# Technical Style Standards

## Version 1.0d

Edition 1

| | |
|---|---|
| **Document ID:** | TSS |
| **Author(s):** | Tristan Austin<br>Jaycee Phua |
| **Reviewed by:** | Jaycee Phua<br>Simon Hutchison<br>Tristan Austin |
| **Date:** | 11 May 1999 |

---

# Revision History

| Date | Modifications | Reason | Version |
|---|---|---|---|
| 1999.03.22 | Document adapted to HTML format (JP) | Submission requirement | 0.3d |
| 1999.03.28 | Document contents formatted (JP) | Conformance to latest Style specification | 0.35d |
| 1999.03.29 | Section added: Technical Document Standards (JP) | Extend on document's purpose | 0.4d |
| 1999.03.31 | Reviewer added to Cover Page (JP) | Quality Assurance content check (SH) | 0.41d |
| 1999.04.11 | Coding Standards significantly revised (TA) | QA Reviewer suggestions (SH) | 0.5d |
| 1999.04.29 | HTML Source and content organisation revised (JP) | Conformance to latest Style specification | 0.6d |
| 1999.04.30 | Reviewer added to Cover Page - TA (JP) | Author's (TA) content reviewed | 0.51d |
| 1999.04.30 | Coding Standards grammar and formatting revised (JP) | Reviewer (Author - TA) suggestions | 0.85d |
| 1999.05.03 | Java Coding Standards content revised and appended (TA) | Author's content re-verified and validated | 0.9d |
| 1999.05.05 | Updated content formatted and style-verified (JP) | Adherence to Style Specification | 0.91d |
| 1999.05.11 | HTML Coding Style section revised & appended (JP) | Section previously incomplete | 0.95d |
| 1999.05.11 | Technical Document Standards section removed (JP) | not considered appropriate TSS content (QA) | 1.0d |
| 1999.05.14 | Certain Java code example comments changed(JP) | Web Browser's tendency to word wrap accounted for. | 1.0d |

# Table Of Contents

# 1. Scope

This document outlines the Technical Style Standards that are to be followed concerning all **Team Synergy** projects. All coding, technical and/or instructional content created for any **Team Synergy** related project must strictly adhere to the details contained within this document specification.

Sections of this document not understood by any persons affiliated with **Team Synergy** should approach the appropriate Style Standards Coordinator for further clarification (through appropriate channels).

Topics covered within this document are primarily relevant to **Team Synergy** *Application Engineers* and those formally known as *Technical Writers* (now members of the *Publishing Group*). *Quality Engineers* should additionally take note of the content, to ensure that documents based on the principles stated within this specification are of the highest standards.

In terms of Programming Languages, this edition of the ***Technical Style Standards*** has been produced with the *Java Programming Language* in mind. It is thus natural to expect instances where *Computer Output* is described. Consequently, for practicality reasons, `monospaced` text within this document signifies *Computer Output*; text coloured in Green will signify *Java Source Code*. Additionally, *HTML Code* will be referenced in Blue.

# 2. Java Coding Style

All the source code produced for any **Team Synergy** product will conform to the style as outlined in this document. By defining class and variable naming conventions, as well as other standards, third party developers attempting to make sense of various modules will be able to understand the disparate components.

This will encourage a more harmonic integration between modules produced by various (relatively independent) developers and make for an easy to read and maintain code base.

## *2.1 White Spacing*

White space is a very important facet of source code. By making sure all code is 'spaced accordingly'. That is to say, all elements of the code syntax should be separated by a single space. This includes items which aren't necessarily required by the compiler.

**eg.**

```
for (int i=0; i<vector.size();i++)
:
```

***incorrect***

---

```
for (int i = 0; i < vector.size(); i++)
:
```

**correct**

Indentation is four characters wide and spaces are to be used as opposed to tabs.

Most *Integrated Development Environments* (IDEs') should support translating tabs to spaces.

## 2.2 Braces

*Braces* ('{','}') are used as a compiler requirement but also serve as an indication of the scope of a block of code. In cases where the compiler does not require the use of braces, we will enforce their use in order to maintain consistency.

Each brace must appear on the line following the statement it relates to and each statement must be on a line of its own.

**eg.**

```
if (object.isTrue())
    object.execute();
```

***incorrect***

---

```
if (object.isTrue()) {
    object.execute();
}
else {
    object.stop();
}
```

***incorrect***

---

```
if (object.isTrue())
{
    object.execute();
}
else
{
    object.stop();
}
```

**correct**

## 2.3 Indentation

Indentation complements the use of braces to indicate scope. Everything within a set of braces needs to be one tab to the right of the alignment of the braces.

**eg.**

```
for (int i = 0; i < vector.size(); i++)
{
    object = vector.elementAt(i);

    if (object.isTrue())
    {
        object.execute();
    }
    else
    {
        object.stop();
    }
}
```

## *2.4 Commenting*

Commenting is very important throughout the code so that no one becomes essential to the maintenance or further development of a module or block of code. Single line comments need to precede all major tasks in the code and anything that, by the developers discretion, may be difficult to understand.

All classes, methods and identifiers must have a multi line comment preceding them with the following tags included (where appropriate), along with a detailed description of what exactly the role of the class, method or identifer is:

| | |
|---|---|
| `@param` | The parameters to the method |
| `@return` | What the method returns |
| `@throws` | The exceptions it throws |
| `@see` | Relevant classes to see |
| `@bugs` | Any known bugs |
| `@author` | The author of the class |
| `@pre` | Any pre conditions of the class |
| `@post` | Any post conditions of the class |
| `@modifies` | Indicates if it modifies the class |
| `@copyright` | The copy right message |
| `@concurrency` | Any concurrency issues |
| `@rref` | Requirement/Analysis/Design |

This is very important as it defines what is produced when we run javadoc over it to create our source documentation.

When you declare a new method, you should write the comments before you write the method. Do not write slabs of code and then go back through and add your comments. Make sure both why and what the method, class etc. is included.

Javadoc looks for multi line comments starting with /** and ending with */ So avoid using the former within code. If this is required use a single star for the starting tag (ie. /* comments */). This should be avoided, you should just use single line comments when describing a code block. If the comment needs to span multiple lines, use multiple single line comments.

If there are any synchronization issues regarding the member, these must be identified in the comments. For example, if a daemon thread is continuously working with a given variable, any issues regarding setting or using that variable need to be identified.

When documenting the contents of a method, you need to provide descriptions of each local variable, all control structures, such as ifs and switches, and the end of a large block of code, such as a for loop with more than ten lines.

**eg.**

```
/**
 * A FourLeggedTable is a
 * table which has four legs and
 * as such behaves accordingly.
 *
 * @author Foo Bar
 */
public class FourLeggedTable extends Table
{
  /**
     * The color of the table
     */
    private Color color = Color.Brown;

  /**
     * Indicates whether this table is the color specified.
     *
     * @param  color  The color to check for
     * @return True if valid, otherwise false
     * @throws NullPointerException
     */
    public boolean isColor(Color color) throws NullPointerException
    {
        return this.color == color;
    }
}
```

## *2.5 Naming Conventions*

All names of methods, variables, and classes need to be named in a manner which makes their purpose obvious. The names should be kept to a realistic length, so avoid using names in excess of 20 characters.

When naming an item, you need to make every subsequent word begin with a capital letter so that it is easy to read. The case and value of the starting letter is dependent up on the type of item you are naming, you'll need to refer to the appropriate section for more detail on each type.

When naming components for the interface, you should append the name of the component to the end of the descriptive name so that it is obvious what it is. You need not specify the entire component name, just enough to describe what it is:

**eg.**
```
okButton
userList
serverComboBox
```

You as a general rule, you should also avoid using similarly named variables, such as differing only in case, to avoid confusion about which does what.

Names should not contain any underscores at all.

## 2.6 Constants

Constants need to be defined as 'final' variables and where possible as 'static'. They will begin with a lower case 'k' followed by a meaningful name with each subsequent word starting with an upper case letter as with variables.

**eg.**
```
private static int kNumberOfLegs = 4;
```

**Note:** Identifier names should not contain numbers, unless the number is at the end of the Identifier name:

**eg.**
```
private int table2 = 2;
```

## 2.7 String Literals

The use of String literals in code is to be avoided at all times. This also goes for numeric values as well. Any reference to a String or values in the code must be via a variable, or constant declaration.

**eg.**
```
system.out.println("File not
found");
```
***incorrect***

---

```
string noFileMsg = "File not found"
system.out.println( noFileMsg );
```
**correct**

## 2.8 Switch Statements

Every switch statement needs to have a default clause at the end. As a general rule, each case in the switch statement shold be terminated by a "break" statement, however some situations require no break statements between cases, this is fine so long as it is required.

Each statement between the case statements need to be enclosed in braces in the following form:

**eg.**
```
switch(size)
{
    case kHuge:
    {
        Debug.doPrintln("Is huge and ");
    }
    case kBig:
    {
        Debug.doPrintln("Is large");
        break;
    }
    default:
    {
        Debug.doPrintln("Type: " + size + " not supported");
        break;
    }
}
```

## *2.9 Classes*

Classes should be named using a noun that represents a purpose. For example, to represent a table, you should have a class 'Table' from which you instantiate a 'diningTable'. Class names will begin with an upper case letter. For multi-word classes, each subsequent word should start with a capital letter.

**eg.** `FourLeggedTable`

There should be no anonymous classes and any classes which constitute an addition to an existing class, such as an ActionListener, should be defined as an inner class. Aside from these inner classes, there should only be one class per ".java" file.

All class names will begin with 'B' followed by the same notation as already stated, to make the distinction between our classes and those from elsewhere.

**eg.** `BClassName`

When referring to static variables or constants, the identifier must be prefixed with the name of the class it is in, even if it is referred to from within the class itself.

Each Class should be accompanied by a comment, stating the date the class is created, along with any modification(s) made to the class; including the date, description and author of the modification. The copyright information must also be included in each source file. This will be a constant message which does not change from file to file.

**eg.**

```
/**
 * This class is just an example of what you need to supply.
 *
 * Created on 1/2/1999
 * Modified on 14/03/1999 by Mike Smith - Added printing functionality

 *
 * This class is the property of Team Synergy, its use is
 * governed by the licence, etc.
 *
 * @author John Doe
 * @version 1.1
 */
public class Foo
{

  /**
     * The constant value for Bar
     */
    public static final String kBar = "Bar"

  /**
     * Prints out the Bar message
     */
    public void printBar()
    {
        System.out.println(Foo.kBar);
    }
}
```

## 2.10 Interfaces

As with classes, Interfaces need to have their name beginning with a capital 'B' to indicate that it is one of ours but also an 'I' to indicate that it is an interface.

**eg.** `BIEventListener`

It also needs to be commented well enough to indicate what it is to be used for and how it should be used to ensure consistant use throughout the system.

## 2.11 Packages

The package names need to conform to a consistent naming convention. All packages created should represent a logically grouped collection of classes and fall under the following high level packages:

- `cohesion.client`

- `cohesion.common`
- `cohesion.network`
- `cohesion.server`

When importing packages, packages from a similar source should be grouped together, ideally, the sources should consist of those in the java or javax name space and our own in the cohesion name space.

**eg.**
```
import java.io.*;
import javax.swing.*;

import cohesion.client.*;
import cohesion.common.*;
```

Packages should be documented with the justification of why this package exists and what it contains. As such, all classes must belong to a package, nothing should be in the "default package".

## *2.12 Members*

Members of a class need to be declared in the following order.

1. Variables
   - *`public`*
   - *`protected`*
   - *`default`*
   - *`private`*
2. Constructor
   - *`public`*
   - *`protected`*
   - *`default`*
   - *`private`*
3. Methods
   - *`public`*
   - *`protected`*
   - *`default`*
   - *`private`*
4. Inner classes

Methods and variables relating to common functions should be grouped together. For example, all *Get/Set* combinations of methods should be grouped together in the following manner:

- `setState(boolean newState)`
- `getState()`

- `setUp(boolean up)`
- `getUp()`

As in the example above, *Get/Set* combinations should be in order of *Set* then *Get* for each pair. Similarly, all *Add* and *Remove* combinations should be grouped together-- *Add* first, followed by *Remove*.

In general the class should be set out in a manner resembling the following:

- Constructors
  - `public Constructor(); //Default constructor`
  - `public Constructor(boolean initialState); //Specific constructor`
- Initialisation Methods used by Constructors
  - `public void doUserInitialisation(); //May be used by constructor`
- Setup Methods which may be called after creating the Object
  - `public void setUpUserManager(); //Tasks to be completed soon after construction`
  - `public boolean isInitialised(); //make sure the class has been set up properly`
- Get/Set Methods
  - `public void setState(boolean newState);`
  - `public boolean getState();`
  - `public void setSize(Size size);`
  - `public Size getSize();`
- Services/Functionality the Class provides
  - `public void doFunctionality();`
  - `public void doVaporware();`
  - `public void doRender();`
- Querying Methods
  - `public boolean isOpen();`
  - `public boolean isTrue();`
- Methods called when a particular Event is fired
  - `public void onFooEvent(FooEvent fooEvent);`
  - `public void onBarEvent(BarEvent barEvent);`
- Adding Listeners to the Class
  - `public void addNetEventListener(NetEventListener newListener);`
  - `public void removeNetEventListener(NetEventListener oldListener);`
  - `public void addActionListener(ActionListener newListener);`
  - `public void removeActionListener(ActionListener oldListener);`
- Inherited Methods
  - `public String toString();`
  - `public Object clone();`
- Converting Methods
  - `public int toInt( object o );`
  - `public BUser toUser( object o );`

## *2.13 Variables*

All variables should be named with their purpose in mind. Referring to the example above, the name 'diningTable' is sufficient to distinguish between it and any other table that may exist.

There are to be no single letter or unmeaningful variable names like such as 'cb' or 't'. The exception to this rule is single letter loop counters which should start at 'i' and for each additional loop counter in the same scope, the identifier should be the following letter (ie. 'j', then 'k').

The first letter of a variable should start with a lower case letter and all subsequent words in the name should begin with a capital.

**eg.** `diningRoomTable`

There should only be one variable declaration declaration per line.

**eg.**
```
int i, j, k = 0;
```
***incorrect***

---

```
int i = 0; // First loop counter
int j = 0; // Second loop counter
int k = 0; // Third loop counter
```
***correct***

## *2.14 Methods*

All public methods should be prefixed with one of the following:

- `get` Does not modify the state of the class
- `set` Modifies the state of the class
- `getCopy` Copy a parameter rather than returning a reference
- `setCopy` Sets a copy of the given property rather than a reference to it
- `to` Converts the given object to another type as specified
- `is` Does not modify the state of the class
- `on` On some event, do this
- `do` Perform some operation
- `add` Adds an object from the given object
- `remove` Removes an object from the current object

Private methods may have names beginning with other prefixes, however they must give a good indication of the purpose of the method.

**eg.** `calculateValue(int value1, int value2)`

Methods with no bodies can be used quite often when implementing an interface of which you don't need to provide functionalitory for all methods. In this case, these "null" methods should be written in the following manner:

**eg.**
```
/**
 * When the user clicks the mouse, this method is called
 *
 * @param  mouseEvent  The event fired
 */
public void mouseClicked(MouseEvent mouseEvent)
{/*Not Required*/}
```

The braces indicating the scope of the method should be on a single line and contain the comment, "Not Required" in block quotes.

When a method, such as a set, or do method, changes the state of the class it is operating on, this fact needs to be documented in the method comments detailing what exactly it changes. The pre and post conditions of a method must also be included using the tags @pre and @post which will be implemented in our own doclet.

## 2.15 Inner Classes

Inner classes should be treated in the exact same manner as normal classes, however they should be declared as the last "members" of the class, after all variables and method declarations.

Inner classes should be used to represent an abstraction within an already existing class. If other classes use this functionality, it should not be an inner class at all.

Inner classes are primarily used for event listeners within a given class. Primarily their role is to redirect program flow to the appropriate methods in its parent class.

See the next section (below) for an example.

## 2.16 Entire Class Sample

The following is an example of an entire class incorporating all of the requirements as pointed out in this document (note-- the first line, highlighted in purple, should be specified because it is used in conjunction with the *Concurrent Versioning System* (CVS). Please refer to the acting Configuration Manager the for further information regarding CVS.

```
//$Header:
/teamproj/cvsroot/teamb/docs/documentation/tss/index-content.html,v 1.3
1999/09/10 07:45:45 tp079554 Exp $

package cohesion.common;

import java.net.*;
import java.awt.*;
import java awt.event.*;

import cohesion.network.*;

/**
 * A BUser provides the details of a particular user
```

```
 * including their name, their location (ip address etc.)
 * and other details associated with each user.
 *
 * @author Tristan Austin
 */
public class BUser
{
  /**
     * The username of the user
     */
    private String userName = null;

  /**
     * The password of the user
     */
    private String password = null;

  /**
     * The add button
     */
    private Button editButton = new Button("Edit");

  /**
     * The listener for the button events
     */
    private BButtonListener buttonListener = new BButtonListener();

  /**
     * Creates a user object with the given user name and
     * the ip address of the client machine
     */
    public BUser(String userName)
    {
        this.userName = userName;
        this.editButton.addActionListener(this.buttonListener);
    }

  /**
     * Enables the setting to password.
     *
     * @param  password  The unencrypted password of the user
     */
    public void setPassword(String password)
    {
        this.password = password;
    }

  /**
     * Gets this user's password
```

```
    *
    * @param  Gets the unecrypted password
    */
  public String getPassword()
  {
      return this.password;
  }

/**
    * Gets the user name of this user
    *
    * @returns The user name
    */
  public String getUserName()
  {
      return this.userName;
  }

/**
    * Gets the user name of this user
    *
    * @returns The user name
    */
  private void analyseType(int type)
  {
      switch(type)
      {
          case kHuge:
          {
              Debug.doPrintln("Is overweight and ");
          }
          case kLarge:
          {
              Debug.doPrintln("is large");
              break;
          }
          default:
          {
              Debug.doPrintln("Type: " + type + " is not supported");
              break;
          }
      }
  }

/**
    * When the edit button is clicked, this method is called.
    *
    * @param  actionEvent  The event fired
    */
```

```
    private void editButtonClicked(ActionEvent actionEvent)
    {
        Debug.doPrintln("Edit button pressed");
    }

  /**
    * This class handles program flow when a user clicks on a button
    * an instance of a button listener is registered with.
    *
    * @author Foo Bar
    */
    public class BButtonListener extends ActionAdapater
    {
      /**
        * When a button is clicked, this method is called.  It simply
        * redirects the program flow to the appropriate method in
        * the main class.
        */
       public void actionPerformed(ActionEvent actionEvent)
       {
           Object source = actionEvent.getSource();

         // Determine which button was clicked
           if (source == BUser.this.editButton)
           {

             // Send the event to the appropriate method
               BUser.this.editButtonClicked(actionEvent);
           }
       }
    }
}
```

# 3. HTML Coding Style

This section outlines certain formatting-related HTML specifics that should be observed when compiling Web-oriented documents. Not all HTML specifics will be covered, as in most cases it is possible to compile an HTML document without even dealing with the code itself. However, it is advised that the instructions outlined in this section are followed, so that all **Team Synergy** documents coded in HTML are:

- easy to read
- easy to re-use
- easy to modify.

In order to read this section, a certain degree of HTML knowledge is assumed to be understood. Explanations will be kept to a minimum, and will only be included where relevant.

**Note:** This section does *not* cover the formatting requirements of a document, nor does it include what should and should not be included as part of **Team Synergy** documentation. This section is only concerned with the formatting of the underlying code that is the basis for the content of an Web-based document. Refer to the ***Web Document Style Standards*** to understand "Document Style"; as opposed to this section, which covers "Document-Code Style"

## *3.1 HTML Tags*

*Tags* are specified with the *Less-Than* (<) and *Greater-Than* (>) operators. These operators will henceforth be signified by the title *Tag Specifiers*. Text affected by Tags shall be known as *Tagged Text* from this point on. Text to be defined within a Tag and its terminator would be within its *Tag Domain*.

Primarily the 3 properties that 'decide' a Tag is the *Tag Name*, *Tag Attribute*, and *Tag Value*.

All Tags (and those contents within those tags) should be entirely in *Lower Case*. Absolutely all of each Tag's Value(s) should be stated in *Double Quotes* (*"*).

Thus, Tags consist of the following properties:

**HTML:** `<font size="2">something</font>`

| | |
|---|---|
| Tag Specifiers | `< >` |
| Tag Name | `font` |
| Tag Attribute | `size=` |
| Tag Value | `"2"` |
| Tagged Text | `something` |
| Tag Terminator | `/font` |

Tag Terminators are not required in some cases. It shall be discussed in the following section.

### 3.1.1 Tag Terminators

Current HTML Coding Standards specify that certain Tag Terminators are optional; not compulsory. **Team Synergy** Style suggests that some of these Tag Terminators with an 'optional' status will not be ignored-- they must be explicitly stated no matter what.

From this point on, each of the following sections shall identify those Tags which have accompanying Terminators that must be included, or may be ignored.

Take the following example, which describes an explicit 'Compulsory' Tag Terminator:

**HTML:**

```
<!-- This is some commented HTML code.  If this comment were not
explicitly terminated, the rest of this document would be considered
part of the comment. -->
<html>
:

:
</html>
```

Within this example, there are 2 Tags with compulsory Terminators. Not specifying the Terminators in these cases would result in havoc when viewing the code.

## 3.2 Code Line Length

HTML code should be limited to *around* 80 characters to a line; this minimises the need to excessively scroll when viewing/modifying code.

If it is possible to follow-on to the next line, then do so. In most cases, it is more of a priority to keep within this recommended line-limit, than to abide by the other rules imposed within this ***HTML Coding Style***. Priorities for the use of these HTML-based rules is provided in a later section.

In extreme cases where this is not possible, exceeding the limit is acceptable. A possible situation where the limit cannot possibly be adhered to is when an "Anchor" tag is concerned:

**HTML:**

```
....v....1....v....2....v....3....v....4....v....5....v....6....v....7....v....8
<a href="http://www.this_is_an_example.com.au/directory/subdirectory/this_is_an_example.html">
This is an Example</a>
```

Tag Contents (that is, those components between Tag Specifiers) should not be split on to separate lines. The exception would be for Table and Images Tags, which sometimes require multiple attributes that cannot be specified on the same line.

The 'cheapest' method to sticking as close to the character limit (as opposed to purchasing an HTML-editor that does it all for you), is by resizing the window of the code-editing program to 80 (monospaced) characters.

One can use the sample of numbered text (above), to calculate the preferred width of an editing window-- copy the text into the editor, and then resize the window corresponding to the space that the copied text takes up.

**eg.**

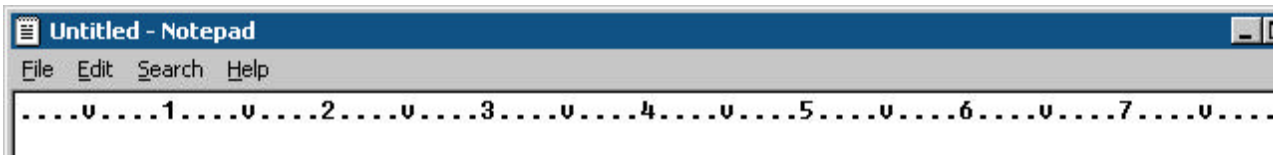


**Figure 3.3 - 1:** Resized Window displaying only 80 Characters per line

## 3.3 Indentation

Indentation is necessary for HTML code, so that a certain portion of code can easily be identified within the document.

Understanding where indentation should be performed is directly related to the use of *Tags*-- knowing when and where tags are applied permits us to say that certain tags have more precedence over others.

In terms of the Style specified here for coding **Team Synergy** documentation, those tags which are considered more significant decide the indentation that should be performed on the code.

### 3.3.1 Indent Size

The *Indent Size*, or "Tab Width" should be 2 characters. If Tab/Indent facilities are unavailable with the editor in use, implement indentation using 2 character spaces.

### 3.3.2 Paragraph Tags

*Paragraph Tags* should be used very sparingly to ensure ease in readability. All Paragraph Tags should be specified on a new line. Additionally, all content within Paragraph Tags should be indented to emphasise the Paragraph Tag's domain.

Where possible, each section within a document should only contain one HTML Paragraph Tag set (" <P> ", " </P>"). When line breaks between paragraphs are necessary, two HTML *Line Breaks* (" <br><br> ") should be implemented in place of Paragraph Tags. The exception to this rule is when a section of HTML Text is separated by a "*Non-Paragraph-Text*" object.

**eg.** **HTML:**

```
  :
  This is the last sentence in the first paragraph.
  <br><br>
  This is the second paragraph.  Note the line break
  being used between the first and second paragraphs.
</p>

<img src="sample.gif">    <-- Non-Paragraph-Text Object

<p>This is the third paragraph.  This paragraph cannot
  belong with the first two paragraphs because a
  Non-Paragraph-Text object is 'obstructing' it.
  <ul>
  <li>This is a list item.
  <li>This is also a list item.
  </ul>
  Note that an ordered/unordered list can be included.
</p>
```

In other words, there are some 'objects' that are allowed within a Paragraph Tag domain, and others that should not be. In the context of the ***Document Content Template***, the sorts of objects that are not recommended within Paragraph Tags include the following:

- Note Box
- Example Text (Box)
- Image
- Table

Those objects stated above should not be included within a Paragraph Tag Bulleted and Numerical (Unordered and Ordered) Lists are among those 'objects' that can be included within a Paragraph Tag domain.

The best way to understand the procedures described in this section is to examine the HTML Source within this ***Technical Style Standards***. Alternatively, referring to the HTML Source contained in the ***Document Content Template*** will also provide a basic outline to what is described here.

## Font Tags

A Font Tag must be terminated before another Font Tag is allowed to begin. There should not be any overlapping of such tags. However, other tags may be applied within Font Tags, and Font Tags may also be applied within other Tags-- the context depends on whether the font style should affect several words, or a section of text.

Sections of text within the body of an HTML document should *not* be left without being defined with a Font Tag.

## Formatting Tags

**HTML:**

```
<b>This text is set to 'Bold'</b>
<i>This text is set to 'Italics'</i>
<u>This text is set to 'Underline'</u>
```

**This text is set to 'Bold'**
*This text is set to 'Italics'*
This text is set to 'Underline'

Ensure that Tagged Text does not begin or end with spaces.

**eg.** `this is <b><i>some </i></b>text.` ***incorrect***

---

`this is <b><i>some</i></b> text.` **correct**

Like Font Tags, a Formatting Tag is permitted to be within another Tag, or other Tags may be within a Formatting Tag, depending on whether that portion of text requires formatting or not.

## Bulleted & Numbered Lists

The equivalent names for "Bulleted Lists" and "Numbered Lists" in HTML-tag convention would be *Unordered Lists* and *Ordered Lists* respectively.

List Items *may* be terminated, however it is perceived by the *HTML Style Coordinator* to be unnecessary.

**HTML:**

```
<ul type="disc">
  <li>list 1</li>
  <li>list 2</li>
  <ul type="circle">
    <li list 2.1</li>
  </ul>
</ul>
```

- list 1
- list 2
    - list 2.1

The following HTML example is a second possible method for displaying lists; use of this method is not condoned nor will it be tolerated:

**HTML:**

```
<dir>
  <li>list 1</li>
  <li>list 2</li>
  <dir>
    <li>list 2.1</li>
  </dir>
</dir>
```

- list 1
- list 2
    - list 2.1

## Anchor Tags

**HTML:**

```
<a href="http://www.example.com/html/example.html">This is an Example</a>
<a href="example.html#SpecificSection">Specific Section</a>
<a name="SpecificSection"></a>
```

*Anchor Tags* should be used when identifying links to either another area of the same document, or another document altogether. An anchor can either be a Link Source or a Link Destination.

Link Destinations should not contain any Tagged Text within the Tag domain.

All anchor tags, in conjunction with its tagged text, should remain on the same line.

### 3.3.3 Tables

Table Tags should appropriately be indented and specified on a new line. All successive table-related Tags should also be indented.

Table attributes are recommended to remain on the same line. However it is anticipated there may be many instances where they span multiple lines in order to remain within the also recommended 80-character line limit. Attributes that span the next line should also be indented.

**HTML:**

```
<table width="100%" border="0" cellspacing="0" cellpadding="5"
    bgcolor="#c0c0c0" bordercolor="#c0c0c0">
  <tr colspan="2" valign="top" align="left">
    <th width="50">
    <font face="Arial, Helvetica, Sans-Serif" size="2">
      Heading
    </font>
  </th>
  <td width="95%">
    <font face="Arial, Helvetica, Sans-Serif" size="2">
      Contents
    </font>
  </td>
  </tr>
</table>
<br>
```

Indentation for Table Tags are especially important, which greatly aids in modifying the table contents when necessary. For the above example, inserting a new row into the table only involves duplicating the section signified by the <tr> Tag and corresponding Terminator.

## *3.4 Spacing*

Spacing is necessary for HTML code, so that a certain portion of code can easily be identified within the document. This assists in readability and reusability.

It is recommended that this document's source code is taken as an example, which is based on the ***Document Content Template***. This template has been appropriately spaced, so that portions of the code can easily be distinguished, and re-used.

### 3.4.1 Headings

HTML code for *Headings* typically appear as follows. Various components of the example vary according to the context of the Heading Level. This example is to be used as a guide only. If actual code is required, refer to the ***Document Content Template***.

**HTML:**

```
<font face="Arial, Helvetica, Sans-Serif" size="{X}">
<p><br><a name="[SectionName]"></a>
  {X.X.X} [Section Name]
</p>
</font>
```

## *3.5 Code Commenting*

HTML code in most cases is obviously self-explanatory; consequently commenting would be considered unnecessary. The writer of the code is free to comment as he/she wishes, but it must be noted that commenting in

**Team Synergy** documentation is used sparingly, and only to emphasise the reusability of a portion of code.

## *3.6 HTML Code Example*

Readers of this *HTML Coding Style* section should refer to the ***Document Content Template*** available from here, for an actual, usable example.

Alternatively, it would be more convenient to refer to the HTML source for this document.