“Let’s Learn Something”

15 Short and Easy-To-Follow Lessons in

Web Design and Development
(First Edition)

By

Itoro Akpan-Iquot

July, 2008

About the Author

Itoro Akpan-Iquot has been involved in web development since 1999. Since then, he has helped develop many sites spanning many sectors (banking, insurance, US governments (federal, state and local), entertainment and medicine, to mention but a few).

His core areas of expertise are XHTML, JavaScript and CSS. Other areas of expertise include JSP and PHP. He also makes use of many of the JavaScript libraries (YUI, Scriptaculous, MooTools, and JQuery) out there to implement rich internet applications for his clients.

In 2007, Itoro launched his own web development company, **Ace Web Solutions**, with branches in the US and Nigeria. His ultimate goal is to move back to Nigeria fulltime and do his development from there where he will have access to the upcoming generations of web developers, turning them into better programmers so as to help them compete with their peers in other countries.

Itoro is the webmaster and moderator of the "Webmasters of Africa (WoA)" website, **http://www.webmastersofafrica.com**, which has over 190 members.

Itoro also serves as a consultant. Should you need him to work on your upcoming project(s), please visit his portfolio at **http://www.acewebdevelopment.net/ace-portfolio** and/or feel free to contact him via email at **info@acewebdevelopment.net**.

Who is this Book for?

This book was written with both the novice and expert in mind. No one knows it all. There are things that even the most experienced of developers, including myself find tricky and not so obvious during the course of web development. This book aims to highlight various “gotchas” that I have encountered during the course of my career, “gotchas” that I know many have also encountered by perhaps haven’t been lucky enough to either figure it out by themselves or have someone figure it out for them. Please enjoy the book, and spread the word about it.

Comments? Suggestions? Criticisms? Please send them to:

info@acewebdevelopment.net.

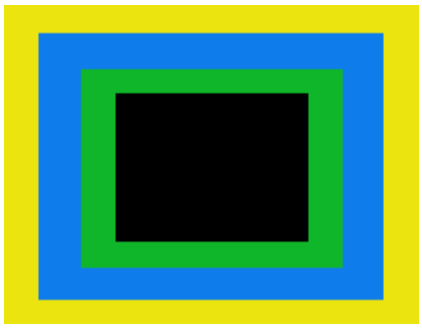
Thanks,

Itoro Akpan-Iquot
Ace Web Solutions
www.acewebdevelopment.net

Table of Contents

Lesson 1: The Box Model

Perhaps the most confusing of concepts with respect to CSS is the box model. If not well understood, it can lead to a lot of frustration resulting in possibly dumping CSS altogether. I will begin with this illustration:



Focus on the different colors. The black represents the element itself; the green represents its padding (the space between the element and "itself"); the blue represents the element's border while the yellow represents its margin (the space between adjacent elements.

Assuming you had a container DIV with a width of 300px and 2 DIVs (left and right) floated left each with a width of 150px, the 2 DIVs would appear side-by-side each other. Now, if I were to change the margin (yellow area) of one of the left DIV from 150px to 155px, one of the DIVs would drop below the other. Why? Because, whereas the width of the container is 300px, the two inner DIVs are now 305px. How do you fix this problem, assuming you want to maintain the container width of 300px? What you do is reduce the width of the left DIV from 150px to 145px so that when you factor in the 5px margin, you still arrive at 150px.

Conclusion:
The Box Model, as stated earlier is perhaps the most tricky and frustrating of all there is to learn with respect to CSS styling. Once mastered however, the sky is your limit.

Lesson 2: CSS Positioning

CSS Positioning (CSSP for short) is another tricky area of CSS. In particular, browser inconsistencies can make it a pain for most developers to implement. As a result, such developers end up abandoning it for tables or whatever design change they can lay their hands on.

For starters, there are 4 position attribute properties:

- **position: static** The default. The only reason you would use this would be if you were overriding a pre-existing position style

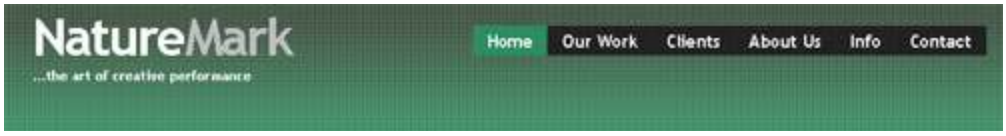
- **position: absolute** The element is positioned at the given co-ordinates within its container. If no container exists, it assumes the default container which is the browser

- **position: relative** The element is positioned relative to its container. As in "position: absolute", if no container exists, it assumes the default container which is the browser.

- **position: fixed** The element is positioned relative to the browser. Once positioned this way, it usually doesn't move and remains fixed at the specified position regardless of scrolling.

Elements are positioned using two properties: top and left. With respect to web development, the two most commonly used position attributes are "absolute" and "relative". These two combine to form a tricky combination for many web developers. To illustrate their power however, I will go through an example:

Let's take this header graphic:



The temptation might be to make the "NatureMark" logo and background one image and then using some kind of table padding, move the menu items up or make the entire menu section and image and again, embed this with the other graphics, and then use an image map to determine clickable areas. The approach, though easy will result in a header that is large in size which in turns adversely affects page download times. The optimal solution would be to have a container DIV positioned relative, then position the "NatureMark (preferably as text)" and the menu absolute (creating a menu list has already been covered in lesson 5). Here is example code that will get the job done (remember, it's best for you to have an external style sheet for your CSS):

```
<div style="position: relative; width: 800px; height: 100px;">
        <img style="position:absolute; top: 10px; left: 30px;" src="natureMarkLogo.gif"
height="50" width="300" />

        <ul style="position: absolute; top: 10px; left: 300px;"><li>Home</li>…</ul>
</div>
```

Please note:
When implementing this code snippet, all inline styles should actually be in an external style sheet in accordance with web standards.

You can always tweak the "top" and "left" properties to achieve the desired results. The numbers I have used are only placeholders.

Conclusion:

Position, if understood, can be very powerful. One can use positioning to totally eliminate the need for nested tables which end up bloating pages with unnecessary mark up not to mention increasing page load time.

Lesson 3: Coding to your top 5 priorities

Having experienced something in recent times with respect to my work, I decided to throw this out to the forum.

Whenever you get a job from a client and they want everything including the kitchen sink, how do you approach it, especially when you are pressured to release something within a relatively short period of time? Do you offer to give them all they want (which more likely than not translates to working 25 hours a day, and snapping at anyone who as much as says hi to you, to get it done)? Do you resist? What do you do?

What I have learned is that its best to, just like in life, have a scale of preference. If I am faced with a tight deadline, I tend to ask the client what must absolutely go out on day one and rank them in decreasing order of importance. I then sit with the client and say, "hey, I would love to get all these done by launch date however I would hate for the quality of work to suffer for want of proper detail. Why don't we do this - I will tackle the first 5 (or whatever number I feel I can get done) on your list and over the following weeks, ease in the rest. If you insist, I will get all this done by the time you want but I would strongly advise that you go with my approach. Besides, the beauty of the net is unlike say MS where they come out with a CD and you have to wait for a year for an update, with the web, you can literally add new functionality to your site daily."

Again, from experience, there might be a little apprehension but I think when they see the maturity in your approach, 99% usually agree with me with little or no resistance.

Conclusion:
Bottom line, just like in life as I said earlier, it is good to prioritize your requirements. It is then and only then that the quality of your work will shine.

Lesson 4: DIV tags and Setting Heights

Many a times, I will see sites that have their contents "bleed" through their DIV tags, for those who use CSS. In particular, this problem tends to present itself in FF and not IE. There is a reason for this:

In FF, once you set the height of a div to say, 600px, that div will never be flexible in height which means that if content is more than the 600px height, it bleeds through that DIV. In IE on the other hand, the 600px is regarded as a minimum height and so if the content is more than the 600px, the DIV expands accordingly. However, if the content equates to a height of 595px for instance, the height of the DIV remains the same at 600px and you will notice the 5px (in this case) gap.

How to fix the problem as a cross-browser solution?
You could declare your style such that it looks like so:

```
div { min-height: 600px; height: auto !important; height: 600px; }
```

Please note:
When implementing this code snippet, all inline styles should actually be in an external style sheet in accordance with web standards.

The "!important" part is a FF hack and IE 6 and below will ignore this line and use the "min-height" and "height: 600px" lines. The min-height just means exactly what it says, it's a minimum height.

Caveat:
Note that the line with "height: 600px !important" comes before the regular line "height: 600px". This is on purpose. For the "!important" rule to work properly, you must declare it before its equivalent IE counterpart.

Conclusion: Generally speaking, it is best not to hard code heights when dealing with DIVs, letting them grow and shrink naturally. However, for those exceptions, using what has been discussed in this chapter should get you up and running in no time.

Lesson 5: Tables are so 1990s…

Nowadays, with most browsers supporting web standards and with more and more devices being internet-ready, it is essential that your websites display properly across the board. Unfortunately, this isn't the case, as many websites, even in the year 2008 are developed using table upon table upon table all of which also add to the weight of a page. Enter CSS and table-less design.

I will illustrate a simple example for converting a 2 column table into its CSS equivalent. Please note that for brevity, I will include all CSS styles inline. You should always have as much of your CSS/JavaScript code in external files as possible.

Let us look at the table:

```
<table border="0" width="600" cellpadding="0" cellspacing="0">
	<tr>
		<td width="300">bla</td>
		<td width="300">bla bla
			<table border="0" cellpadding="0" cellspacing="0">
				<tr>
					<td width="150">blab bla bla</td>
					<td width="150">bla bla bla bla</td>
				</tr>
			</table>
		</td>
	</tr>
</table>
```

The above is an oversimplified table but hopefully you get the point. It is basically a 2-column table with the right column containing another table that has its own 2 columns. How can this be accomplished in CSS? Please read on…

```
<div style="width: 600px;">
	<div style="width: 300px; float: left;">bla</div>
	<div style="width: 300px; float: left;">bla bla
		<div style="width: 150px; float: left;">blab la bla</div>
		<div style="width: 150px; float: left;">bla bla bla bla</div>
		<br style="clear: both;" />
	</div>
	<br style="clear: both;" />
</div>
```

Please note a few things:

1) Inline styles aside, the CSS version is much more compact and cleaner.

2) The use of the BR tag: I could have used a DIV tag but too many DIV tags constitute "divitis" (where developers abuse the use of the DIV tag). Also the "clear: both;" ensures that in FireFox, the floated DIVs remain within

the flow of the document (otherwise, they would “bleed” through the outer DIV).

3) The “float: left;” attributes enable the DIV tags, being block-level elements, to “sit” side-by-side.

4) The DIV with width of 600 serves as the container DIV, holding the floated DIVs together.

Please note:
When implementing this code snippet, all inline styles should actually be in an external style sheet in accordance with web standards.

And that is how you would convert a tabled to a table-less design.

Conclusion:

Contrary to popular belief, converting any site from tabled to table-less design is quite easy and not as intimidating as it may seem. All it takes is constant practice which as they say, makes perfect.

Lesson 6: Much Ado about DocTypes

As some of you might be aware, the very first line of code in your (X)HTML files should be the doctype. They come in many flavors. Here are a few:

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/**html4/strict.dtd**">

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/**xhtml1-strict.dtd**">

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/**xhtml1-transitional.dtd**">

Now, I won't bore you with details and would advise you check the w3c site and/or Google for more information. However, as an explanation, I will explain what the above 3 mean and why, in general, specifying a doctype is very important. Focus on the bold portions please:

#1 and #2 declare that the document will be validated against the strict interpretation of HTML and XHTML respectively. #3 basically declares that the document will be validated against a loose interpretation of XHTML which means that it will overlook things like using tables while still enforcing things like making sure all tags are opened and closed. For the most part, sites today are validated against #3 as not all browsers are web standards-compliant. You can also define your own doctype but that is beyond the scope of this post.

Why is specifying a doctype important?
Because it is important that, especially if you are using a web standards-compliant browser, the document gets "interpreted" properly. Where no doctype is specified, "funny" things an happen. This is what is known as the browser entering "quirks mode" which is another way of saying that the browser will fall back to some default mode. In IE for instance, quirks mode means that your document will default to being displayed as though you were using IE4, not something you would want of course.

Conclusion:
Always specify a doctype. #3 is preferred as it ensures the minimum required to guarantee that your sites, even without testing, will appear and function properly on a PC, PDA, Screen reader, etc. I have developed many sites over the years and taken for granted the importance of specifying a doctype only to appreciate it as many years later, not one of my sites has been found wanting on a variety of devices.

Lesson 7: To List or not to List…

In this lesson, we will look at using lists both for horizontal and vertical menus.
In general, using lists for menus is better than using tables as by its very nature, it represents a list of items and in the case of menus, it represents a list of menu items. Lists also work well with Search Engines as the Search Engines can easily identify text that is within list tags.

Whether it's horizontal or vertical, using lists for menus follows the same structural pattern. What actually sets one style apart from the other is in the CSS styling.

Vertical Menus:
Here is a typical declaration of a list:

```
<ul id="navoptions">
        <li><a href="#">Item A</a></li>
        <li><a href="#">Item B</a></li>
        <li><a href="#">Item C</a></li>
        <li><a href="#">Item D</a></li>
</ul>
```

The anchor tag determines where each link goes and the width of each anchor tag can be determined via CSS ("width: 15px" for example). Here are the styles that you could apply to the above:

```
1) ul#navoptions li a {
        padding: 5px 0 0 5px;
        height: 25px;
        background: #008000;
        color: white;
        font-weight: bold;
        text-decoration: none;
        border-bottom: 1px solid #000;
        display: block;
}

2) ul#navoptions li a:hover {
        background: #808000;
        color: #fff;
        text-decoration: none;
}

3) ul#navoptions {
        list-style-type: none;
        padding: 0;
        margin: 0;
        width: 300px;
}
```

Please note:

1) When implementing the code snippets in this lesson, all inline styles should actually be in an external style sheet, in accordance with web standards.

2) #1 sets up the attributes of each anchor tag

3) #2 sets up the attributes of each list item for when you hover over them

4) #3 sets up the attributes for the UL tag. You will notice the margin and padding are set to zero pixels. This is because in IE and FireFox, those attributes are interpreted differently. In fact, zeroing out the margins works for IE and zeroing out the padding works for FireFox. The “list-style-type” attribute is what removes the default bullets associated with lists.

Horizontal Menus:

Using the example above, all you would need do is override the styles above so that they now look like this:

```
ul#navoptions li a {
        padding: 5px 0 0 5px;
        height: 25px;
        background: #008000;
        color: white;
        font-weight: bold;
        text-decoration: none;
        border-left: 1px solid #000;
        width: 75px;
}

ul#navoptions li a:hover {
        background: #808000;
        color: #fff;
        text-decoration: none;
}

ul#navoptions li {
        display: block;
        float: left;
        text-align: center;
        padding: 0;
        margin: 0;
        color: #fff;
}

ul#navoptions {
        list-style-type: none;
        padding: 0;
        margin: 0;
        width: 300px;
}
```

Experiment with each menu option, modify the styles (widths, colors, etc) and note the changes in behavior. This concludes our lesson on lists.

Conclusion:
Navigational menus form perhaps the very foundation upon which a website is built. If you mess it up, your visitors are bound to leave. So in order for your users to have the best experience possible, including those with disabilities as well as search engines, it is best you move away from using tables to using lists.

Lesson 8: A Lesson in Website Optimization

So, I don't know about you all but I have been very frustrated lately when at a cyber cafe. I try to open many nigerian-based sites and it takes forever. So I have been motivated to come up with a tutorial, if you will, on things you could do to speed up your sites a little. For this "tutorial", I will be using the site: http://www.basepointconsultants.com/

Please note:
I am not deliberately picking on this site developed by one of us for fun-making or bashing purposes. It just happens to be the first site I found. Also, this isn't to say I am a saint though I am within 95% at the very least of practicing what I am about to preach.

Let the LEARNing begin (for more: http://www.websiteoptimization.com/services/analyze/):

1) There are 17 separate requests to the server, 15 of which are images. Needless to say, a navigation system using CSS rollovers (yes, you can set backgrounds with CSS rollovers) could have eliminated 12 of those images (that is 6 for regular state and 6 for hover state).

2) The largest image is almost 30K (http://www.basepointconsultants.com/home_r1_c1.jpg). I probably would have created a DIV with a black background, made the textual parts actual text, and for the blue gradient, used CSS to position it absolutely to the right. As for the "base point", I would have made that the background of the DIV.

3) There are 16 separate calls to a spacer.gif file. CSS margins/paddings could have taken care of this.

4) Separating the JS and CSS code into separate files would also help reduce the load time of the pages

5) Addendum to #4 above: Going for a table-less layout would have further reduced the load time on the pages.

Conclusion:
With internet bandwidth increasing by the day, it might be tempting to write code without bearing in mind that there is a minority out there who don't have access to such bandwidth. Though in the minority, such people might be the very people who need your website most so make sure you code for the lowest common denominator, based on findings in your web logs of course.

Lesson 9: CSS Resetting

How many times have you had to do some kind of CSS hack or conditional because whereas your code looked good in one browser, it was totally off, sometimes by an annoying few pixels, in another browser? I feel your pain so not to worry. Fortunately, there is a solution that is better than hacking your way through life. It's called CSS resetting.

What CSS resetting basically means is you "zero" out all XHTML elements so that by default, they start out the same. For example:

```
body,div,dl,dt,dd,ul,ol,li,h1,h2,h3,h4,h5,h6,pre,form,fieldset,input,
textarea,p,blockquote,th,td {
        margin:0;
        padding:0;
}
```

The above sets the margin and padding of the aforementioned elements to zero. Now that you are starting from a blank slate, you can then add back paddings and margins to specific elements via their "id"s. An example:

```
div#links {
        margin-left: 5px;
}
```

The aforementioned code resets the div with id of "links" such that its left margin is 5 pixels. Now, regardless of browser, the margin for this element will be the same and you don't have to worry about consistency.

Conclusion:
If you want to stop pulling your hair out each time you have to write a hack that might end up breaking in the future with the release of a newer browser, in my opinion, resetting your CSS styles is not only necessary, it's imperative.
Speaking of margins…

Lesson 10: TRouBLe TRouBLes those who TRouBLe TRouBLe

Before you start wondering if this is a lesson in tongue-twisting, please bear in mind that this one more lesson on an issue that plagues many developers.

TRBL in the CSS world is an acronym that stands for Top Bottom Right Left and applies to the attributes of an element's padding and margin. Typically, some people declare say, margins like so:

```
div {
        margin-left: 5px;
        margin-bottom: 4px;
}
```

…and so on. Rather than type four lines for each of the attributes, you could combine them into one line thus saving space and over time, saving bandwidth, like so:

```
div {
        margin: 4px 10px;
}
```

The above basically means **Top: 4px; Right: 10px; Bottom: 4px; Left: 10px**. Taking the first letter of each attribute, TRB and L, combine to form the acronym TRBL or "trouble" which is easier to remember.

Given the 2nd example above, the style could further be reduced to 2 attributes. Whenever the 1st and 3rd and/or the 2nd and 4th attributes are the same, you could combine them into 2: Margin: 4px 10px; this statement means, "top and bottom have 4px and left and right have 10px". This is only applicable under the stated conditions.

Conclusion:
This might seem like a lot of nit-picking but if you are dealing with multiple CSS files and/or a file that contains hundreds of lines of CSS code, you could be saving a lot of bandwidth.

As much as possible, always go for brevity of code.

Lesson 11: Coding to Web Standards

What does “coding to web standards” mean? Well, it means many things to many people. Generally speaking however, it deals with 3 areas:

1) You must specify the appropriate DTD.

2) Writing well-formed HTML (XHTML).

3) Separating as much layout (CSS) and logic (JavaScript) from the presentation as possible (via use of external files).
4) Using Semantic Mark up

5) Writing XHTML code that validates

Let’s take this one item at a time:

You must specify the appropriate DTD
This is required for a validator to know what to validate against. Please refer to the lesson on DTDs for an explanation of what each means, with respect to XHTML.

Writing your code using XHTML
Majority of the web browsers out there have been developed with web standards in mind. This means that if you write well-formed HTML (XHTML), it is guaranteed that your pages will display exactly alike across the board. Writing well-formed code means the following:

- Every opened tag must have a closed tag. Even if the tag is orphaned (like a BR tag, which would be written as <*br* />), it still has to be closed

- If you have a tag like this: <*input type=”text”* /> and you want to specify that it be disabled, in the past, you would write it like this: <*input type=”text” disabled* />. This however isn’t well-formed. The well-formed version would be written like so: <*input type=”text” disabled=”disabled”* />. Note that the input tag has is closed (“/>”).

- All tags must be written in lowercase.

Separating as much layout (CSS) and logic (JavaScript) from the presentation as possible (via use of external files)
This helps with respect to download times in general but especially on portable devices where air time is a concern. Unless you are writing code that is specific to a particular page, all styles and JavaScript code should be within CSS and JavaScript external files respectively.

Using Semantic Mark up
This basically means that your code should be written the way it would have been if you were say, verbally writing your code. This is where FireFox comes in handy. If you were to download the Web Developer Toolkit plug-in and turn off the styles, you would see what your pages look like for someone who either turns off styles, uses a text-based browser like Lynx or browses on a portable device. Here are some examples:

- Page headers should be represented as such (h1, h2, etc) and not as bolded P tags or sentences separated by BR tags. You can always style your header tags so that they don't conform to the defaults.

- A paragraph should be represented as a paragraph and not lines separated by BR tags

- Items that represented as lists should be represented as such and not lines separated by non-space break tags

- Only use tables to represent tabular data.

Writing XHTML code that validates
This also helps ensure that your pages display consistently across browsers. There are many XTHML validators including the one at the W3C website. Again, FireFox comes in handy here. There is a validator plug-in which gives you live information which is better than visiting another website. The plug-in also includes a CSS validator.

Conclusion:
Coding to web standards, though annoying and frustrating reaps big time dividends. The more you conform to them, the more they become second nature.

Lesson 12: CSS Rollovers vs. Image Rollovers

I have already delved into the sphere of CSS rollovers in lesson 5. Please refer to that lesson for more. This particular lesson is less hands-on and more of a discussion on why it is best that you dump image rollovers in favor of CSS rollovers.

Suppose you have a menu with 5 items and each item is actually an image. Initially, as the first page of your site loads, each image associated with each menu item has to be loaded separately. Depending on the size of each image, the site could load quickly or slowly. Not to mention, if you had a menu item named, “Boy” and you wanted to change the name to, “Boys”, you would have to open up your favorite graphics editor, make the change and then re-upload the new image. What’s worse? If you are no longer in charge of the site and never bothered to leave behind a style guide behind, the new designer might have issues when trying to figure out which font set you used or even finding whatever background you may have had.

With CSS on the other hand, making a change is a breeze. All you have to do is make a textual change in the CSS file and re-upload, that’s it! There is also little or no load on the server as generally speaking; CSS files are very light compared to their image counterparts. Adding a background to your menu items is also not a bid deal in CSS. Finally, search engines, which love chewing up text, love CSS menus.

Conclusion:
If you haven’t used CSS Menus/Rollovers on your projects, it is about time you did. You will be surprised as to how much time, effort and perhaps most important of all, bandwidth, that you save in the process.

Lesson 13: Website Architecture

From childhood, we have constantly been told that the story of the 3 pigs clearly teaches that a house built on any foundation that isn't solid is bound to collapse. Websites and applications aren't immune from that either. Creating a solid foundation from scratch will help make your life as a developer easier as well as any developer that comes in after you to make changes. Website architecture not only covers the front end but the back end as well.

There are many factors that come in to play when it comes to building a solid foundation for your website. Listing them all is outside the scope of this EBook but the following should suffice as they are the most common that I have dealt with in my career:

Use of Server-side includes and functions whenever possible:
Whenever you use some piece of code that is or will be repeated more than once, it's best to use a server-side include for this. That way, future changes are made just once. Also, if you find yourself writing code that should go into a function, by all means, make that code a function. Same reasons apply.
When I am in the process of setting up my layouts, I have the following include files at the very least: header, footer, and navigation.

Page width:
Most users surf the net on a 1024x768 screen. To that effect, don't go for the maximum resolution. Most developers go with Yahoo!'s recommended width of 974px. That way, you can center your content area and still have room to play with.

Table-less design:
Using CSS and XHTML, you can avoid the clutter and confusion that results from nesting table upon table upon table. Table-less design makes your code more readable and easier to maintain.

Use External CSS and JavaScript files:
Unless you absolutely have to, using inline styles and code only leads to possible repetition which makes maintenance and pain.

Database interaction abstraction:
In the event that your site will be hitting a database for whatever reason and multiple times at that, rather than doing the same calls to the database, create a function that returns a Boolean value. Thus all you are doing is, "if(true)…connect…else print error", a 1-line statement as opposed to typing out the same 5 or so lines of code in different places. What's more? Given #1 above, should a change in the password for example need to be made, creating a function ensures that you make the change just once.

Indent your code:
This one is self-explanatory. I can't count how many times I have had to spend minutes just trying to follow some line of spaghetti code before finally deciding to indent the code for myself beginning with line 1. It is only after I finish that I then begin to focus on the issue many a times which amounts to 5 minutes or less of work. Think of all that time I could have saved if only the code were indented. Separate back end code from front end code: This comes in very handy especially where the 2 developer roles are separate. If both logic and front end code are on the same level, it could overwhelm whatever party goes in to make changes. Here is a good example (though a simplified one) of multi-level indentation using in this case, PHP:

```
<?php if(condition) { ?>
        <p>Say hi</p>
<?php } else { ?>
        <p> Say bye</p>
<?php } ?>
```

Know what infrastructure works best for your site:
How many times have you visited a newspaper site, for instance, and noticed that all the pages are HTML pages. If you have encountered such a site and that site is being updated daily, I would say that is an exception.

Basically what I am saying here is you shouldn't take the "Hammer and Nail" approach of "if all I have is a hammer, everything will be a nail to me" because you *will* fail. Take the newspaper site for instance. Since the site will be updated daily at the very least, it would be better for such a site to have a content management system behind it that manages everything from updates to archiving etc. Likewise, if all you have to do is create a 5-page informational site for a client, a content management system would be overkill for this project.

Conclusion:
A majority of your time in development should be spent planning out your infrastructure. I would say as much as 80%. It might sound like much but trust me, it is better than spending 80% of your time trying to debug your site/application.

Lesson 14: Are you in LoVe? HA!

Before your mind starts wondering too far, this actually has to do with website design and development. As we all know, it is the concept of linking that has ensured the success of the web; the ability to go from the home page to the “thank you for ordering” page that has turned many into instant millionaires. Thus many have spent millions paying people to determine where to place links, how they should look and what types of links work best where.

There are times, if you are like me, where in implementing some cool linking technique, you notice that your links aren’t functioning properly: they don’t highlight/underline when you hover over them, visited links don’t change to the color you specified or maybe your links show in bold font when they are not supposed to. The possibilities are endless here. From experience, this is usually because not all developers know that the order in which they specify their anchor tags from within CSS matters.

To get straight to the point, when specifying styles, you *must* follow the following rule: **a:link**, **a:visited**, **a:hover** and **a:active**. If you capitalize the first letter of each property, you come up with **LVHA** or “**Love HA**”, which is my way of spelling it for easy recall. Even if you are not specifying all of them, you still must follow the rules. So if, for example, you want to specify all but the “hover” state, you would specify the styles as “**LVA**”. In the future, should you want to add the “hover” state, it would need to go in the right spot, right after the “**V**”.

Conclusion:
You must follow the rules for your code to work. Unfortunately in this case, there is no exception.

Lesson 15: Testing/Quality Assurance

Last but not least, I will be talking about an issue which though ignored, is perhaps as important as coding itself.

How many times have you visited a website and clicked around for instance only to find broken links, broken images, pages that take forever to load or links that don't do anything, to mention but a few? How did that make you feel? Did you ever go back to those sites? Well, if you are like me, you end up frustrated, and chances are that unless it is a site you visit regularly, you will never visit that site again. Especially with respect to African sites, it seems we aren't doing enough in the area of testing and quality assurance. In the US for instance, there are Testers earning 6-figure incomes just from testing websites. Don't get me wrong. Especially on large-scale applications, testing isn't as simple as clicking from A to Z. It usually involves the running of test scripts. Regardless of approach, there should always be some kind of testing done on a site prior to release.

When I am implementing a site, I usually go for the unit-testing approach: test each page for links, functionality, layout, etc and make sure that each page passes my test before moving on to the next page. Once I am done with all my pages, I go through another round of testing (integration testing) just to make sure I didn't miss something. On projects where a client hands me requirements, I usually write up a test plan based on those requirements and execute the outlined steps after I have conducted my own personal tests. For me, it is an embarrassment for a user to point out an error, no matter how trivial, to me.

On the issue of testing, many will argue saying, "Hey! Even Microsoft isn't above mistakes. After all, we end up finding their mistakes for them for free". Very true but 2 wrongs don't make a right. What is good for the goose isn't always good for the gander.

Conclusion:
Bottom line – spend the time upfront to test your pages and give your users the best experience possible. Otherwise, get ready to save face and risk losing your well earned reputation when something goes wrong.

APPENDIX

- **Webmasters of Africa (WoA) Website:**
  http://www.webmastersofafrica.com

- **WoA Discussion Forum:**
  http://www.webmastersofafrica.com/vanilla

- **WoA Resources:**
  http://www.webmastersofafrica.com/webmasters-of-africa-resources/

- **WoA Membership Form:**
  http://www.webmastersofafrica.com/become-webmaster-of-africa-member/

- **How to Design a Web 2.0 Website:**
  http://www.webdesignfromscratch.com/web-2.0-design-style-guide.cfm