, ten degrees anti-clockwise.

Skewing allows you to tip the horizontal and vertical so the following…

transform: skew(20deg,10deg);

…will tip over the x-axis by 20 degrees on the y-axis by 10 degrees.

You can also specify one angle, such as skew(20deg), which is the equivalent of skew(20deg,0).

Obviously, you can change width and height properties on a box, but that won’t affect the size of anything inside it. Scaling, however, will multiply not only the width and height, but the size of everything contained in the box as well:

transform: scale(2);

This will multiply the size by two. You can use any positive number, including a value less than “1”, such as “0.5”, if you want to use a shrink ray.

You can also scale the horizontal and vertical dimensions separately:

transform: scale(1,2);

This will leave the horizontal as is (because it’s a scale of 1) and multiply the vertical by two.

You can shift a box horizontally and vertically using transform: translate:

transform: translate(100px,200px);

Similar to position: relative; left: 100px; top: 200px;, this will move a box 100 pixels to the right and 200 pixels down.

As well as the values mentioned, if you want to target an individual axis, you can also use skewX, skewY, scaleX, scaleY, translateX, and translateY.

Want to rotate and scale at the same time? You crazy kid. You can do this by simply separating values with spaces, such as:

transform: rotate(-10deg) scale(2);

The order of the values is important - the latter will be executed before the former. In the previous example, the box will be scaled and then rotated. It is, therefore, different to transform: scale(2) rotate(-10deg);, which will be rotated and then scaled.

Alternatively, you could use the matrix function. matrix does the whole lot - rotating, skewing, scaling, and translating. It takes six values:

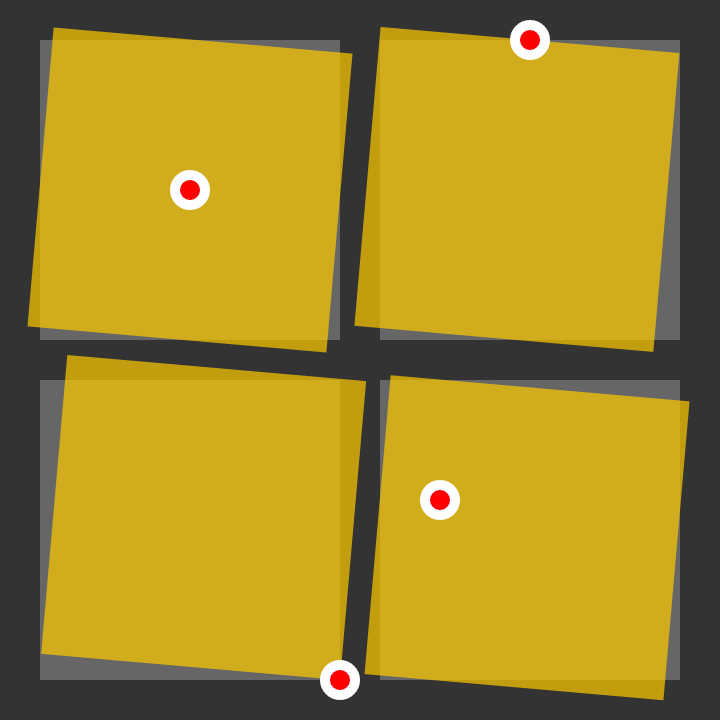
transform: matrix(2,-0.35,0.35,2,0,0);

These values aren’t entirely straightforward and involve maths (or just one math if you’re of the American persuasion) that, if you really want to tackle (there are benefits not only in brevity but also in precision), it may be worth giving the specs a gander.

There’s one important aspect missing. If you are transforming a box, there is a further parameter involved: the origin of the transformation. If you are rotating, for example, it will, by default, turn on the point that is the center of the box; if you had a piece of card, stuck a pin right through the middle of it, and then stuck that to your forehead (don’t do this), the card would spin from the middle. You can change where in the card the pin is stuck with transform-origin, however:

transform-origin: 0 0;

This example will tell the box to transform (rotate, in the previous example) from the top left, the first “0” being horizontal, the second being vertical. These values could be different, of course — like all other x-y, and you can use the usual center, top, right, bottom, left, length and percentage values, including negatives.



**Fig. 8.7. The same clockwise-rotated box with different transform origins.**

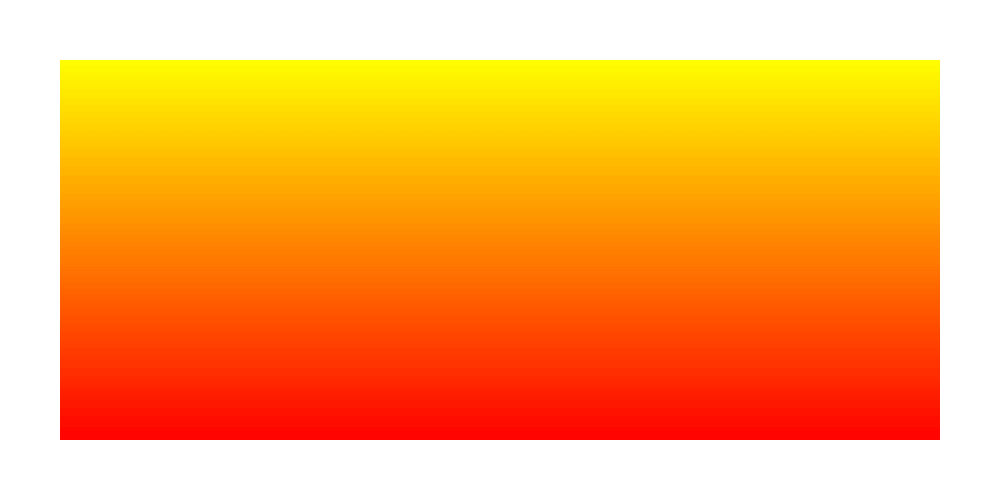
And all of that’s just with two measly dimensions. transform is a leviathan with even greater power that can also be used for three-dimensional magic. On the most basic level, you can use rotateX and rotateY, which will rotate “towards” or “away” from you on the x- and y-axis, and there are the likes of translate3d, scale3d, and the intimidating matrix3d, all of which have even greater browser difficulties than their 2D counterparts.

Images showing a smooth dissolve from one color to another are plastered all over the web. However, CSS 3 allows you to place them in your designs without having to create an actual image file.

There is no special property for this; you simply use the background or background-image property and define your gradient in its value. You can create both linear and radial gradients this way.

For an even spread of two colors, fading from one at the top to another at the bottom, a declaration can simply be something like:

background: linear-gradient(yellow, red);



**Fig. 8.8. A simple yellow-to-red linear gradient background.**

To manipulate the angle of the fading, you slot in “to” and the destination you want the transition to head to. You can head to one side:

background: linear-gradient(to right, orange, red);

Or one corner:

background: linear-gradient(to bottom right, orange, red);

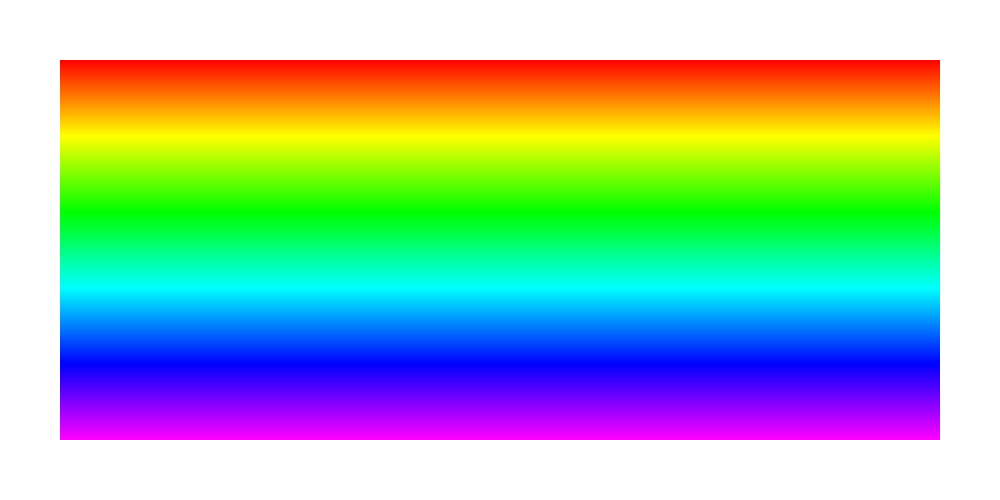
Or any angle that tickles your fancy:

background: linear-gradient(20deg, orange, red);

Note that the “to” is excluded with angles because there isn’t an explicit destination.

And why stop at two colors? Specify as many as you dare:

background: linear-gradient(hsl(0,100%,50%),hsl(60,100%,50%),hsl(120,100%,50%),hsl(180,100%,50%),hsl(240,100%,50%),hsl(300,100%,50%));



**Fig. 8.9. A colorful gradient extravaganza.**

To get gradients to work on as many browsers as possible, you will probably also want to use -webkit-linear-gradient to cover Safari and older versions of Chrome. The values of these must also exclude the “to” part, if used.

CSS gradients won’t play at all with IE 9 and below, but you can still make them, and any other incapable browser, use the traditional method of a good old fashioned image by specifying that first, as a fallback (proceeding declarations will just be ignored).

So, all-in, your gradients might look something like this:

body {

background: #666 url(fade.png) 0 0 repeat-y;

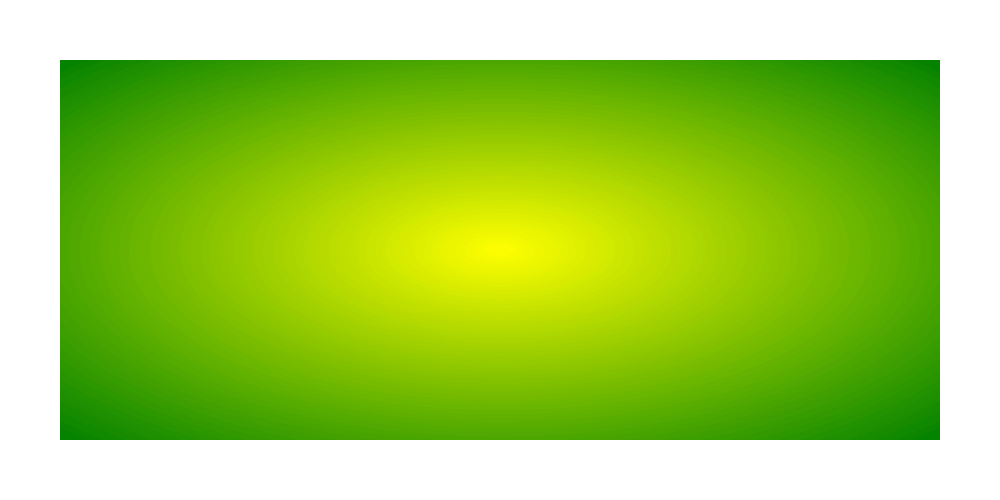
background: -webkit-linear-gradient(right, #000, #666);

background: linear-gradient(to right, #000, #666);

}

Radial gradients, with one color starting from a central point and fading to another color, use a similar syntax:

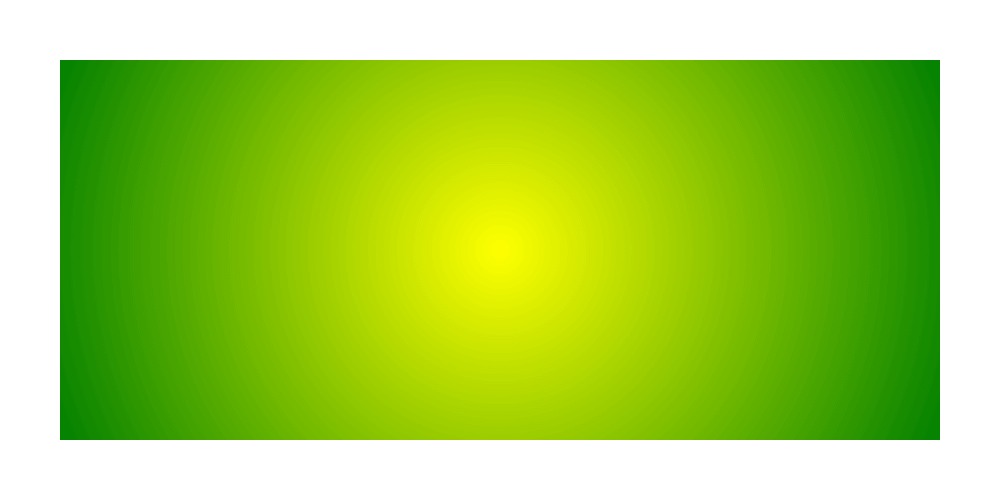
background: radial-gradient(yellow, green);



**Fig. 8.10. A simple yellow-to-green radial gradient background.**

You can also specify the shape of the fade. By default it is an ellipse, spreading to fill the background box, but you can force it to be circular, regardless of the shape of the box:

background: radial-gradient(circle, yellow, green);



**Fig. 8.10. A circular radial gradient.**

Using “closest-side”, “closest-corner”, “farthest-side” and “farthest-corner” you can also specify if the gradient is contained by the sides or corners of the box nearest to or furthest away from the origin:

background: radial-gradient(circle closest-side, yellow, green);

And if you wanted to place the origin of the gradient somewhere specific, you can also use “at”:

background: radial-gradient(at top left, yellow, green);



**Fig. 8.11. A radial gradient emanating from the top-left corner of a box.**

More compatibility notes: additional -webkit-radial-gradient usage is wise. The order of the parameters needs to be changed here, and, like “to”, “at” is excluded. So:

background: radial-gradient(circle closest-side at 100px 200px , black, white);

Would be accompanied by:

background: -webkit-radial-gradient(100px 200px, circle closest-side, black, white);

If you don’t want a uniform blend across your gradient, you can specify exactly where in the gradient each color kicks in, straight after each color, starting at “0” and ending at “100%” (although lengths can also be used).

So, just to make it clear before tinkering:

* linear-gradient(black 0, white 100%) is the equivalent of linear-gradient(black, white)
* radial-gradient(#06c 0, #fc0 50%, #039 100%) is the same as radial-gradient(#06c, #fc0, #039)
* linear-gradient(red 0%, green 33.3%, blue 66.7%, black 100%) will have the same result as linear-gradient(red, green, blue, black)

That’s because, when the positions are stated in these examples, they evenly space out the colors, which is the default when no color stops are explicitly defined.

So, to get on with that tinkering, you can pull and stretch away with those stops:

background: linear-gradient(135deg, hsl(36,100%,50%) 10%, hsl(72,100%,50%) 60%, white 90%);

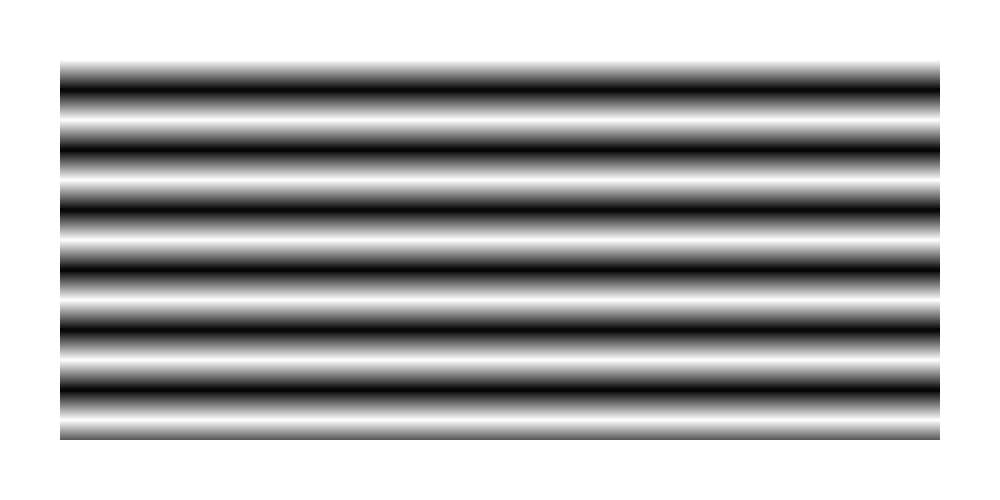


**Fig. 8.12. Colour stops in a linear gradient.**

A single gradient will fill a box with the previous methods but you can use “repeating-linear-gradient” and “repeating-linear-gradient” to build on the color stops and, well, repeat the gradient.

For basic bars of black-and-white bars, for example:

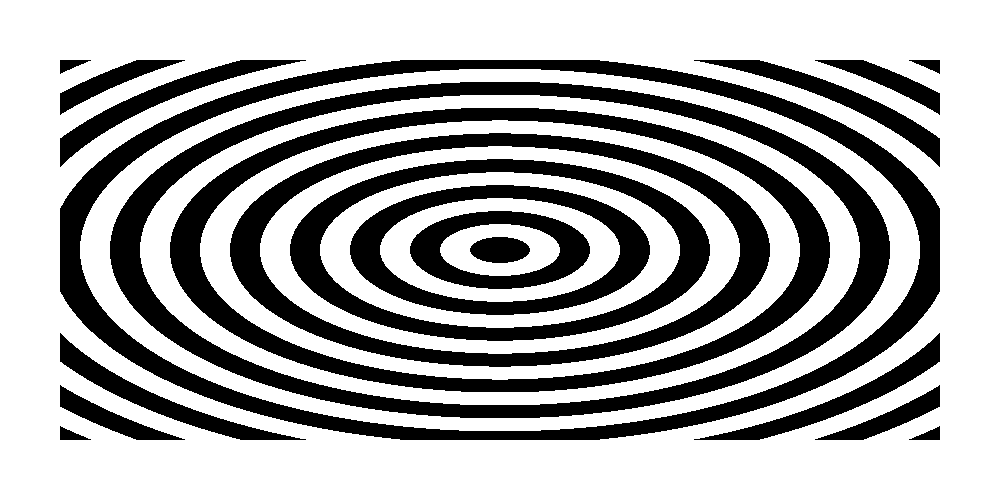
background: repeating-linear-gradient(white, black 15px, white 30px);

****

**Fig. 8.13. A repeating linear gradient.**

Or something a bit more solid:

background: repeating-radial-gradient(black, black 15px, white 15px, white 30px);

****

**Fig. 8.14. A repeating radial gradient.**

@media at-rules, used to target styles at specific media, such as screen or print, have already been covered. But this can be pushed to an even greater level of sophistication, enabling you to specify different design choices depending on screen size. A page can then be optimized and laid out completely differently for mobile phones, tablets, and varying browser window sizes.

To recap, if we want to apply some CSS solely to screen-based media, for example, one option would be to slot something like the following in at the bottom of a stylesheet:

@media screen {

body { font: 12px arial, sans-serif }

#nav { display: block }

}

To extend the previous example, not only can screen-based media be targeted, screen-based media of a certain size can be targeted as well:

@media screen and (max-width: 1000px) {

#content { width: 100% }

}

This is telling a browser to apply a block of CSS when its viewport is 1000 pixels wide or less. You could use this to do something as simple as making a content area or navigation narrower or you could completely re-arrange a page layout (like stacking page sections on top of one another instead of displaying them in columns).

You can apply more than one @media rule, so you could have a number of different designs that are browser size dependent:

@media screen and (max-width: 1000px) {

#content { width: 100% }

}

@media screen and (max-width: 800px) {

#nav { float: none }

}

@media screen and (max-width: 600px) {

#content aside {

float: none;

display: block;

}

}

Note that if, for example, a layout area was 600 pixels wide or less, all three of these would be applied (because it would be less than or equal to 1000px, 800px, and 600px).

As well as using a general max-width on the main content area (the article elements), this site also has a number of size-dependent CSS rules. If you’re able to do so, try resizing your browser to see the changes take effect.

You could also use “min-width” in place of “max-width” if you want to do things the other way around and apply CSS to sizes equal to or over a certain length.

If you have a hankering for applying CSS depending on the orientation of the browser, fill your boots with something like the following:

@media screen and (orientation: landscape) {

#nav { float: left }

}

@media screen and (orientation: portrait) {

#nav { float: none }

}

This could be especially useful with mobile devices…

We’re not talking different CSS for each and every brand and model of laptop, phone, and donkey - that would be sinful - but we can, with a relatively clear conscience (as long as we’re sensible), specify the likes of the device’s screen dimensions and its pixel ratio.

A “handheld” media type does exist and it could be used as @media handheld… but it isn’t widely supported and the distinction of what is “handheld” has become blurred due to the proliferation of all manner of devices with all manner of screen sizes.

device-width, device-height, min-device-width, max-device-width, min-device-height and max-device-height can be used to target the computed resolution of a device:

@media screen and (min-device-height: 768px) and (max-device-width: 1024px) {

/\* You can apply numerous conditions separated by "and" \*/

}

It’s important to keep in mind that a CSS pixel need not be the same as a physical pixel. While a screen may physically be 720 pixels wide, a browser may actually apply CSS assuming that it is 480 pixels wide, for example. This is so that a standard designed web page will more likely fit on the screen. In this example, there is a pixel ratio of 1.5:1: There are 1½ physical pixels to every CSS pixel. A bog-standard desktop monitor will have a pixel ratio of 1:1: One CSS pixel to every physical pixel.

If you want to apply styles only to these devices, you can use something like:

@media (device-pixel-ratio: 2) {

body { background: url(twiceasbig.png) }

}

This would apply to the likes of the iPhone (4 and above), with a pixel ratio of 2:1. You can also use min-device-pixel-ratio and max-device-pixel-ratio.

Talking of iPhones, you know the deal: also use -webkit-device-pixel-ratio, etc, etc…

You might also want to fiddle with a device’s viewport to get it to play how you want. Leaping back over to HTML, the following meta element will force a device to render a page at the width of the viewport (rather than attempting to show a wider-width design and zooming out, which it will do by default if the page can be wider than the viewport’s width) and also prevent the user from zooming in and out:

<meta name="viewport" content="width=device-width, initial-scale=1, maximum-scale=1, user-scalable=no">

The benefit of this is that you can control exactly what is applied to what physical screen size. And although it’s painful to remove user controls, if the design is delightfully swell and made just for that diddy little screen, there shouldn’t be a need to zoom.

The HTML Dog web site takes this approach: instead of a small device attempting to render a bigger, fatter web page by shrinking it down, the CSS turns it into a single-column design made specifically for such a device.

You can also apply styles depending on a device’s resolution:

@media screen and (resolution: 326dpi) { /\* \*/ }

@media screen and (min-resolution: 96dpi) { /\* \*/ }

Or on its aspect ratio:

@media screen and (device-aspect-ratio: 16/9) { /\* \*/ }

**Questions**

1. Explain how to use media queries for different screens.
2. What transforms are possible in CSS?
3. How to use gradient for the background?

Nanoo.

## 

## LECTURE 9

USING FLEXBOX TO FORMAT TEXT

The Flexbox Layout (Flexible Box) module (a W3C Candidate Recommendation as of October 2017) aims at providing a more efficient way to lay out, align and distribute space among items in a container, even when their size is unknown and/or dynamic (thus the word “flex”).

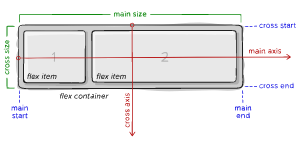
The main idea behind the flex layout is to give the container the ability to alter its items’ width/height (and order) to best fill the available space (mostly to accommodate to all kind of display devices and screen sizes). A flex container expands items to fill available free space or shrinks them to prevent overflow.

Most importantly, the flexbox layout is direction-agnostic as opposed to the regular layouts (block which is vertically-based and inline which is horizontally-based). While those work well for pages, they lack flexibility (no pun intended) to support large or complex applications (especially when it comes to orientation changing, resizing, stretching, shrinking, etc.).

Note: Flexbox layout is most appropriate to the components of an application, and small-scale layouts, while the Grid layout is intended for larger scale layouts.

Since flexbox is a whole module and not a single property, it involves a lot of things including its whole set of properties. Some of them are meant to be set on the container (parent element, known as “flex container”) whereas the others are meant to be set on the children (said “flex items”).

If “regular” layout is based on both block and inline flow directions, the flex layout is based on “flex-flow directions”. Please have a look at this figure from the specification, explaining the main idea behind the flex layout.



**Fig. 9.1. Display:flex funtioning**

Items will be laid out following either the main axis (from main-start to main-end) or the cross axis (from cross-start to cross-end).

* main axis – The main axis of a flex container is the primary axis along which flex items are laid out. Beware, it is not necessarily horizontal; it depends on the flex-direction property (see below).
* main-start | main-end – The flex items are placed within the container starting from main-start and going to main-end.
* main size – A flex item’s width or height, whichever is in the main dimension, is the item’s main size. The flex item’s main size property is either the ‘width’ or ‘height’ property, whichever is in the main dimension.
* cross axis – The axis perpendicular to the main axis is called the cross axis. Its direction depends on the main axis direction.
* cross-start | cross-end – Flex lines are filled with items and placed into the container starting on the cross-start side of the flex container and going toward the cross-end side.
* cross size – The width or height of a flex item, whichever is in the cross dimension, is the item’s cross size. The cross size property is whichever of ‘width’ or ‘height’ that is in the cross dimension.

## Properties for the Parent

## (flex container)

#### **display**

This defines a flex container; inline or block depending on the given value. It enables a flex context for all its direct children.

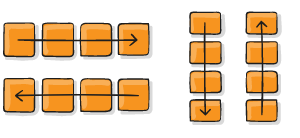
.container {

display: flex; /\* or inline-flex \*/

}

Note that CSS columns have no effect on a flex container.

#### flex-direction



**Fig.9.2. Usage of flex-direction**

This establishes the main-axis, thus defining the direction flex items are placed in the flex container. Flexbox is (aside from optional wrapping) a single-direction layout concept. Think of flex items as primarily laying out either in horizontal rows or vertical columns.

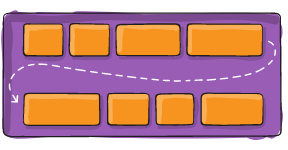
.container {

flex-direction: row | row-reverse | column | column-reverse;

}

* row (default): left to right in ltr; right to left in rtl
* row-reverse: right to left in ltr; left to right in rtl
* column: same as row but top to bottom
* column-reverse: same as row-reverse but bottom to top

#### flex-wrap



**Fig.9.2. Usage of flex-wrap**

By default, flex items will all try to fit onto one line. You can change that and allow the items to wrap as needed with this property.

.container {

flex-wrap: nowrap | wrap | wrap-reverse;

}

* nowrap (default): all flex items will be on one line
* wrap: flex items will wrap onto multiple lines, from top to bottom.
* wrap-reverse: flex items will wrap onto multiple lines from bottom to top.

#### flex-flow

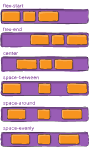
This is a shorthand for the flex-direction and flex-wrap properties, which together define the flex container’s main and cross axes. The default value is row nowrap.

.container {

flex-flow: column wrap;

}

#### justify-content



**Fig.9.3. Usage justify-content**

This defines the alignment along the main axis. It helps distribute extra free space leftover when either all the flex items on a line are inflexible, or are flexible but have reached their maximum size. It also exerts some control over the alignment of items when they overflow the line.

.container {

justify-content: flex-start | flex-end | center | space-between | space-around | space-evenly | start | end | left | right ... + safe | unsafe;

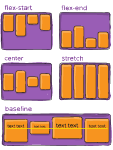
}

* flex-start (default): items are packed toward the start of the flex-direction.
* flex-end: items are packed toward the end of the flex-direction.
* start: items are packed toward the start of the writing-mode direction.
* end: items are packed toward the end of the writing-mode direction.
* left: items are packed toward left edge of the container, unless that doesn’t make sense with the flex-direction, then it behaves like start.
* right: items are packed toward right edge of the container, unless that doesn’t make sense with the flex-direction, then it behaves like end.
* center: items are centered along the line
* space-between: items are evenly distributed in the line; first item is on the start line, last item on the end line
* space-around: items are evenly distributed in the line with equal space around them. Note that visually the spaces aren’t equal, since all the items have equal space on both sides. The first item will have one unit of space against the container edge, but two units of space between the next item because that next item has its own spacing that applies.
* space-evenly: items are distributed so that the spacing between any two items (and the space to the edges) is equal.

Note that that browser support for these values is nuanced. For example, space-between never got support from some versions of Edge, and start/end/left/right aren’t in Chrome yet. MDN has detailed charts. The safest values are flex-start, flex-end, and center.

There are also two additional keywords you can pair with these values: safe and unsafe. Using safe ensures that however you do this type of positioning, you can’t push an element such that it renders off-screen (e.g. off the top) in such a way the content can’t be scrolled too (called “data loss”).

#### align-items



**Fig.9.4. Usage of align-items**

This defines the default behavior for how flex items are laid out along the cross axis on the current line. Think of it as the justify-content version for the cross-axis (perpendicular to the main-axis).

.container {

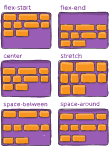
align-items: stretch | flex-start | flex-end | center | baseline | first baseline | last baseline | start | end | self-start | self-end + ... safe | unsafe;

}

* stretch (default): stretch to fill the container (still respect min-width/max-width)
* flex-start / start / self-start: items are placed at the start of the cross axis. The difference between these is subtle, and is about respecting the flex-direction rules or the writing-mode rules.
* flex-end / end / self-end: items are placed at the end of the cross axis. The difference again is subtle and is about respecting flex-direction rules vs. writing-mode rules.
* center: items are centered in the cross-axis
* baseline: items are aligned such as their baselines align

The safe and unsafe modifier keywords can be used in conjunction with all the rest of these keywords (although note browser support), and deal with helping you prevent aligning elements such that the content becomes inaccessible.

#### align-content



**Fig.9.5. Usage of align-content**

This aligns a flex container’s lines within when there is extra space in the cross-axis, similar to how justify-content aligns individual items within the main-axis.

Note: This property only takes effect on multi-line flexible containers, where flex-flow is set to either wrap or wrap-reverse). A single-line flexible container (i.e. where flex-flow is set to its default value, no-wrap) will not reflect align-content.

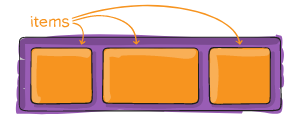
.container {

align-content: flex-start | flex-end | center | space-between | space-around | space-evenly | stretch | start | end | baseline | first baseline | last baseline + ... safe | unsafe;

}

* normal (default): items are packed in their default position as if no value was set.
* flex-start / start: items packed to the start of the container. The (more supported) flex-start honors the flex-direction while start honors the writing-mode direction.
* flex-end / end: items packed to the end of the container. The (more support) flex-end honors the flex-direction while end honors the writing-mode direction.
* center: items centered in the container
* space-between: items evenly distributed; the first line is at the start of the container while the last one is at the end
* space-around: items evenly distributed with equal space around each line
* space-evenly: items are evenly distributed with equal space around them
* stretch: lines stretch to take up the remaining space

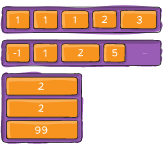
The safe and unsafe modifier keywords can be used in conjunction with all the rest of these keywords (although note browser support), and deal with helping you prevent aligning elements such that the content becomes inaccessible.



**Fig.9.6. Children items**

## Properties for the Children

#### order



**Fig.9.7. Order option**

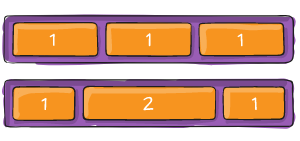
By default, flex items are laid out in the source order. However, the order property controls the order in which they appear in the flex container.

.item {

order: 5; /\* default is 0 \*/

}

#### flex-grow



**Fig.9.8. Usage of Flex-grow**

This defines the ability for a flex item to grow if necessary. It accepts a unitless value that serves as a proportion. It dictates what amount of the available space inside the flex container the item should take up.

If all items have flex-grow set to 1, the remaining space in the container will be distributed equally to all children. If one of the children has a value of 2, the remaining space would take up twice as much space as the others (or it will try to, at least).

.item {

flex-grow: 4; /\* default 0 \*/

}

Negative numbers are invalid.

#### flex-shrink

This defines the ability for a flex item to shrink if necessary.

.item {

flex-shrink: 3; /\* default 1 \*/

}

Negative numbers are invalid.

#### flex-basis

This defines the default size of an element before the remaining space is distributed. It can be a length (e.g. 20%, 5rem, etc.) or a keyword. The auto keyword means “look at my width or height property” (which was temporarily done by the main-size keyword until deprecated). The content keyword means “size it based on the item’s content” – this keyword isn’t well supported yet, so it’s hard to test and harder to know what its brethren max-content, min-content, and fit-content do.

.item {

flex-basis: | auto; /\* default auto \*/

}

If set to 0, the extra space around content isn’t factored in. If set to auto, the extra space is distributed based on its flex-grow value.

#### flex

This is the shorthand for flex-grow, flex-shrink and flex-basis combined. The second and third parameters (flex-shrink and flex-basis) are optional. The default is 0 1 auto, but if you set it with a single number value, it’s like 1 0.

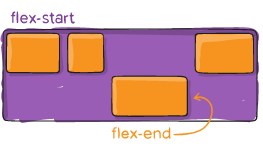
.item {

flex: none | [ <'flex-grow'> <'flex-shrink'>? || <'flex-basis'> ]

}

It is recommended that you use this shorthand property rather than set the individual properties. The shorthand sets the other values intelligently.

#### align-self



**Fig. 9.9. Usage of align-self**

This allows the default alignment (or the one specified by align-items) to be overridden for individual flex items.

Please see the align-items explanation to understand the available values.

.item {

align-self: auto | flex-start | flex-end | center | baseline | stretch;

}

Note that float, clear and vertical-align have no effect on a flex item.

Flexbox requires some vendor prefixing to support the most browsers possible. It doesn’t just include prepending properties with the vendor prefix, but there are actually entirely different property and value names. This is because the Flexbox spec has changed over time, creating an “old”, “tweener”, and “new” versions.

Perhaps the best way to handle this is to write in the new (and final) syntax and run your CSS through Autoprefixer, which handles the fallbacks very well.

Alternatively, here’s a Sass @mixin to help with some of the prefixing, which also gives you an idea of what kind of things need to be done:

@mixin flexbox() {

display: -webkit-box;

display: -moz-box;

display: -ms-flexbox;

display: -webkit-flex;

display: flex;

}

@mixin flex($values) {

-webkit-box-flex: $values;

-moz-box-flex: $values;

-webkit-flex: $values;

-ms-flex: $values;

flex: $values;

}

@mixin order($val) {

-webkit-box-ordinal-group: $val;

-moz-box-ordinal-group: $val;

-ms-flex-order: $val;

-webkit-order: $val;

order: $val;

}

.wrapper {

@include flexbox();

}

.item {

@include flex(1 200px);

@include order(2);

}

Flexbox is certainly not without its bugs. The best collection of them is Philip Walton and Greg Whitworth’s Flexbugs. It’s an open-source place to track all of them, so it’s best to just link to that.

**Questions**

1. What CSS rules are used to position the container?
2. What CSS rules are used to position the children elements?
3. Which browsers support flex?

REFERENCES

1. Interneting is hard [Online]. Access mode: <https://www.internetingishard.com/>
2. HTML, CSS and JS Tutorials [Online]. Access mode: <https://www.htmldog.com/>
3. A complete guide to flexbox [Online]. Access mode: <https://css-tricks.com/snippets/css/a-guide-to-flexbox/>
4. How does the Internet work? [Online]. Access mode: <https://user3141592.medium.com/how-does-the-internet-work-edc2e22e7eb8>