**Emobiix Application Development Guide**

For PEEK INC Only

Table of Contents

Introduction 3

Application XML Elements 4

Understanding This Section 4

Common Fields 4

Application 6

Image 7

View 8

Box 9

Label 10

Text 11

Button 12

Entry 13

Stack 14

Array 15

Set 16

Progress 17

Checkbox 18

Radio 19

Scroll 20

Frame 21

Formatted Text 22

Memory and Performance Behavior 22

Bold 23

Underlined 23

Italic 24

Color 24

Background Color 24

Font Size 25

Scripting Reference 26

DataObject API 27

# Introduction

This document outlines the general architecture of application developed for the emobiix platform along with specifics on aspects of applications that are independent of architecture design.

# Application XML Elements

Emobiix applications contain an XML component to describe the visual characteristics, UI, scripts, data usage, and policy resident on the client side. This section defines the XML structure and how elements relate to their client side aspect.

## Understanding This Section

This chapter outlines various XML tags, fields, and contents. Understanding the following sections involves understanding how relevant information about each entity is presented. The format of following sections is as specified.

XML tag names are represented in bold text. For example the **label** tag would appear in bold in text.

XML field names are represented in italic text. For example the *id* field would appear italic.

Field names will be presented in a table for each tag type. The first column in the table indicates the field name, the second indicates the possible values, the third provides a description. Possible values could be blank to indicate that the possible values are not reasonably finite or useful to document. In XML tag field lists, mandatory field will be indicated with a blue background whereas optional fields will have a yellow background as shown. In the example below, headings are present for each column to aid in documentation, however these headings will not appear in actual definition lists

|  |  |  |
| --- | --- | --- |
| **Field Name** | **Possible Types** | **Description** |
| *name* |  | Identifying |
| *canfocus* | “0” or “1” | Marks widget as focusable or not |

A definition of an entity may contain an example. Such an example will be shown in the form below, always proceeded by a description of the example:

|  |
| --- |
| <tagname field=”value1” field2=”value2”/> |

Using a **tagname** to hold values for *field1* and *field2* -

## Common Fields

Emobiix widgets all derive from the same basic widget type. This basic type offers some characteristics that are relevant for most widget types and thus considered common fields. These fields largely control layout, interaction, style, and function.

Although common fields apply to many widget types, not all widgets can make use of all common fields, therefore common fields will be outlined in each widgets’ representing XML node section.

In addition to fields appearing on the common field list, they may be repeated in a node’s section if the definition is slightly different or serves a different purpose.

|  |  |  |
| --- | --- | --- |
| *id* | string | Style identifier of node element. Controls visual aspect. Default implies default style |
| *name* | string | Unique identifier used to locate element |
| *width* | number or  percentage | Requested width of widget. May be a number to specify pixels, a percentage to specify width relative to parent, or omitted to imply based on children |
| *height* | number or  percentage | Requested height of widget. May be a number to specify pixels, a percentage to specify height relative to parent, or omitted to imply based on children |
| *alignment* | “left”, “center”, or “right” | Positioning of widget relative to parent. For vertical alignment, values can be taken literally. For horizontal alignment, “left” represents “top” and “right” represents bottom. Default is “left” |
| *packing* | “horizontal” or “vertical” | Positioning applied to children widgets relative to self. Horizontal packing places widgets next-to-each-other, vertical packing places them on-top-of-each-other |
| *script* | Lua script | Specifies a script context to be used for this and child widgets |
| *canfocus* | “0” or “1” | Specifies if a widget can be focused |
| *onreturn* | Lua script | Script executed when the widget is focused and the return input is actioned |
| *accesskey* | Character | Specifies that when an unhandled key press matching the character is received by the application, the onreturn script for this widget should be executed |
| *color* | RGBA hex color | Color in the hex format of RRGGBBAA |
| *weight* | “normal” or “bold” | Weight of a text format, either normal, or bold for thicker darker text. |
| *reference* | Name or URL | DataObject name or location that should be used for contents data instead of this field’s widget itself. |

## Application

**application** is the root XML node for any emobiix application and defines meta-information and application-wide properties.

|  |  |  |
| --- | --- | --- |
| *name* | string | Name used when presenting application in any UI objects, eg application lists, task switching, etc |
| *startupview* | name | Object name of view in application to be used on application launch. Initial view focused |
| *description* | string | Brief explanation of application purpose |
| *Icon* | name | Object name of icon for application. Name must resolve to an image element within the application. |
| *onload* | Lua script | Script to execute when application is loaded |
| *onfocus* | Lua script | Script to execute when application becomes focused |
| *onunfocus* | Lua script | Script to execute when application looses focus |

Common Fields: *script*

|  |
| --- |
| <application name=”Hello World” startupview=”mainview” icon=”appicon”>  <image name=”appicon” src=”hello\_world.png”>  <view name=”mainview”>  <label>Hello World</label>  </view>  </application> |

Hello world **application** with an *icon* reference and a single **view** -

### Image

An **image** defines a standard graphics file component. The contents of the image is specified in the *src* field.

Supported image file formats are PNG, and JPG. MNG, GIF, and GIF-GCE may also be supported largely depending on patent licensing and platform capabilities.

|  |  |  |
| --- | --- | --- |
| *src* | filename | Filename of supported image file |
| *transparency* | “none”, “stencil”, “full” | If “none”, pixels from image are presented to display unaltered. If “stencil”, color 0 (black) is considered completely transparent, every other color is considered completely opaque. If “full”, RGBA images will have alpha map to opacity, black and white images will have intensity map to opacity. Default is “full” |
| *color* | Hex string | RGBA color to be applied to a black and white image to interpret as intensity of color. Alpha component is ignored. Default is white |

Common fields: *id*, *name*, *packing*, *alignment, script*, *canfocus*, *onreturn*, *accesskey*

|  |
| --- |
| <image name=”childindicator” src=”leaf\_mask.png” transparency=”stencil” color=”8F000000”/> |

Stencil **image** displayed in red -

### View

A view is a top-level widget that contains child widgets to make up what is seen on the client. A view can be considered the root visible node. An application is typically made up from many views, switching between them to control application flow.

A view itself has no visual representation other than it is a container for other widgets. It can however, effect layout of child widgets since it is a container.

Common fields: *id, packing, name, alignment, script, width, height*

|  |
| --- |
| <image name=”childindicator” src=”leaf\_mask.png” transparency=”stencil” color=”8F0000”/> |

Stencil **image** displayed in red -

### Box

A **box** is a basic layout element used to vary packing and alignment styles. Optionally, the box may also have a visual representation if so defined by its assigned style.

Label1

Label2

Label3

Label4 Label5 Label6 Label7

Box showing horizontal layout (dotted), and one showing visual style (label8) -

Label8

An example usage of a box is where you have a view of vertically packed elements where you would like to add some additional elements that are horizontally packed. You can use a box without visual representation to achieve this. Another use would be if you wish to create a visual box around some elements, you could insert them into a box with a defined visual style.

Common fields: *id, name, packing, alignment, script, width, height, canfocus, onreturn, accesskey*

|  |
| --- |
| <box name=”labelbox” width=”90%” packing=”horizontal”>  …  </box> |

Grouping box to align children horizontally in 90% of parent width -

### Label

A label is a widget that can represent a single line body text in a single format. It is non-interactive and often static but can be dynamic where needed. Labels are efficient at display simple text and should be used wherever acceptable in place of more complex text widgets.

The displayed text of a label is controlled by the contents of the XML node.

Width and height of a label can be automatically determined to contain the text within the label. It is also possible to explicitly set the size of the label widget. Any additional text that would fall outside the size allocation is not displayed.

Common fields: *id, packing, name, alignment, width, height, color, weight*

|  |
| --- |
| <label color=”AAAAAA00”>Label Text</label> |

Simple **label** to display the text “Label Text” in grey -

### Text

A **text** widget is a more advanced version of the label widget. It can represent a single or multiple lines of formatted text. It is non-interactive and often static but can be dynamic where needed. Text widgets should be used when advanced formatting is required within a single block of text or if multiple lines of text are needed in a single widget.

The displayed text of a label is controlled by the contents of the XML node.

Height of a label can be automatically determined to contain the text within the label if the width is given. It is also possible to explicitly set the height of the text widget. Any additional text that would fall outside the size allocation is not displayed.

Common fields: *id, packing, name, alignment, width, height, color, weight*

|  |
| --- |
| <text width=”90%”>&lt;b&gt;Sample text:&lt;/b&gt;   1. Find tags 2. Replace tags 3. ... 4. &lt;u&gt;Profit&lt;/u&gt;</text> |

Simple **text** to display the formatted complex text -

### Button

A button widget is specifically for receiving user interaction from button clicks. It is typically used to trigger scripting events.

The visual representation of a button is similar to a box in that it takes a rectangular shape and changes in appearance when focused if focus is allowed. A button may contain any other widgets to visually represent the outcome of user interaction.

Common fields: *id, packing, name, alignment, script, width, height, canfocus, onreturn accesskey*

|  |
| --- |
| <button onreturn=”toScreen(mainscreen)”>  <label>To Main Screen</label>  </button> |

UI **button** with script action and label -

### Entry

An **entry** widget is a UI widget to allow text to be entered by the user for application use. An entry may either by multi-line or single line. For single line entry, the widget is seen as a window in a single line of text. For multi-line entry, the widget is seen as a window into multiple lines of text that can be scrolled up and down with an indication of where the window is within the text.

Input from the user is via the keypad for characters, and via the wheel for navigation. Moving the wheel down, moves the cursor forward a character, up, moves the cursor backward a character. Using shift and the wheel moves the cursor up a line when moved up, and down a line when moved down.

|  |  |  |
| --- | --- | --- |
| *onentry* | Lua script | Called when the text within the entry is changed |
| *onnavigate* | Lua script | Called when the cursor is moved without altering the contents of the text entry |
| *multiline* | “true” or “false” | Specifies if the entry should contain one line or multiple lines of text. Default “false” |

Common fields: *id,, name, alignment, script, width, height, canfocus, onreturn accesskey, reference*

|  |
| --- |
| <entry width=”90%” height=”40” multiline=”true/> |

Text **entry** with multiple lines -

### Stack

A stack is a non-visual layout widget allowing you to stack widgets on top of each other. A basic description is that every child of a stack is drawn at the same location. The stack takes on the largest sized child where width and height could come from different children.

A stack could be used to add emblems to objects, display overlays, fancy overlaid graphics, or other artistic displays. At a system level, stacks are used to display popup menus and dialog boxes.

Common fields: *id, name, alignment, script, width, height, canfocus, onreturn accesskey, reference*

|  |
| --- |
| <stack>  ...  </stack> |

Simple **stack** with children -

### Array

A fundamental widget for displaying record data is the **array** widget. An array widget works by taking a record and creating one instance of the arrays children for each element in the record. This effectively allows you to specify how one record should be displayed and have the application display any record using this template.

Arrays can be used in two different modes. The first mode is to *enclose* the elements in a container and provide a scroll bar to navigate through the container. This provides a good interface for navigating large numbers of record, or when you need to be able confine the array to a given size container on the display.

The second mode an array can be used is to *expand* out into a non-scrollable list. This format is ideal when you already have content in a scrollable area and you want to have some additional content above and below the array that will scroll with the records.

The contained mode is most useful for displaying traditional list information in applications whereas the expanded mode is most useful for creating arrangements that resemble a webpage where the whole page is scrollable, not just the items in the record array.

Special fields for array children control whether or not their data is sourced from a record or sourced normally. The *arraysource* field when set to “true” informs the application that this widget should be rendered with its contents from the record. The *datafield* field controls which field of the record is mapped to the widgets content.

Common fields: *id*, *name*, *packing*, *alignment, script*, *canfocus*, *onreturn*, *accesskey, reference*

|  |
| --- |
| <array reference=”tcp://host/all\_records/”>  <label arraysource=”true” datafield=”name”/>  </array>  Combined with array :  [David, Sam, James, Derrick]  Yields the same as:  <label>David</label>  <label>Sam</label>  <label>James</label>  <label>Derrick</label> |

Label **array** showing name values from remote object -

### Set

The **set** widget is a unique widget that embodies the switch-case construct from C-like programming languages. A **set** allows completely changing which widgets are display based on DataObject values.

Each child in a set can represent a widget that is drawn if a DataObject’s field matches a given value. Importantly, at most one child will match and be displayed at any one time.

The sister object to the **set** is the **setitem** which, continuing with the switch-case analogy, is the case statement. It has fields to determine if this item is the current item to draw.

An example usage of a set would be to display a status icon dependent on another DataObject.

|  |  |  |
| --- | --- | --- |
| *fieldname* | string | Field of referenced object or self to use when deciding which setitem to render. |
| *fieldvalue* | string | Value used to match data from *fieldname* to determine if setitem is to be displayed. If empty, this will always match. This field applies to setitem only. |

Common fields: *id*, *name*, *packing*, *alignment, script*, *canfocus*, *onreturn*, *accesskey, reference*

|  |
| --- |
| <set fieldname=”data”>  <setitem fieldvalue=”1”><label>On</label></setitem>  <setitem fieldvalue=”0”><label>Off</label></setitem>  <setitem><label>Error</label></setitem>  </set> |

Label **set** with values for on, off, and error -

### Progress

A **progress** widget is used to display the progress of some task or that some progress is taking place.

Visually a **progress** widget is shown as a box that is partially filled depending on the progress value assigned to it. Alternatively, if there is no quantity of progress then the progress will simply animate to represent that progress is taking place to assure the user that their usage is having an effect.

|  |  |  |
| --- | --- | --- |
| *progressfield* | string | Field to take progress value from. Default is to use the contents of widget |
| *totalfield* | string | Field to take total number used for progress calculation – progressfield / totalfield \* 100%. Default is the value 100. |
| *indeterminate* | “true”, or “false” | If true, then an animation is played without any indication of progress value, if false, then a bar showing percentage progressed. |

Common fields: *id, packing, name, alignment, script, width, height, canfocus, onreturn accesskey*

|  |
| --- |
| <progress width=”100” height=”20” reference=”downloadbytes” totalfield=”size” progressfield=”recv”/> |

Download **progress** showing bytes complete -

## Checkbox

A **checkbox** widget is used to display a Boolean selection. It can either be checked, or unchecked.

Common fields: *id, packing, name, alignment, script, width, height, canfocus, onreturn accesskey*

|  |
| --- |
| <checkbox name=”secureon”/> |

Security **checkbox** -

## Radio

A **radio** widget is used in a group to represent a mutually exclusive choice in a group of **radio**s.

Radio widgets are typically used to represent choices where only one and only one option is acceptable. In addition to making a choice, the radio group can be used as a source of the choice.

The first radio to appear in the application XML for a group will be the default selected **radio**.

|  |  |  |
| --- | --- | --- |
| *group* | string | Name of DataObject to use as a group reference. This object controls which radio is chosen |
| *choicename* | string | Name to have the group know this radio by. When referencing the group, this name will be the contents if this radio is chose |

Common fields: *id, packing, name, alignment, script, width, height, canfocus, onreturn accesskey*

|  |
| --- |
| <box name=”radiosecurity”>  <radio group=”radiosecurity” choicename=”0”/><label>None</label>  <radio group=”radiosecurity” choicename=”1”/><label>3DES</label>  <radio group=”radiosecurity” choicename=”2”/><label>AES</label>  </box> |

Security **radio** allowing selection of standard -

## Scroll

A **scroll** widget is used to layout child widgets in an area that will automatically scroll with user interaction.

A scrolled area is typically used to contain large vertical lists of child widgets that wouldn’t normally fit on a single display by make sense to be in the same container. An example is a long list of controllable settings, or a list of messages in “web” style layout.

A scroll allows for any size child area that is automatically determined by its children. A scroll bar is only shown if the region is too large to be completely displayed.

Common fields: *id, packing, name, alignment, script, width, height, canfocus, onreturn accesskey*

|  |
| --- |
| <scroll>  ...  </scroll> |

Layout **scroll** window -

## Frame

A **frame** widget allows an application to be contained within another application. Displaying another application allows simple applications to be included in larger applications to allow reuse and simplification of application development.

An example usage of a **frame** would be to embed a traditional desktop applet such as a clock or weather service so that these could be added to a display.

Common fields: *id, packing, name, alignment, script, width, height, canfocus, onreturn accesskey*

|  |
| --- |
| <frame with=”10” height=”10” reference=”tcp://server/weather-icon/NY”/> |

External application displayed in a **frame** -

## Formatted Text

Some widgets are capable of displaying text that has more than just a single uniform format. Formatted text loosely resembles basic HTML text formatting tags but differs in the exact tags and their usage.

Formatting works on a formatting state concept where a tag will change the current state of formatting. A section of text without formatting will use the formatting in the current state until the text ends or another tag occurs in the text, which could then change the formatting state. The state has some concept of default state, which could be specified by the widget.

Back to default color

Bold Removed

Passage of text <b>with</b> some <cAAFF0000>tags</c> along the way

Default state

Color Added

Bold Added

Any combination of formatting states can be active at the same time. There is no restriction on keeping tags even or concentric.

The remainder of this section outlines the tags and their usage with relevant examples. Technical considerations for each tag are shown specific to the platform this document is a variant for.

### Memory and Performance Behavior

Each character that is drawn to the screen is stored in an internal font cache as a bitmap rendered from a true type font file. Understanding the conditions on when a character can be reused and when one has to be regenerated can give understanding to the memory usage and performance of an application.

As a general rule, each character has a unique bitmap. A character may be a lower case ‘a’, an upper case ‘A’, any other letter, or symbol ‘$’. Importantly lower case and upper case letters require unique bitmaps. A bitmap is also unique for the size of a font so for example, having text with three different sizes of font will use more memory than using only two sizes of font – but this would only be the case in an isolated text instance. You should consider the sizes and styles used throughout your entire application and in other applications since the bitmaps are shared across text sections and even applications.

Character bitmaps are particular to each font file. Certain formatting may also incur additional bitmaps.

Below is an example of two text sections and all the bitmaps used in each section.

This is some text  
 - 9 bitmaps (Thisometx)

Thi**s** Is some TEXT  
 - 11 bitmaps (Thi**s**IsomeEX)

Each bitmap represents each pixel by four bits. A sample bitmap five pixels wide and eight pixels tall will use 20 bytes of memory as calculated by width times by height divided by two.

A general rule about keeping font memory consumption low is to decide upon a set of font sizes and not vary beyond these. For example, using three font sizes of 12 pixels, 10 pixels, and 8 pixels gives some variation without excessive memory consumption that would result from using all sizes between twelve and eight.

### Bold

The bold tag works nearly identical to the HTML <strong> tag. It works by either locating a bold variation of the font in use or uses an embolder to generate bold characters from standard glyphs.

Bolding is an acceptable way of adding emphasis to text and special care has been taken to allow for no performance difference when using bold text. However, in either case of there being a bold font or using the embolder, there is a memory overhead equal to one byte per two pixels of glyph where a glyph is reused for every instance of the same character.

|  |
| --- |
| This text contains some <b>bold</b> formatting. |
| This text contains some **bold** formatting |

Example of bold formatting in text -

### Underlined

The underline tag works nearly identical to the HTML <u> tag. Underlining is applied by drawing a 1-pixel line under each glyph in the current color of the font.

Since underlining does not alter the character’s appearance it has no storage overhead and negligible performance overhead.

|  |
| --- |
| This text <u>is underlined</u> in places |
| This text is underlined in places |

Example of underlined formatting in text -

### Italic

The italic tag works nearly identical to the HTML <i> tag. Italic is applied by either using an italic font if available or by slanting the upright bitmap.

Using a font for slanting incurs the overhead of the additional font. Slanting a bitmap incurs zero overhead. Slanting a bitmap produces sharp edges on each character – a tradeoff between performance and memory overhead, and visual appearance.

|  |
| --- |
| This text looks <i>italic</i> a bit |
| This text looks *italic* a bit |

Example of italic formatting in text -

### Color

Text can have the pixels that make up the character colored. This tag is unique in its formatting and use. Colors interact with the state to either change to a new color or return to the default color of the text. It is possible to have many opening tags to change the color many times and only use one closing tag to return to the default color.

The tag is represented by a ‘c’ character followed by a 32bit hex color value in RGBA format, standard color format, where the A component is ignored. A closing tag is just a simple closing ‘c’ without color.

Text coloring will also affect the color of any text decorations such as underlining.

Text coloring uses the internal font rendering system to colorize the characters and incurs zero overhead.

|  |
| --- |
| Text can go from <cFF000000>red to <c0000FF00>blue and back to </c>default |
| Text can go from red to blue and back to default |

Example of color formatting in text -

### Background Color

The ‘g’ tag is used to change the color of the background behind each character while not altering the color of the pixels of the character itself. Background colors interact with the state to either change to a new color or return to the default color It is possible to have many opening tags to change the color many times and only use one closing tag to return to the default color.

The background tag ‘g’ is followed by a 32bit hex color value in RGBA format, standard color format, where the A component is ignored. A closing tag is just a simple closing ‘g’ without color.

A default background color can be considered zero overhead and a changed background overhead can be considered trivially more performance overhead.

|  |
| --- |
| Text can go from <gFF000000>red to <g0000FF00>blue and back to </g>default |
| Text can go from red to blue and back to default |

Example of background color formatting in text -

### Font Size

The size of a font can be varied within text using the ‘s’ tag. This tag is unique in its formatting and use. Font sizes interact with the state to either change to a new size or return to the default size of the text. It is possible to have many opening tags to change the size many times and only use one closing tag to return to the default size.

The size tag ‘s’ is followed by an integer representing the pixel height of the font. A closing tag is just a simple closing ‘s’ without size.

Changing a fonts size alters every character that is draw and thus imposes a memory overhead for each unique character drawn.

|  |
| --- |
| Text <s12>size can <s14>be</s> varied |
| Text size can be varied |

Example of font size formatting in text -

# Scripting Reference

Applications can specify functionality beyond the implicit operation of the interface using scripting. The scripting language used is Lua ([**http://ww.lua.org/**](http://ww.lua.org/)), a common language for embedded or limited environments.

Lua comes with a number of standard packages that are exposed in the application for achieving standard programming goals. The standard libraries included are as follows.

* String library – extended to use UTF-8
* Pattern matching
* Tables
* Mathematical Functions
* Operating System Facilities – I/O and environment functions unavailable
* Debug

Absent standard libraries include:

* I/O -  
  There is no sense of input or output from file in the application framework so it is thus not possible to perform I/O from script

In addition to the standard libraries, application environment specific API is available. This API provides access to the DataObject concept of applications allowing modification of application elements.

## DataObject API

In a basic understanding, the DataObject API allows scripts to locate specific DataObjects, either within an application or at a remote network location, and allow scripts to modify DataObjects.

*DataObject:locate(location)*  
Returns a DataObject specified by the URL location. If the DataObject is does not have a local cached instance, a place holder object is returned with its state set to “initializing” while the remote object is fetched.

The location need not be remote, it may specify a locally URL addressable object.

*o:toScreen()*  
Returns true if ‘o’ is a DataObject of ‘view’ type and the object could be set to the current view. This has the effect of changing the active view to another view.

A typical order of operations would be to locate a local view DataObject then use *toScreen* to change to that view.

*DataObject:find(name)*  
Returns a DataObject within the current application with given *name*. If the DataObject is not found, *nil* is returned.

The name for local DataObjects is specified by the name attribute in their XML definition.

*o:getValue([name])*  
Returns the value for field specified by name on DataObject ‘o’ or if name is omitted, it returns the contents of the DataObject ‘o’. DataObjects with contents are those that display text or images. If no field exists with given name or DataObject does not have any contents, then nil is returned.

Actual return type may be string, integer, array, etc, depending on field value. It is common to have numeric field returned as string.

*o:setValue(value, [name])*  
Sets the value for field of DataObject ‘o’ specified by *name* to *value*. If name is not specified then the contents of the object is set. DataObjects with contents are those that display text or images.

If it is not possible to set the field to given value then nil is returned, otherwise true is returned.

Using a **tagname** to hold values for field1 and field2 - 1 4

Hello world **application** with an icon reference and a single **view** - 2 6

Stencil **image** displayed in red - 3 7

Stencil **image** displayed in red - 4 8

Grouping box to align children horizontally in 90% of parent width - 5 9

Simple **label** to display the text “Label Text” in grey - 6 10

Simple **text** to display the formatted complex text - 7 11

UI **button** with script action and label - 8 12

Text **entry** with multiple lines - 9 13

Simple **stack** with children - 10 14

Label **array** showing name values from remote object - 11 15

Label **set** with values for on, off, and error - 12 16

Download **progress** showing bytes complete - 13 17

Security **checkbox** - 14 18

Security **radio** allowing selection of standard - 15 19

Layout **scroll** window - 16 20

External application displayed in a **frame** - 13 21

Example of bold formatting in text - 8 23

Example of underlined formatting in text - 9 23

Example of italic formatting in text - 10 24

Example of color formatting in text - 10 24

Example of background color formatting in text - 12 25

Example of font size formatting in text - 15 25