# What is the main functionality of the browser?

The software application used to access information on the World Wide Web is called Web Browser. When a user requests certain information, the web browser downloads the data from the web server and displays the web page on the user screen.  
  
The main task is to retrieve the information from the World Wide Web and make it available to users.

Visiting any website can be done using a web browser. Once the URL is entered in the browser, the web server redirects it to that website

# High Level Components of a browser

# **The user interface**: this includes the address bar, back/forward button, bookmarking menu, etc. Every part of the browser displays except the window where you see the requested page.

1. **The browser engine**: marshals actions between the UI and the rendering engine.
2. **The rendering engine:** responsible for displaying requested content. For example, if the requested content is HTML, the rendering engine parses HTML and CSS, and displays the parsed content on the screen.
3. **Networking**: for network calls such as HTTP requests, using different implementations for different platforms behind a platform-independent interface.
4. **UI backend**: used for drawing basic widgets like combo boxes and windows. This backend exposes a generic interface that is not platform specific. Underneath it uses operating system user interface methods.
5. **JavaScript interpreter**. Used to parse and execute JavaScript code.
6. **Data storage**. This is a persistence layer. The browser may need to save all sorts of data locally, such as cookies. Browsers also support storage mechanisms such as localStorage, IndexedDB, WebSQL and FileSystem.



# Rendering engine and its use.

The responsibility of the rendering engine is well... Rendering, that is display of the requested contents on the browser screen.

By default, the rendering engine can display HTML and XML documents and images. It can display other types of data via plug-ins or extensions; for example, displaying PDF documents using a PDF viewer plug-in. However, in this chapter we will focus on the main use case: displaying HTML and images that are formatted using CSS.

## **Rendering engines**

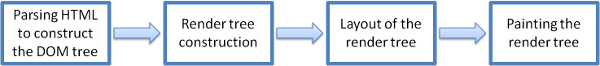
Different browsers use different rendering engines: Internet Explorer uses Trident, Firefox uses Gecko, Safari uses WebKit. Chrome and Opera (from version 15) use Blink, a fork of WebKit.

WebKit is an open-source rendering engine which started as an engine for the Linux platform and was modified by Apple to support Mac and Windows.

**The main flow**

The rendering engine will start getting the contents of the requested document from the networking layer. This will usually be done in 8kB chunks.

After that, this is the basic flow of the rendering engine:



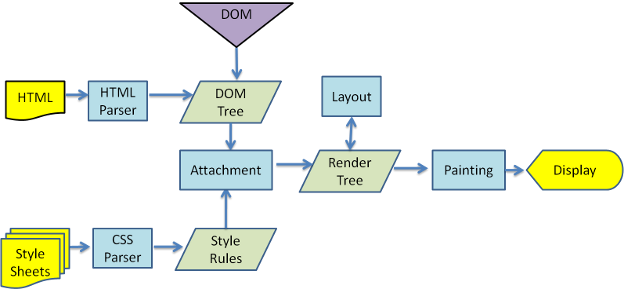
The rendering engine will start parsing the HTML document and convert elements to DOM nodes in a tree called the "content tree". The engine will parse the style data, both in external CSS files and in style elements. Styling information together with visual instructions in the HTML will be used to create another tree: the render tree.

The render tree contains rectangles with visual attributes like color and dimensions. The rectangles are in the right order to be displayed on the screen.

After the construction of the render tree it goes through a "layout" process. This means giving each node the exact coordinates where it should appear on the screen. The next stage is painting–the render tree will be traversed and each node will be painted using the UI backend layer.

It's important to understand that this is a gradual process. For better user experience, the rendering engine will try to display contents on the screen as soon as possible. It will not wait until all HTML is parsed before starting to build and layout the render tree. Parts of the content will be parsed and displayed, while the process continues with the rest of the content that keeps coming from the network.

### **Main flow examples**

  
  
Gecko calls the tree of visually formatted elements a "Frame tree". Each element is a frame. WebKit uses the term "Render Tree" and it consists of "Render Objects". WebKit uses the term "layout" for the placing of elements, while Gecko calls it "Reflow". "Attachment" is WebKit's term for connecting DOM nodes and visual information to create the render tree. A minor non-semantic difference is that Gecko has an extra layer between the HTML and the DOM tree. It is called the "content sink" and is a factory for making DOM elements. We will talk about each part of the flow:

# Parsers (HTML, CSS, etc)

The job of the HTML parser is to parse the HTML markup into a parse tree  
**The HTML grammar definition**

The vocabulary and syntax of HTML are defined in specifications created by the W3C organization.

### **Not a context free grammar**

As we have seen in the parsing introduction, grammar syntax can be defined formally using formats like BNF.

Unfortunately all the conventional parser topics do not apply to HTML (I didn't bring them up just for fun–they will be used in parsing CSS and JavaScript). HTML cannot easily be defined by a context free grammar that parsers need.

There is a formal format for defining HTML–DTD (Document Type Definition)–but it is not context free grammar.

This appears strange at first sight; HTML is rather close to XML. There are lots of available XML parsers. There is an XML variation of HTML–XHTML–so what's the big difference?

The difference is that the HTML approach is more "forgiving": it lets you omit certain tags (which are then added implicitly), or sometimes omit start or end tags, and so on. On the whole it's a "soft" syntax, as opposed to XML's stiff and demanding syntax.

This seemingly small detail makes a world of a difference. On one hand this is the main reason why HTML is so popular: it forgives your mistakes and makes life easy for the web author. On the other hand, it makes it difficult to write formal grammar. So to summarize, HTML cannot be parsed easily by conventional parsers, since its grammar is not context free. HTML cannot be parsed by XML parsers.

### **HTML DTD**

HTML definition is in a DTD format. This format is used to define languages of the SGML family. The format contains definitions for all allowed elements, their attributes and hierarchy. As we saw earlier, the HTML DTD doesn't form a context-free grammar.

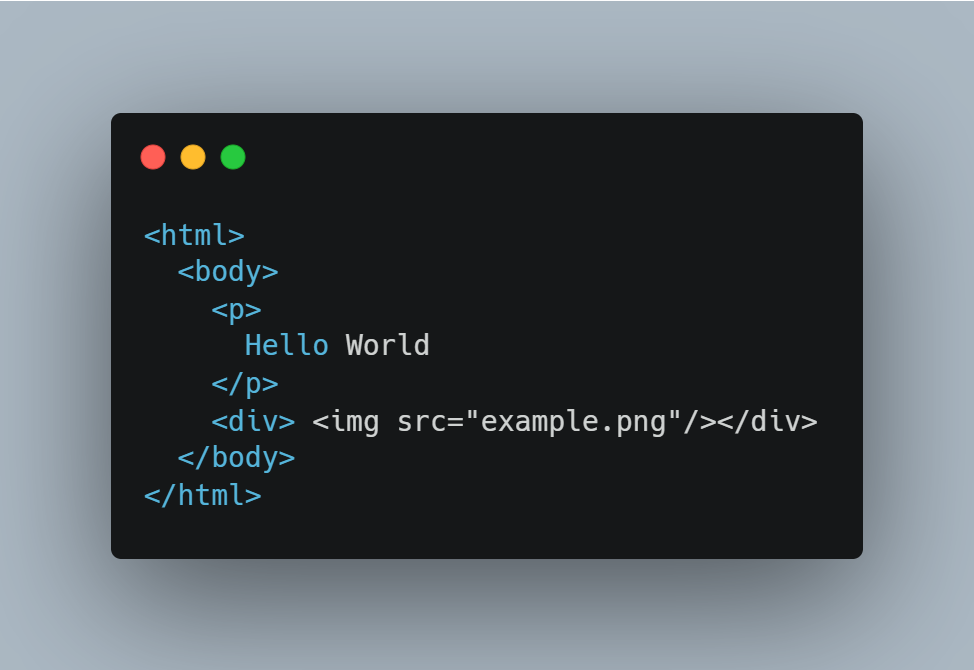
There are a few variations of the DTD. The strict mode conforms solely to the specifications but other modes contain support for markup used by browsers in the past. The purpose is backwards compatibility with older content.

**DOM**

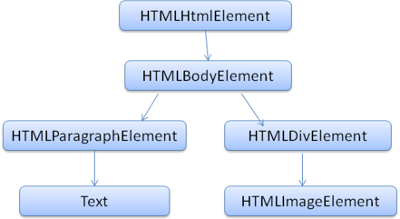
The output tree (the "parse tree") is a tree of DOM elements and attribute nodes. DOM is short for Document Object Model. It is the object presentation of the HTML document and the interface of HTML elements to the outside world like JavaScript.

The root of the tree is the "Document" object.

The DOM has an almost one-to-one relation to the markup. For example:



This markup would be translated to the following DOM tree:



#### **The parsing algorithm**

As we saw in the previous sections, HTML cannot be parsed using the regular top down or bottom up parsers.

The reasons are:

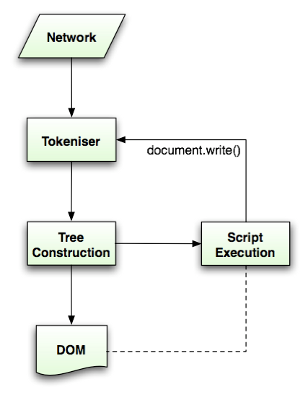
1. The forgiving nature of the language.
2. The fact that browsers have traditional error tolerance to support well known cases of invalid HTML.
3. The parsing process is reentrant. For other languages, the source doesn't change during parsing, but in HTML, dynamic code (such as script elements containing document.write() calls) can add extra tokens, so the parsing process actually modifies the input.

Unable to use the regular parsing techniques, browsers create custom parsers for parsing HTML.

The parsing algorithm is described in detail by the HTML5 specification. The algorithm consists of two stages: tokenization and tree construction.

Tokenization is the lexical analysis, parsing the input into tokens. Among HTML tokens are start tags, end tags, attribute names and attribute values.

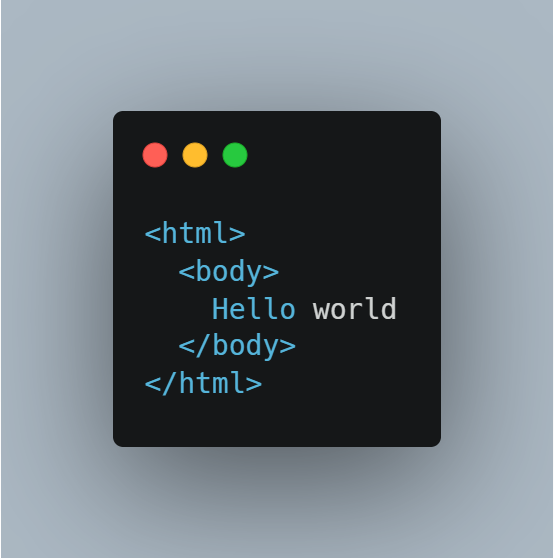
The tokenizer recognizes the token, gives it to the tree constructor, and consumes the next character for recognizing the next token, and so on until the end of the input.



**The tokenization algorithm**

The algorithm's output is an HTML token. The algorithm is expressed as a state machine. Each state consumes one or more characters of the input stream and updates the next state according to those characters. The decision is influenced by the current tokenization state and by the tree construction state. This means the same consumed character will yield different results for the correct next state, depending on the current state. The algorithm is too complex to describe fully, so let's see a simple example that will help us understand the principle.

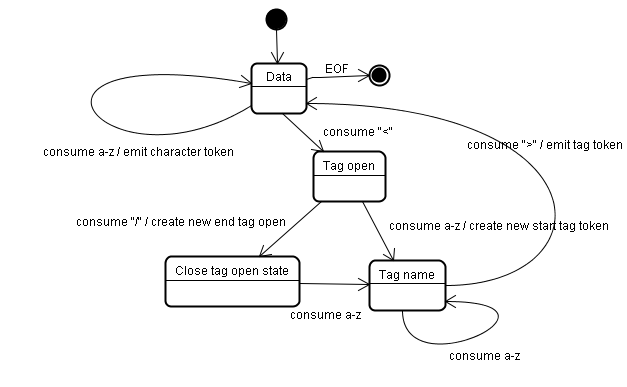
Basic example–tokenizing the following HTML:



The initial state is the "Data state". When the < character is encountered, the state is changed to **"Tag open state"**. Consuming an a-z character causes creation of a "Start tag token", the state is changed to **"Tag name state"**. We stay in this state until the > character is consumed. Each character is appended to the new token name. In our case the created token is an html token.

When the > tag is reached, the current token is emitted and the state changes back to the **"Data state"**. The <body> tag will be treated by the same steps. So far the html and body tags were emitted. We are now back at the **"Data state"**. Consuming the H character of Hello world will cause creation and emitting of a character token, this goes on until the < of </body> is reached. We will emit a character token for each character of Hello world.

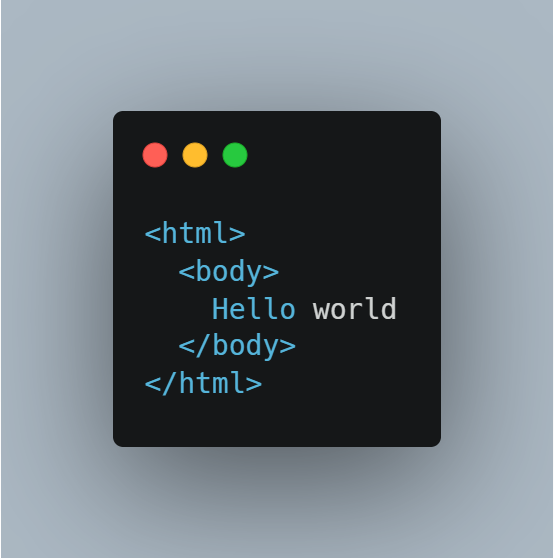
We are now back at the **"Tag open state"**. Consuming the next input / will cause creation of an end tag token and a move to the **"Tag name state"**. Again we stay in this state until we reach >.Then the new tag token will be emitted and we go back to the **"Data state"**. The </html> input will be treated like the previous case.



#### **Tree construction algorithm**

When the parser is created the Document object is created. During the tree construction stage the DOM tree with the Document in its root will be modified and elements will be added to it. Each node emitted by the tokenizer will be processed by the tree constructor. For each token the specification defines which DOM element is relevant to it and will be created for this token. The element is added to the DOM tree, and also the stack of open elements. This stack is used to correct nesting mismatches and unclosed tags. The algorithm is also described as a state machine. The states are called "insertion modes".

Let's see the tree construction process for the example input:



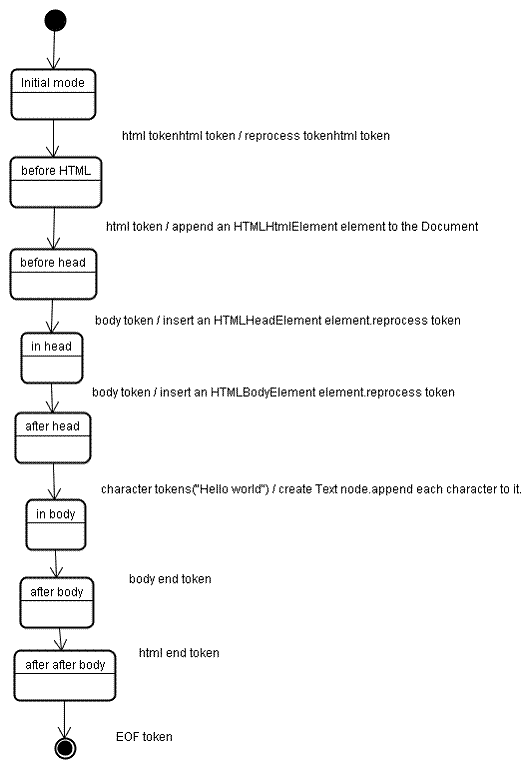
The input to the tree construction stage is a sequence of tokens from the tokenization stage. The first mode is the **"initial mode"**. Receiving the "html" token will cause a move to the **"before html"** mode and a reprocessing of the token in that mode. This will cause creation of the HTMLHtmlElement element, which will be appended to the root Document object.

The state will be changed to **"before head"**. The "body" token is then received. An HTMLHeadElement will be created implicitly although we don't have a "head" token and it will be added to the tree.

We now move to the **"in head"** mode and then to **"after head"**. The body token is reprocessed, an HTMLBodyElement is created and inserted and the mode is transferred to **"in body"**.

The character tokens of the "Hello world" string are now received. The first one will cause creation and insertion of a "Text" node and the other characters will be appended to that node.

The receiving of the body end token will cause a transfer to **"after body"** mode. We will now receive the html end tag which will move us to **"after after body"** mode. Receiving the end of file token will end the parsing.



### 

### **Actions when the parsing is finished**

At this stage the browser will mark the document as interactive and start parsing scripts that are in "deferred" mode: those that should be executed after the document is parsed. The document state will be then set to "complete" and a "load" event will be fired.

## **CSS parsing**

Let's see some examples:

The lexical grammar (vocabulary) is defined by regular expressions for each token:

comment \/\\*[^\*]\*\\*+([^/\*][^\*]\*\\*+)\*\/

num [0-9]+|[0-9]\*"."[0-9]+

nonascii [\200-\377]

nmstart [\_a-z]|{nonascii}|{escape}

nmchar [\_a-z0-9-]|{nonascii}|{escape}

name {nmchar}+

ident {nmstart}{nmchar}\*

"ident" is short for identifier, like a class name. "name" is an element id (that is referred by "#" )

The syntax grammar is described in BNF.  
  
**The order of processing scripts and style sheets**

#### **Scripts**

The model of the web is synchronous. Authors expect scripts to be parsed and executed immediately when the parser reaches a <script> tag. The parsing of the document halts until the script has been executed. If the script is external then the resource must first be fetched from the network–this is also done synchronously, and parsing halts until the resource is fetched. This was the model for many years and is also specified in HTML4 and 5 specifications. Authors can add the "defer" attribute to a script, in which case it will not halt document parsing and will execute after the document is parsed. HTML5 adds an option to mark the script as asynchronous so it will be parsed and executed by a different thread.

### **Speculative parsing**

Both WebKit and Firefox do this optimization. While executing scripts, another thread parses the rest of the document and finds out what other resources need to be loaded from the network and loads them. In this way, resources can be loaded on parallel connections and overall speed is improved. Note: the speculative parser only parses references to external resources like external scripts, style sheets and images: it doesn't modify the DOM tree–that is left to the main parser.

### **Style sheets**

Style sheets on the other hand have a different model. Conceptually it seems that since style sheets don't change the DOM tree, there is no reason to wait for them and stop the document parsing. There is an issue, though, of scripts asking for style information during the document parsing stage. If the style is not loaded and parsed yet, the script will get wrong answers and apparently this causes lots of problems. It seems to be an edge case but is quite common. Firefox blocks all scripts when there is a style sheet that is still being loaded and parsed. WebKit blocks scripts only when they try to access certain style properties that may be affected by unloaded style sheets.

## **Render tree construction**

While the DOM tree is being constructed, the browser constructs another tree, the render tree. This tree is of visual elements in the order in which they will be displayed. It is the visual representation of the document. The purpose of this tree is to enable painting the contents in their correct order.

**The render tree relation to the DOM tree**

The renderers correspond to DOM elements, but the relation is not one to one. Non-visual DOM elements will not be inserted in the render tree. An example is the "head" element. Also elements whose display value was assigned to "none" will not appear in the tree (whereas elements with "hidden" visibility will appear in the tree).

There are DOM elements which correspond to several visual objects. These are usually elements with complex structure that cannot be described by a single rectangle. For example, the "select" element has three renderers: one for the display area, one for the drop down list box and one for the button. Also when text is broken into multiple lines because the width is not sufficient for one line, the new lines will be added as extra renderers.

Another example of multiple renderers is broken HTML. According to the CSS spec an inline element must contain either only block elements or only inline elements. In the case of mixed content, anonymous block renderers will be created to wrap the inline elements.

Some render objects correspond to a DOM node but not in the same place in the tree. Floats and absolutely positioned elements are out of flow, placed in a different part of the tree, and mapped to the real frame. A placeholder frame is where they should have been.

# 

#### **The flow of constructing the tree**

In Firefox, the presentation is registered as a listener for DOM updates. The presentation delegates frame creation to the FrameConstructor and the constructor resolves style (see style computation) and creates a frame.

In WebKit the process of resolving the style and creating a renderer is called "attachment". Every DOM node has an "attach" method. Attachment is synchronous, node insertion to the DOM tree calls the new node "attach" method.

Processing the html and body tags results in the construction of the render tree root. The root render object corresponds to what the CSS spec calls the containing block: the top most block that contains all other blocks. Its dimensions are the viewport: the browser window display area dimensions. Firefox calls it ViewPortFrame and WebKit calls it RenderView. This is the render object that the document points to. The rest of the tree is constructed as a DOM nodes insertion.

See [the CSS2 spec on the processing model](http://www.w3.org/TR/CSS21/intro.html#processing-model).

### **Style Computation**

Building the render tree requires calculating the visual properties of each render object. This is done by calculating the style properties of each element.

The style includes style sheets of various origins, inline style elements and visual properties in the HTML (like the "bgcolor" property).The later is translated to matching CSS style properties.

The origins of style sheets are the browser's default style sheets, the style sheets provided by the page author and user style sheets–these are style sheets provided by the browser user (browsers let you define your favorite styles. In Firefox, for instance, this is done by placing a style sheet in the "Firefox Profile" folder).

Style computation brings up a few difficulties:

1. Style data is a very large construct, holding the numerous style properties, this can cause memory problems.

2. Finding the matching rules for each element can cause performance issues if it's not optimized. Traversing the entire rule list for each element to find matches is a heavy task. Selectors can have complex structure that can cause the matching process to start on a seemingly promising path that is proven to be futile and another path has to be tried.  
For example–this compound selector:  
div div div div{

...

1. }  
   Means the rules apply to a <div> who is the descendant of 3 divs. Suppose you want to check if the rule applies for a given <div> element. You choose a certain path up the tree for checking. You may need to traverse the node tree up just to find out there are only two divs and the rule does not apply. You then need to try other paths in the tree.
2. Applying the rules involves quite complex cascade rules that define the hierarchy of the rules.

Let's see how the browsers face these issues:

### **Sharing style data**

WebKit nodes references style objects (RenderStyle). These objects can be shared by nodes in some conditions. The nodes are siblings or cousins and:

1. The elements must be in the same mouse state (e.g., one can't be in: hover while the other isn't)
2. Neither element should have an id
3. The tag names should match
4. The class attributes should match
5. The set of mapped attributes must be identical
6. The link states must match
7. The focus states must match
8. Neither element should be affected by attribute selectors, where affected is defined as having any selector match that uses an attribute selector in any position within the selector at all
9. There must be no inline style attribute on the elements
10. There must be no sibling selectors in use at all. WebCore simply throws a global switch when any sibling selector is encountered and disables style sharing for the entire document when they are present. This includes the + selector and selectors like :first-child and :last-child.

## **Layout**

When the renderer is created and added to the tree, it does not have a position and size. Calculating these values is called layout or reflow.

HTML uses a flow based layout model, meaning that most of the time it is possible to compute the geometry in a single pass. Elements later ``in the flow'' typically do not affect the geometry of elements that are earlier ``in the flow'', so layout can proceed left-to-right, top-to-bottom through the document. There are exceptions: for example, HTML tables may require more than one pass ([3.5](https://www.html5rocks.com/en/tutorials/internals/howbrowserswork/#3_5)).

The coordinate system is relative to the root frame. Top and left coordinates are used.

Layout is a recursive process. It begins at the root renderer, which corresponds to the <html> element of the HTML document. Layout continues recursively through some or all of the frame hierarchy, computing geometric information for each renderer that requires it.

The position of the root renderer is 0,0 and its dimensions are the viewport–the visible part of the browser window.

All renderers have a "layout" or "reflow" method, each renderer invokes the layout method of its children that need layout.

### **The layout processes**

The layout usually has the following pattern:

1. Parent renderer determines its own width.
2. Parent goes over children and:
   1. Place the child renderer (sets its x and y).
   2. Call the child layout if needed–they are dirty or we are in a global layout, or for some other reason–which calculates the child's height.
3. Parent uses the children's accumulative heights and the heights of margins and padding to set its own height–this will be used by the parent renderer's parent.
4. Sets its dirty bit to false.

Firefox uses a "state" object(nsHTMLReflowState) as a parameter to layout (termed "reflow"). Among others the state includes the parents width.

The output of the Firefox layout is a "metrics" object(nsHTMLReflowMetrics). It will contain the renderer's computed height.

## **Painting**

In the painting stage, the render tree is traversed and the renderer's "paint()" method is called to display content on the screen. Painting uses the UI infrastructure component.

### **Global and Incremental**

Like layout, painting can also be global–the entire tree is painted–or incremental. In incremental painting, some of the renderers change in a way that does not affect the entire tree. The changed renderer invalidates its rectangle on the screen. This causes the OS to see it as a "dirty region" and generate a "paint" event. The OS does it cleverly and coalesces several regions into one. In Chrome it is more complicated because the renderer is in a different process then the main process. Chrome simulates the OS behavior to some extent. The presentation listens to these events and delegates the message to the render root. The tree is traversed until the relevant renderer is reached. It will repaint itself (and usually its children).

### **The painting order**

CSS2 defines the order of the painting process. This is actually the order in which the elements are stacked in the stacking contexts. This order affects painting since the stacks are painted from back to front. The stacking order of a block renderer is:

1. background color
2. background image
3. border
4. children
5. outline