h**uey:** Internal maintenance specification

John W. Shipman

2011-11-04 15:54

Abstract

Describes the implementation of **huey**, a graphical application for selecting colors and fonts, using the Python programming language and the Tkinter widget set.

This publication is available in Web form\(^1\) and also as a PDF document\(^2\). Please forward any comments to **tcc-doc@nmt.edu**.

Table of Contents

1. Introduction ............................................................................................................................ 4

   1.1. Files discussed in this publication ................................................................................... 4

2. References ............................................................................................................................... 4

3. Design notes ............................................................................................................................ 4

4. Code prologue ......................................................................................................................... 6

   4.1. Imports ......................................................................................................................... 6

   4.2. Manifest constants ......................................................................................................... 7

5. **main():** The main program ............................................................................................. 7

6. **class Color:** A specific color ............................................................................................ 8

   6.1. **Color.__init__():** Constructor .................................................................................. 9

   6.2. **Color.__standardize():** Standardize color component representation ....................... 9

   6.3. **Color.__str__():** Convert to a string ........................................................................ 10

   6.4. **Color.__cmp__():** Compare two colors ..................................................................... 10

7. **class ColorModel:** Base class for color models ............................................................. 11

   7.1. **ColorModel.__init__():** Constructor ...................................................................... 11

   7.2. **ColorModel.paramsToColor():** ............................................................................... 12

   7.3. **ColorModel.colorToParams():** ............................................................................... 12

   7.4. **ColorModel.normalize():** Normalize an integer color parameter .............................. 12

   7.5. **ColorModel.discretize():** Discretize an integer color parameter .............................. 13

8. **class HSVModel:** Hue-saturation-value color model ...................................................... 13

   8.1. **HSVModel.__init__():** Constructor ........................................................................ 13

   8.2. **HSVModel.paramsToColor():** ............................................................................... 14

   8.3. **HSVModel.colorToParams():** ............................................................................... 15

9. **class RGBModel:** Red-green-blue color model .............................................................. 16

   9.1. **RGBModel.__init__():** Constructor ........................................................................ 16

   9.2. **RGBModel.paramsToColor():** ............................................................................... 16

   9.3. **RGBModel.colorToParams():** ............................................................................... 17

10. **class CMYModel:** Cyan-magenta-yellow color model .................................................. 17

   10.1. **CMYModel.__init__():** .................................................................................. 17

---


10.2. CMYModel.paramsToColor() .................................................. 17
10.3. CMYModel.colorToParams() .................................................. 18

11. class Application: The outermost frame .......................................... 18
11.1. Application.__init__(): Constructor ............................................. 19
11.2. Application.__createWidgets(): Set up the widgets ......................... 19
11.3. Application.__nameHandler(): Name picker handler ...................... 21
11.4. Application.__adjustHandler(): Adjuster handler ......................... 21

12. class MenuBar: General controls ..................................................... 21
12.1. MenuBar.__init__(): Constructor ................................................ 22
12.2. MenuBar.__createHelp(): Create the Help menu .............................. 23
12.3. MenuBar.__cascadeNamePicker() .................................................. 24
12.4. MenuBar.__helpTyping() ............................................................ 25
12.5. MenuBar.__helpPicking() ............................................................ 26
12.6. MenuBar.__cascadeAdjuster() ...................................................... 26
12.7. MenuBar.__helpReadout() .......................................................... 27
12.8. MenuBar.__helpModelSelector() .................................................. 27
12.9. MenuBar.__helpHSV() ............................................................... 27
12.10. MenuBar.__helpRGB() .............................................................. 28
12.11. MenuBar.__helpCMY() .............................................................. 28
12.12. MenuBar.__helpSliders() ........................................................... 28
12.13. MenuBar.__cascadeViewing() .................................................... 29
12.15. MenuBar.__cascadeFonts() ........................................................ 30
12.16. MenuBar.__helpFontFamily() ..................................................... 30
12.17. MenuBar.__helpFontSize() ........................................................ 31
12.18. MenuBar.__helpFontStyle() ....................................................... 31
12.19. MenuBar.__helpImporting() ....................................................... 31
12.20. MenuBar.__helpAuthor() .......................................................... 32
12.21. MenuBar.__dialog(): Pop up a dialog widget .................................. 32

13. class NamePicker: Select colors by name .......................................... 33
13.1. NamePicker.__init__(): Constructor ............................................ 34
13.2. NamePicker.__createWidgets(): Create all widgets ......................... 34
13.3. NamePicker.__entryHandler(): Handler for the Enter key .................. 35
13.4. NamePicker.__setByName(): Try to convert a color name to a color ....... 36
13.5. NamePicker.__pickListHandler(): Somebody clicked on a color .......... 37

14. class PickList: The color names pick list .......................................... 37
14.1. PickList class constants .................................................................. 39
14.2. PickList: Default color list ............................................................ 40
14.3. PickList.lookupName(): Convert a color name to a color value ............ 45
14.4. PickList.__init__(): Constructor .................................................. 45
14.5. PickList.__addColors(): Populate the color list ............................... 47
14.6. PickList._findColorsFile(): Read the rgb.txt file ........................... 48
14.7. PickList.__readColorsFile(): Process the rgb.txt file ...................... 49
14.8. PickList.__readColorLine(): Process one line from rgb.txt ............... 50
14.9. PickList.__convert8(): Convert an 8-bit number ............................. 52
14.10. PickList.__useDefaultColors(): Set up a default color list ................ 52
14.11. PickList.__cleanColorMap(): Remove redundant colors ................... 53
14.12. PickList.__nameCleaner(): Remove redundant color names ............... 54
14.13. PickList.__anglicize(): Convert “gray” to “grey” ............................. 55
14.14. PickList.__lowerize(): Convert intercapitalized names to lowercase ...... 56
14.15. PickList.__purgeName(): Remove one redundant color name ............ 56
14.16. PickList.__pickHandler(): Somebody clicked on a color name ........... 57
15. class Adjuster: Color adjustment and color model selection ........................................ 58
  15.1. Manifest constants ........................................................................................................ 59
  15.2. Adjuster.set(): Change the color ............................................................................... 60
  15.3. Adjuster.textColor(): Return the current text color ............................................... 60
  15.4. Adjuster.bgColor(): Return the current text color ............................................... 60
  15.5. Adjuster.isText(): Which color is being displayed? ............................................... 61
  15.6. Adjuster.__init__(): Constructor ........................................................................... 61
  15.7. Adjuster.__createWidgets(): Create internal widgets ........................................... 62
  15.8. Adjuster.__readoutHandler(): Change between text and background colors .......... 63
  15.9. Adjuster.__modelHandler(): Change the current color model ................................. 64
  15.10. Adjuster.__sliderHandler(): Handler for slider changes ....................................... 64

16. class ColorReadout: Text and background color readouts .............................................. 65
  16.1. ColorReadout.isText(): Which color are we displaying? ........................................ 66
  16.2. ColorReadout.set(): Change the displayed color ............................................... 67
  16.3. ColorReadout.internalSet() .................................................................................... 67
  16.4. ColorReadout.textColor(): Return the current text color ................................... 68
  16.5. ColorReadout.bgColor(): Return the current background color ......................... 68
  16.6. ColorReadout.__init__(): Constructor ................................................................... 68
  16.7. ColorReadout.__createWidgets() ............................................................................ 69
  16.8. ColorReadout.__radioHandler(): Radiobutton state change handler .................. 71
  16.9. ColorReadout.__fixNames(): Prevent user modification of the Entry widgets .......... 71

17. class ModelSelector: Widgets to select a color model .................................................... 72
  17.1. ModelSelector: Constants ....................................................................................... 73
  17.2. ModelSelector.getModel(): Return the current model ........................................... 73
  17.3. ModelSelector.__init__(): Constructor ................................................................ 74
  17.4. ModelSelector.__createWidgets() ......................................................................... 74
  17.5. ModelSelector.__radioHandler(): Handler for radiobutton action ....................... 75

18. class ColorSliders: Color model parameter sliders ...................................................... 76
  18.1. ColorSliders.setModel(): Display a new color model ........................................... 77
  18.2. ColorSliders.setColor(): Change the displayed color ......................................... 78
  18.3. ColorSliders.__init__(): Constructor ................................................................... 78
  18.4. ColorSliders.__createWidgets() ............................................................................ 79
  18.5. ColorSliders.__sliderHandler(): Handler for a parameter change ....................... 80

19. class ParamSlider: Scale widget for a color model parameter ......................................... 80
  19.1. ParamSlider.get(): Retrieve the current parameter value ..................................... 82
  19.2. ParamSlider.setModel(): Change the color model .............................................. 82
  19.3. ParamSlider.setColor(): Change the current color ............................................ 82
  19.4. ParamSlider.__init__(): Constructor ................................................................ 83
  19.5. ParamSlider.__createWidgets() ......................................................................... 84
  19.6. ParamSlider.__plusHandler(): Increment the parameter ..................................... 85
  19.7. ParamSlider.__minusHandler(): Decrement the parameter .................................. 86
  19.8. ParamSlider.__scaleHandler(): Somebody dragged the scale ............................... 86

20. class Swatch: Font and color samples ........................................................................... 87
  20.1. Class constants .................................................................................................... 88
  20.2. Swatch.setTextColor() ......................................................................................... 88
  20.3. Swatch.setBgColor() .......................................................................................... 88
  20.4. Swatch.__init__(): Constructor ........................................................................... 89
  20.5. Swatch.__createWidgets() .................................................................................. 89
  20.6. Swatch.__fontHandler(): Handle a font change .................................................. 90

21. Code epilogue ................................................................................................................ 90
1. Introduction

This document describes the implementation of huey, a graphical user interface tool for displaying various colors and text fonts. See the external specification, huey: a color and font selection tool\(^3\).

For those interested in analyzing this program as an example of Python and Tkinter programming, the entire Python source for the program is presented here using a lightweight literate programming (LLP) style. For more on LLP, see the author's literate programming page\(^4\).

1.1. Files discussed in this publication

Here are online links to all the files required by this program (other than Python and Tkinter installs).

- huey\(^5\): The main script.
- scrolledlist.py\(^6\): Source for the ScrolledList compound widget.
- fontselect.py\(^7\): Source for the FontSelect compound widget.
- hueyims.xml\(^8\): DocBook source for this document.

2. References

The author has relied heavily on one book as a comprehensive reference on color theory as it relates to computer graphics.


3. Design notes

Before proceeding, be sure you have a basic understanding of how Tkinter applications are built. Strongly recommended is the author's Tkinter reference\(^9\).

This program has a lot of widgets. To keep the design conceptually simple, the architecture of the program uses boxes within boxes within boxes, grouping related widgets into frames, and related frames into larger frames, so that each grouping has a well-defined function, and connections within a grouping of widgets are hidden within a single Python class.

One way to describe this technique is that we build compound widgets out of Tkinter's basic widgets. A compound widget is a Python class that inherits from the Frame widget.

This application uses the \texttt{.grid()} geometry manager to position widgets. At the very highest level, the application is organized into two rows and three columns like this:

\(^3\) http://www.nmt.edu/tcc/help/lang/python/examples/huey/
\(^4\) http://www.nmt.edu/~shipman/soft/litprog/
\(^5\) http://www.nmt.edu/tcc/help/lang/python/examples/huey/ims/huey
\(^6\) http://www.nmt.edu/tcc/help/lang/python/examples/huey/ims/scrolledlist.py
\(^7\) http://www.nmt.edu/tcc/help/lang/python/examples/huey/ims/fontselect.py
\(^8\) http://www.nmt.edu/tcc/help/lang/python/examples/huey/ims/hueyims.xml
\(^9\) http://www.nmt.edu/tcc/help/pubs/tkinter/
The three columns of row 0 are spanned to make one wide, short grid cell that contains the general controls. The name .menuBar has a dot in front of it because it is an attribute of the Application class, the Frame that contains all the application's widgets. The compound widget that appears in this grid cell is an instance of class MenuBar; see Section 12, “class MenuBar: General controls” (p. 21).

Similarly, three compound widgets fill out row 1. Column 0 contains a NamePicker compound widget, which is stored in attribute .namePicker of the Application class. Column 1 is an Adjuster widget, and column 2 is a Swatch widget.

Below this level, design of each of the four major areas proceeds by stepwise refinement. Here is a diagram showing the layout of all the compound widgets in this program.

- Section 11, “class Application: The outermost frame” (p. 18) is the entire application.
- Section 12, “class MenuBar: General controls” (p. 21) contains the controls along the top edge.
- Section 13, “class NamePicker: Select colors by name” (p. 33) contains all controls for selecting colors by name.
- Section 14, “class PickList: The color names pick list” (p. 37) is the list of standard colors.
- Section 15, “class Adjuster: Color adjustment and color model selection” (p. 58) contains controls for selecting text or background color, selecting the color model, and adjusting colors.
- Section 16, “class ColorReadout: Text and background color readouts” (p. 65) shows the current text and background color names and allows the user to select which color is being adjusted.
- Section 17, “class ModelSelector: Widgets to select a color model” (p. 72) allows the user to select a color model.
- Section 18, “class ColorSliders: Color model parameter sliders” (p. 76) is a set of Scale widgets for making fine color adjustments.
- Section 19, “class ParamSlider: Scale widget for a color model parameter” (p. 80) is one of those Scale widgets.
- Section 20, “class Swatch: Font and color samples” (p. 87) displays the current font in the current text color against the current background color, and includes widgets for selecting the font.
The actual code for *huey* starts in Section 4, “Code prologue” (p. 6). The actual program logic starts with Section 5, “main(): The main program” (p. 7).

## 4. Code prologue

The actual code starts with the usual Unix “pound-bang line” that makes the script self-executing. This is followed by a comment pointing the reader at this documentation.

```python
#!/usr/local/bin/python
#---------------------------------------------
# huey: A color and font selection tool
# For documentation, see:
#     http://www.nmt.edu/tcc/help/lang/python/examples/huey/ims/
#---------------------------------------------

PROGRAM_NAME     = "huey"
EXTERNAL_VERSION = "1.0"
```

### 4.1. Imports

Here are the module imports for this script. First is the standard Python `math` module.

```python
import math
```

We’ll need the `os.path` module to do file name manipulation.

```python
import os
```

All the Tkinter GUI package is added directly to the global namespace. The `tkFont` module is necessary to select fonts. The `Dialog` module makes it easy to create pop-up menus.

```python
from Tkinter import *
import tkFont
from Dialog import Dialog
```

`ScrolledList` is a compound widget, a combination of a `Listbox` and a vertical `Scrollbar`. For its documentation, see [*ScrolledList: A Tkinter scrollable list widget*](http://www.nmt.edu/tcc/help/lang/python/examples/scrolledlist/).

```python
from scrolledlist import ScrolledList
```

The other compound widget described elsewhere is `FontSelect`: a group of controls for selecting and displaying fonts. For documentation, see [*FontSelect: A Tkinter widget to select fonts*](http://www.nmt.edu/tcc/help/lang/python/examples/fontselect/).

```python
from fontselect import FontSelect
```
4.2. Manifest constants

The constant `MAX_PARAM` defines the maximum value for a color parameter. For example, black is represented as the triple (0,0,0), while white is represented as (65535,65535,65535).

```python
#-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
# Manifest constants
#-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
MAX_PARAM = 65535
```

The constant `N_PARAMS` is the number of parameters in a color model. This will always be three, but we use a manifest constant so that readers will not have to look at a literal “3” and try to figure out three of what.

```python
N_PARAMS = 3
```

The constant `MAX_BYTE` is the largest number that can be stored in a byte as an unsigned integer.

```python
MAX_BYTE = 255
```

Also global are the names of the fonts used throughout the application. `BUTTON_FONT` is used more or less everywhere; `MONO_FONT` is used where a monospaced font is required.

```python
BUTTON_FONT = ('times', 12)
MONO_FONT = ('lucidatypewriter', 14, 'bold')
```

5. main(): The main program

The graphical user interface as a whole is represented by the `Application` class. Calling the constructor initializes the application. See Section 11, “class Application: The outermost frame” (p. 18).

```python
# - - - - - m a i n
def main():
    """Main program.
    [ display a graphical application that allows the user to
test various fonts in various colors displayed on various
background colors ]
    """
    #-- 1 --
    # [ the screen := the screen with a graphical application
    # app := the graphical application ]
    app = Application()
```

Before actually running the application, we need to place the title of the application in the top-level window. This title will be displayed in the decorations applied by the user’s window manager. The `winfo_toplevel()` method returns the top-level window of the application; the `title()` method sets the title in that window. If we don’t do this, the default window title will be just “tk”.
6. class Color: A specific color

An instance of the Color class is used to represent a specific color. There are two obvious ways to represent the red, green, and blue components: as floating point numbers or as integers. The obvious way to use a float is to let 0.0 represent none and let 1.0 represent all of a component. However, the "#RRGGBB" representation is also common, and in that form each component is an integer in the range [0, MAX_PARAM]. An internal precision of 16 bits per color is plenty.

Here is the class interface:

```python
# - - - - - class Color
class Color:
    """Represents an arbitrary color.

   Exports:
    Color ( red, green, blue ): 
        [ (red is the red value as a float in [0.0,1.0] or as an int in [0,MAX_PARAM]) and 
        (green is the green value as a float in [0.0,1.0] or as an int in [0,MAX_PARAM]) and 
        (blue is the blue value as a float in [0.0,1.0] or as an int in [0,MAX_PARAM]) -> 
        return a new Color instance with those color values ]
```

Internally, a color is represented in the integer form. The three components are exported as read-only attributes.

```python
    .r: [ the red component as an int in [0,MAX_PARAM] ]
    .g: [ the green component as an int in [0,MAX_PARAM] ]
    .b: [ the blue component as an int in [0,MAX_PARAM] ]
```

The __str__() method defines the behavior of the built-in Python str() function when applied to instances of this class. See Section 6.3, "Color.__str__(): Convert to a string" (p. 10).

```python
    __str__(self):
        [ return self as a string "#RRGGBB" ]
```
The \_\_cmp\_\_() function is used to compare two colors to see if they are the same color. See Section 6.4, “Color.\_\_cmp\_\_(): Compare two colors” (p. 10).

```python
.huey
    \_\_cmp\_\_ (self, other):
        [ other is a Color instance ->
        if self's color name is less than other's ->
            return a negative number
        else if self's color name is greater than other's ->
            return a positive number
        else -> return 0 ]
```

### 6.1. Color.\_\_init\_\_(): Constructor

The constructor accepts each of the three color components as either a float or an integer. Floats are in the range [0.0, 1.0], but integers are in the range [0, MAX_PARAM]. The \_\_standardize() method translates the float form into the integer form; see Section 6.2, “Color.\_\_standardize(): Standardize color component representation” (p. 9).

```python
# - - - C o l o r . \_\_ i n i t \_\_      

def \_\_init\_\_ ( self, red, green, blue ):
    """Constructor for Color
    """

    # -- 1 --
    # [ if red is a float in [0.0, 1.0] ->
    #   self.r := red mapped onto [0, MAX_PARAM]
    # if red is an int in [0,MAX_PARAM] ->
    #   self.r := red ]
    self.r = self.\_\_standardize ( red )

    # -- 2 --
    # [ simile ]
    self.g = self.\_\_standardize ( green )

    # -- 3 --
    self.b = self.\_\_standardize ( blue )
```

### 6.2. Color.\_\_standardize(): Standardize color component representation

This method takes care of translating float-type arguments for color components into the standard integer range.

```python
# - - - C o l o r . \_\_ s t a n d a r d i z e

def \_\_standardize ( self, rawValue ):
    """Standardize representation of a color component.
```
6.3. Color.__str__(): Convert to a string

Converting each 16-bit color component to its 8-bit equivalent is a simple matter of shifting out the low-order bits.

```python
# --- C o l o r . _ _ s t r _ _

def __str__ ( self ):
    """Convert self to an X color name."
    ""
    r8 = self.r >> 8
g8 = self.g >> 8
b8 = self.b >> 8
    return "#%02X%02X%02X" % (r8, g8, b8)
```

6.4. Color.__cmp__(): Compare two colors

This method implements the usual Python comparison function. It can be used to test two colors for equality. The definition of inequality is pretty arbitrary, so we will just use their #RRGGBB color names to do the comparison.

```python
# --- C o l o r . _ _ c m p _ _

def __cmp__ ( self, other ):
    """Compare two colors.""
    return cmp ( str(self), str(other) )
```
7. class ColorModel: Base class for color models

The various color model we support will be concrete classes that share an interface defined in this base class.

```
# - - - - - c l a s s  C o l o r M o d e l

class ColorModel:
    """Base class for color models.

Exports:
    ColorModel ( modelName, labelList ): [ (modelName is the name of this model as a string) and (labelList is a sequence of three strings naming the parameters of this model) -> return a ColorModel object with those names ]
    .modelName: [ as passed to constructor, read-only ]
    .labelList: [ as passed to constructor, read-only ]

Functionally, a color model translates between a Color object (which internally uses RGB color space) and a trio of parameters that describe that color in the given model. The next two methods are the two conversions; see Section 7.2, “ColorModel.paramsToColor()” (p. 12) and Section 7.3, “ColorModel.colorToParams()” (p. 12).

    .paramsToColor ( params ): [ params is a sequence of three numbers in [0,MAX_PARAM] -> return that color in self's model as a Color instance ]
    .colorToParams ( color ): [ color is a Color instance -> return color's parameters in self's model as a sequence of three numbers in [0,MAX_PARAM] ]

The class also contains two static methods used to convert between numbers in [0,MAX_PARAM] and floats in [0.0, 1.0]. See Section 7.4, “ColorModel.normalize(): Normalize an integer color parameter” (p. 12) and Section 7.5, “ColorModel.discretize(): Discretize an integer color parameter” (p. 13).

    ColorModel.normalize(n): [ n is an int in [0,MAX_PARAM] -> return float(n)/MAX_PARAM ]
    ColorModel.discretize(n): [ n is a float in [0.0, 1.0] -> return int(n*MAX_PARAM) ]

```

7.1. ColorModel.__init__(): Constructor

All the constructor does is to store the arguments in the instance's namespace.

```
# - - -  C o l o r M o d e l . __ i n i t __
```
```python
def __init__(self, modelName, labelList):
    """Constructor for ColorModel."""
    self.modelName = modelName
    self.labelList = labelList

7.2. ColorModel.paramsToColor()

The polite way to implement a virtual method in Python (like this one) is to raise the NotImplemen-
tedError exception. For the interface, see Section 7, “class ColorModel: Base class for color mod-
els” (p. 11).

```python
# - - - C o l o r M o d e l . p a r a m s T o C o l o r

def paramsToColor(self, params):
    raise NotImplementedError, "ColorModel.paramsToColor"

```python
7.3. ColorModel.colorToParams()

A virtual method. For the interface, see Section 7, “class ColorModel: Base class for color mod-
els” (p. 11).

```python
# - - - C o l o r M o d e l . c o l o r T o P a r a m s

def colorToParams(self, params):
    raise NotImplementedError, "ColorModel.colorToParams"

```python
7.4. ColorModel.normalize(): Normalize an integer color parameter

This static method converts a color parameter in [0, MAX_PARAM] to a normalized float in [0.0, 1.0]. Values
outside the range are forced back into the range.

```python
# - - - C o l o r M o d e l . n o r m a l i z e
#
@staticmethod
def normalize(n):
    result = float(n)/MAX_PARAM
    if result < 0.0:
        return 0.0
    elif result > 1.0:
        return 1.0
    else:
        return result
normalize = staticmethod(normalize)
```
7.5. ColorModel.discretize(): Discretize an integer color parameter

This static method converts a color parameter in [0.0, 1.0] to a discrete integer in [0, MAX_PARAM]. Values outside the range are forced back into the range; this prevents potential unpleasantnesses such as a color parameter being rounded to hex 100 when it must be no more than hex FF.

```python
# - - - C o l o r M o d e l . d i s c r e t i z e

@staticmethod
def discretize(n):
    result = int(n*MAX_PARAM)
    if result < 0:
        return 0
    elif result > MAX_PARAM:
        return MAX_PARAM
    else:
        return result

discretize = staticmethod(discretize)
```

8. class HSVModel: Hue-saturation-value color model

The parameters of this color model are:

- Hue: an angle around the “color wheel”; angles 0° and 360° are red; 60° is yellow; 120° is green; 180° is cyan; 240° is blue; and 300° is magenta.
- Saturation: 1.0 for a fully saturated, pure color; reducing it toward zero moves toward white.
- Value: 1.0 for a fully saturated, pure color; reducing it toward zero moves toward black.

Here is the class interface.

```python
# - - - - - c l a s s H S V M o d e l
class HSVModel(ColorModel):
    """Represents the hue-saturation-value color model."
    ""
```

8.1. HSVModel.__init__() : Constructor

This constructor has nothing to do but call its parent constructor and pass it the names of the model and its parameters. Refer to Section 7, “class ColorModel: Base class for color models” (p. 11).

```python
# - - - H S V M o d e l . _ _ i n i t _ _

def __init__(self):
    ColorModel.__init__(self, "HSV",
                        ("hue", "saturation", "value")
```
8.2. HSVModel.paramsToColor()

This method converts a set of HSV parameters to a Color instance. This algorithm comes from Foley & van Dam's algorithm “HSV_To_RGB”; see Section 2, “References” (p. 4).

```python
# --- HSVModel.paramsToColor

def paramsToColor(self, params):
    """Convert the three HSV color parameters to a Color."
```

First we normalize the three color components to [0.0, 1.0]; see Section 7.4, “ColorModel.normalize()”: Normalize an integer color parameter” (p. 12).

```python
h = ColorModel.normalize(params[0])
s = ColorModel.normalize(params[1])
v = ColorModel.normalize(params[2])
```

Note that the Color() constructor will accept float values in [0.0, 1.0] as well as integer values in [0, MAX_PARAM).

```python
if s == 0.0:
    return Color(v, v, v)
```

Treat a hue of 1.0 or more as a hue of 0.0 due to wraparound. Otherwise, normalize the value of h to the half-open interval [0,6).

```python
if h >= 1.0:
    h = 0.0
else:
    h = h * 6.0
```

The math.modf() function returns a tuple (f, i) where f is the fractional part and i is the integral part. The rest follows Foley & van Dam’s algorithm, using the same names.

```python
(f,i) = math.modf(h)
i = int(i)
p = v * (1.0 - s)
q = v * (1.0 - s * f)
t = v * (1.0 - s * (1.0 - f))
```

```python
if i == 0:
    return Color(v, t, p)
elif i == 1:
    return Color(q, v, p)
elif i == 2:
    return Color(p, v, t)
elif i == 3:
    return Color(p, q, v)
elif i == 4:
    return Color(t, p, v)
else:
    return Color(v, p, q)
```
8.3. HSVModel.colorToParams()

This method converts a Color object into the three parameters of the HSV model, each in the range [0,MAX_PARAM]. The algorithm is Foley & van Dam’s “RGB_To_HSV”; see Section 2, “References” (p. 4).

```python
# -- - - - H S V M o d e l . c o l o r T o P a r a m s
def colorToParams( self, color):
    """Convert a Color to the three HSV parameters.
    """

    First we normalize the three RGB parameters to [0.0, 1.0]. See Section 7.4, “ColorModel.normalize()”: Normalize an integer color parameter” (p. 12).

    rNorm = ColorModel.normalize(color.r)
gNorm = ColorModel.normalize(color.g)
bNorm = ColorModel.normalize(color.b)

    The value parameter v is the maximum of the three values. We also calculate their minimum.

    v = maxColor = max ( rNorm, gNorm, bNorm )
    minColor = min ( rNorm, gNorm, bNorm )

    Next, calculate the saturation. If all the colors are zero, the saturation is zero.

    if maxColor == 0.0:
        s = 0.0
    else:
        s = ( maxColor - minColor ) / maxColor

    Next, find the hue. If the saturation is zero, arbitrarily use a value of zero (red) for the hue.

    if s == 0.0:
        h = 0.0  # Hue is undefined; use red arbitrarily
    else:
        delta = maxColor - minColor
        if rNorm == maxColor:  # Between Y and M
            h = ( gNorm - bNorm ) / delta
        elif gNorm == maxColor:  # Between C and Y
            h = 2.0 + ( bNorm - rNorm ) / delta
        else:  # Between M and C
            h = 4.0 + ( rNorm - gNorm ) / delta

        if h < 0.0:
            h = h + 6.0

        h = h / 6.0  # Normalize to [0,1]
```

These three cases are: colors between Y and M; colors between C and Y; and colors between M and C. Negative values are then rotated to the range [0,6]. Finally, the hue is normalized to [0.0, 1.0].

---

New Mexico Tech Computer Center  huey: Internal maintenance specification
Finally, return the three parameters, discretized back to $[0, \text{MAX\_PARAM}]$. See Section 7.5, “ColorModel.discretize(): Discretize an integer color parameter” (p. 13).

```
return ( ColorModel.discretize ( h ),
        ColorModel.discretize ( s ),
        ColorModel.discretize ( v ) )
```

9. class RGBModel: Red-green-blue color model

This is a concrete class inheriting from the base ColorModel class. It implements the red-green-blue color model. It doesn’t have much work to do, since the Color object uses the RGB model as its internal representation.

```
# --- --- c l a s s R G B M o d e l

class RGBModel(ColorModel):
    """Represents the red-green-blue color model."
    ""

9.1. RGBModel.__init__(): Constructor

All the constructor has to do is to call the parent class constructor, and supply the model’s name and the names of its three parameters. See Section 7, “class ColorModel: Base class for color models” (p. 11).

```
# --- R G B M o d e l . _ _ i n i t _ _

def __init__ ( self):
    """Constructor for RGBModel.""
    ColorModel.__init__ ( self, "RGB",
                          ("red", "green", "blue") )
```

9.2. RGBModel.paramsToColor()

This method takes a tuple $(red, green, blue)$ and returns a color as a Color instance. The ""*"" used inside the call to the Color() constructor takes advantage of one of Python’s features: that constructor expects three positional arguments, but since params is a three-element sequence, the ""*"" tells Python to allocate those three elements to the corresponding positional arguments.

```
# --- R G B M o d e l . p a r a m s T o C o l o r

def paramsToColor ( self, params):
    """Convert three RGB parameters to a Color."
    ""
    return Color ( *params )
```
9.3. RGBModel.colorToParams()

This method takes a Color instance and returns the three parameters of that color in the RGB model. No conversion is necessary, since the Color instance uses the RGB model internally.

```python
# - - - - R G B M o d e l . c o l o r T o P a r a m s
def colorToParams( self, color):
    """Convert a Color to the three RGB parameters."
    return ( (color.r, color.g, color.b))
```

10. class CMYModel: Cyan-magenta-yellow color model

This class implements the CMY color model. It is a concrete class that inherits from the ColorModel abstract class.

There is little more logic here than in the RGBModel class, since each parameter is the complement of one of the RGB parameters used internally by the Color class.

```python
# - - - - - c l a s s C M Y M o d e l
class CMYModel(ColorModel):
    """Represents the cyan-yellow-magenta color model."
```

10.1. CMYModel.__init__()

This constructor calls the parent class constructor and supplies the name of the model and the names of its three parameters. See Section 7, “class ColorModel: Base class for color models” (p. 11).

```python
# - - - C M Y M o d e l . _ _ i n i t _ _
def __init__( self):
    """Constructor for CMYModel."
    ColorModel.__init__( self, "CMY",
                        ("cyan", "yellow", "magenta") )
```

10.2. CMYModel.paramsToColor()

This method takes a triple \((c, m, y)\) containing the three CMY parameters of a color, and returns the equivalent color as a Color instance.

```python
# - - - C M Y M o d e l . p a r a m s T o C o l o r
def paramsToColor( self, params):
    """Convert CMY parameters to a Color."
    ```
Each parameter is the complement of one of the RGB colors: cyan is opposite red, magenta is opposite green, and yellow is opposite blue. Mathematically, we can convert a value by subtracting it from **MAX_PARAM**.

```python
cyan, magenta, yellow = params
red = MAX_PARAM - cyan
green = MAX_PARAM - magenta
blue = MAX_PARAM - yellow
return Color(red, green, blue)
```

### 10.3. CMYModel.colorToParams()

Converting a **Color** instance to cyan, magenta, and yellow values is straightforward, as each of the CMY colors is the complement of the corresponding RGB colors stored in attributes of the **Color** instance. To complement a number in [0,MAX_PARAM], we subtract that number from **MAX_PARAM**.

```python
# - - - C M Y M o d e l . c o l o r T o P a r a m s

def colorToParams(self, color):
    """Convert a color to CMY parameters.
    """
    cyan = MAX_PARAM - color.r
    magenta = MAX_PARAM - color.g
    yellow = MAX_PARAM - color.b
    return (cyan, magenta, yellow)
```

### 11. class Application: The outermost frame

This class encapsulates the entire graphical application. It inherits from Tkinter’s **Frame** class, which makes **Application** for all intents and purposes a Tkinter widget.

```python
# - - - - - c l a s s A p p l i c a t i o n

class Application(Frame):
    """Contains the entire application.
    """
    Contained widgets:
    .menuBar:    [ a MenuBar widget ]
    .namePicker: [ a NamePicker widget ]
    .adjuster:   [ an Adjuster widget ]
    .swatch:     [ a Swatch widget ]

    Grid plan:
    0 1 2
    +----------------+----------------+----------------+
    0 | .__menuBar     |                |
    +----------------+----------------+----------------+
    1 | .__namePicker  | .__adjuster    | .__swatch      |
```
This class is a container for four compound widgets. The MenuBar widget across the top edge contains general controls for the application. The bulk of the space is taken up by the three major sections of the application: the NamePicker widget for selecting colors by name; the Adjuster widget for adjusting colors and selecting color models; and the Swatch widget displaying the selected colors and fonts, and including font selection controls.

At this level, there are only two important linkages between the contained widgets:

- Whenever the user selects a color in the NamePicker widget, that widget tells the Adjuster widget to display that color.
- Whenever the user adjusts the current color in the Adjuster widget, or when that widget is set by an outside call to change its color, it tells the Swatch widget to change to the new color as well.

The current text and background colors reside inside the Adjuster widget.

### 11.1. Application.__init__(): Constructor

The constructor creates the application window and populates it with its four contained widgets.

```python
#- - - Application. __init__ - - -

def __init__(self):
    """Constructor for Application."
    
Because this class inherits from Frame, the first order of business is to call the parent class’s constructor. The first argument is self, the new instance’s namespace. The second argument is the parent window; passing None has the effect of creating a new root window.

Calling the .grid() method on self is necessary to register the new window; if this isn’t done, the application will not appear on the screen.

```python
#-- 1 --
# [ self := self as a new root window Frame ]
Frame.__init__(self, None)
self.grid()
```

Widget creation is handled by Section 11.2, “Application.__createWidgets(): Set up the widgets” (p. 19).

```python
#-- 2 --
# [ self := self with all widgets created and gridded ]
self.__createWidgets()
```

### 11.2. Application.__createWidgets(): Set up the widgets

This method creates and grids all the widgets for the Application class.
def __createWidgets ( self ):
    """Create and grid all widgets."
    
    The MenuBar compound widget occupies the full width of row 0. The sticky parameter makes it adhere to the left (west) side of its grid cell. See Section 12, “class MenuBar: General controls” (p. 21).

Next we add the NamePicker widget. The first argument to a widget constructor is always the parent widget, self in this case. The second argument is a callback function that is called whenever the user selects a color. See Section 13, “class NamePicker: Select colors by name” (p. 33).

The Adjuster widget constructor also gets a second argument containing the name of a callback function, which will be called whenever the user adjusts the current color. See Section 15, “class Adjuster: Color adjustment and color model selection” (p. 58).

Last to go in is the Swatch widget; see Section 20, “class Swatch: Font and color samples” (p. 87).
11.3. Application.__nameHandler(): Name picker handler

This callback is called by the NamePicker widget whenever the user picks a color by name. The argument is a Color object representing the selected color. It passes the new color to the .set() method of the Adjuster widget, so the new color can be shown there. See Section 15.2, “Adjuster.set(): Change the color” (p. 60).

```python
# - - - A p p l i c a t i o n . _ _ n a m e H a n d l e r
def __nameHandler(self, newColor):
    """Handler for the NamePicker widget.
    [ newColor is a Color instance ->
      self := self with its Adjuster displaying newColor ]
    ""
    self.__adjuster.set(newColor)
```

11.4. Application.__adjustHandler(): Adjuster handler

This method is called whenever the Adjuster widget changes the color that it shows. It gets two arguments. The first argument is 0 for background, 1 for text color. The second argument is the new color. It passes that new color on to the Swatch widget to be displayed; for these methods, see Section 20, “class Swatch: Font and color samples” (p. 87).

```python
# - - - A p p l i c a t i o n . _ _ a d j u s t H a n d l e r
def __adjustHandler(self, isText, newColor):
    """Handler for the Adjuster widget.
    [ newColor is a Color instance ->
      if isText ->
        self := self with its Swatch displaying its
text in color (newColor)
      else ->
        self := self with its Swatch displaying its
background in color (newColor) ]
    ""
    if isText:
        self.__swatch.setTextColor(newColor)
    else:
        self.__swatch.setBgColor(newColor)
```

12. class MenuBar: General controls

This class is a compound widget containing all the general controls for the program:

- The Quit button.
- The Help button.
Because this widget is designed to take up a fairly thin slice across the top of the application, these controls are gridded in a horizontal row. The class inherits from Tkinter’s Frame class, so it is effectively a compound widget—just a big widget with other widgets inside.

```python
# - - - - - c l a s s   M e n u   B a r

class MenuBar(Frame):
    """Compound widget containing general program controls.

    Contained widgets:
    .__helpButton: [ a Menubutton that displays help text ]
    .__quitButton: [ a Button that terminates the application ]

    Grid plan:
    0 1
    +---------------+---------------+
    0 | .__helpButton | .__quitButton |
    +---------------+---------------+
    """

12.1. MenuBar.__init__(): Constructor

The constructor starts by calling its parent class’s constructor, which makes it a proper Frame widget. It is not gridded in the parent; that is the caller’s job. Then the contained widgets are created and gridded within self.

```python
# - - - M e n u   B a r .   _ _ i n i t _ _

def __init__ ( self, parent ):
    """Constructor for MenuBar""

    #-- 1 --
    # [ self := self as a Frame, ungridded ]
    Frame.__init__ ( self, parent )

    The Help button is a menubutton, but there are many submenus. This whole drop-down menu structure is built by Section 12.2, “MenuBar.__createHelp(): Create the Help menu” (p. 23).

```python
#-- 2 --
# [ self := self with a new Menubutton added and
#     gridded, leading to the help menu cascades ]
self.__helpButton = self.__createHelp()
self.__helpButton.grid ( row=0, column=0 )

The Quit button is linked to the .quit() method that exists on all Tkinter widgets.

```python
#-- 3 --
# [ self := self with a Quit button added and gridded
#     that terminates execution ]
self.__quitButton = Button ( self, text='Quit',
    font=BUTTON_FONT, command=self.quit )
self.__quitButton.grid ( row=0, column=1 )
```
12.2. `MenuBar.__createHelp()`: Create the Help menu

The visible Help button is a Menubutton widget that leads to a cascade of submenus. When the user reaches an actual text item, that item appears in a pop-up Dialog.

```python
# - - -   M e n u B a r . __ c r e a t e H e l p

def __createHelp( self):
    """Create the help menubutton and its cascade.

    [ self is a Frame ->
    self := self with a new Menubutton added but not
    gridded, leading to the help text
    return that new Menubutton ]
    """
```

Here is an outline showing the structure of the help cascade, with links to the methods that set them up.

- Selecting a standard color: Section 12.3, “MenuBar.__cascadeNamePicker()” (p. 24).
- Typing in a standard color name: Section 12.4, “MenuBar.__helpTyping()” (p. 25).
- Picking a standard color name: Section 12.5, “MenuBar.__helpPicking()” (p. 26).
- Adjusting colors: Section 12.6, “MenuBar.__cascadeAdjuster()” (p. 26).
- Selecting background or text color: Section 12.7, “MenuBar.__helpReadout()” (p. 27).
- Color models and color space: Section 12.8, “MenuBar.__helpModelSelector()” (p. 27).
- The HSV color model: Section 12.9, “MenuBar.__helpHSV()” (p. 27).
- The RGB color model: Section 12.10, “MenuBar.__helpRGB()” (p. 28).
- The CMY color model: Section 12.11, “MenuBar.__helpCMY()” (p. 28).
- Using the color sliders: Section 12.12, “MenuBar.__helpSliders()” (p. 28).
- Viewing colors and fonts: Section 12.13, “MenuBar.__cascadeViewing()” (p. 29).
- Selecting a font: Section 12.15, “MenuBar.__cascadeFonts()” (p. 30).
  - Selecting a font family: Section 12.16, “MenuBar.__helpFontFamily()” (p. 30).
  - Changing the font size: Section 12.17, “MenuBar.__helpFontSize()” (p. 31).
  - Changing font weight and slant: Section 12.18, “MenuBar.__helpFontStyle()” (p. 31).
- Importing colors into other applications: Section 12.19, “MenuBar.__helpImporting()” (p. 31).
- Who made this tool? Section 12.20, “MenuBar.__helpAuthor()” (p. 32).

First we create the two interlinked widgets that make up a drop-down menu: the Menubutton, which appears all the time as the Help button, and the Menu widget, which contains the cascades that appear when the user clicks the Menubutton.

```python
#-- 1 --
#
# [ mb := a new Menubutton with self as its parent ]
mb = Menubutton( self, font=BUTTON_FONT,
               relief=RAISED,
               text="Help")
```

Creating the mutual linkage between the Menubutton and its first-level Menu is not completely obvious. The upward linkage is established by making the Menubutton the parent of the Menu. The downward linkage is established by setting the Menu’s menu attribute to the Menubutton.
There are two kinds of choices on a drop-down Menu:

- A **cascade** is a choice that leads to another level of drop-down menu. This type of choice is identified by a right-pointing arrowhead.

- Terminal (non-cascade) choices are identified by a simple text label without a right-pointing arrowhead.

The finished `Menubutton` is returned to the caller for gridding.

#### 12.3. `MenuBar.__cascadeNamePicker()`

This method adds one selection to the first-level help menu, along with its subcommand cascade.
"Add the help cascade for picking colors by name

[ menu is a Menu widget ->
  menu := menu with choices added for help topics
  about picking colors by name ]
"

A “cascade” is a choice on a drop-down Menu that leads to another Menu.

---

Both choices here are terminal choices. For a discussion of cascade and terminal choices, see Section 12.2, “MenuBar.__createHelp(): Create the Help menu” (p. 23). For the methods that create the pop-up menus, see Section 12.4, “MenuBar.__helpTyping()” (p. 25) and Section 12.5, “MenuBar.__helpPicking()” (p. 26).

12.4. MenuBar.__helpTyping()

This method creates a pop-up menu for help with typing standard color names.

---

All our dialogs are stereotyped, so their creation is handled by Section 12.21, “MenuBar.__dialog(): Pop up a dialog widget” (p. 32).
If the color name you type is not known to the system, you will get a popup dialog menu.

12.5. **MenuBar.__helpPicking()**

This Dialog popup explains the color picker.

```python
# - - - M e n u B a r . __ h e l p P i c k i n g

def __helpPicking ( self):
    
    self.__dialog ( "Help: Picking a standard color name",
                "Under the label 'Or click on a name:' you will see a "
                "list of all the standard color names. Click on the color "
                "name to select that color." )
```

12.6. **MenuBar.__cascadeAdjuster()**

Adds to the help menu a subtopic “Adjust a color” along with its cascade. This is structurally similar to Section 12.3, “MenuBar.__cascadeNamePicker()” (p. 24).

```python
# - - - M e n u B a r . __ c a s c a d e A d j u s t e r

def __cascadeAdjuster ( self, menu):
    
    menu.add_cascade ( menu=select,
        label="Adjusting a color"
    )

For the methods that created the sub-cascades, see Section 12.7, “MenuBar.__helpReadout()” (p. 27), Section 12.8, “MenuBar.__helpModelSelector()” (p. 27), Section 12.9, “MenuBar.__helpHSV()” (p. 27), Section 12.10, “MenuBar.__helpRGB()” (p. 28), Section 12.11, “MenuBar.__helpCMY()” (p. 28), and Section 12.12, “MenuBar.__helpSliders()” (p. 28).
```

```python
#-- 2 --
# [ select := select with pop-ups added that relate to color adjustment ]

select.add_command ( label='Selecting background or text color',
    command=self.__helpReadout )

select.add_command ( label='Color models and color space',
    command=self.__helpModelSelector )
```
select.add_command ( label='The HSV color model',
                     command=self.__helpHSV )
select.add_command ( label='The RGB color model',
                     command=self.__helpRGB )
select.add_command ( label='The CMY color model',
                     command=self.__helpCMY )
select.add_command ( label='Using the color sliders',
                     command=self.__helpSliders )

12.7. MenuBar.__helpReadout()

def __helpReadout ( self ):
    """Help for selecting bg/text color
    ""
    self.__dialog ( 'Help: Selecting background or text color',
                   "When the radiobuttons labeled 'Background color' is set, "
                   "you can select the background color by name or by "
                   "adjusting the three color sliders." "\n\tWhen 'Text color' is selected, you can select the "
                   "color of the displayed text." )

12.8. MenuBar.__helpModelSelector()

def __helpModelSelector ( self ):
    """Help for selecting a color model.
    ""
    self.__dialog ( 'Help: Selecting a color model',
                   "A 'color model' is a way of describing a specific color as "
                   "a set of three values from 0 and 255. Use the radiobuttons "
                   "under 'Select a color model' to pick one of the color "
                   "models." )

12.9. MenuBar.__helpHSV()

def __helpHSV ( self ):
    """Help for the HSV color model.
    ""
    self.__dialog ( 'Help: The HSV color model',
In the HSV color model, color is specified by these three parameters:

- **Hue** controls the base color. As you drag the Hue slider, the color goes through red, yellow, green, cyan, blue, magenta, and then back to red again.
- **Saturation** controls color intensity. Drag the Saturation slider down for pale colors, drag it up for intense colors.
- **Value** controls brightness. Drag the Value slider down to darken a color, up to lighten it.

12.10. **MenuBar.__helpRGB()**

```python
def __helpRGB(self):
    """Help for the RGB color model.
    """
    self.__dialog('Help: The RGB color model',
        "In the RGB color model, color is specified by three parameters: red, green, and blue. These are called the 'additive primary colors;' if you drag all three sliders all the way up, you get white. RGB color mixing is used in display screens and also in stage lighting."
)
```

12.11. **MenuBar.__helpCMY()**

```python
def __helpCMY(self):
    """Help for the CMY color model.
    """
    self.__dialog('Help: The CMY color model',
        "In the CMY color model, color is specified by three parameters: cyan, magenta, and yellow. These are called the 'subtractive primary colors;' if you drag all three sliders all the way up, you get black. CMY color mixing is used in printing and in color darkroom work."
)
```

12.12. **MenuBar.__helpSliders()**

```python
def __helpSliders(self):
    """Help for the color sliders.
    """
    self.__dialog('Help: The color sliders',
        "To adjust a color, drag any of the three sliders with the "
```
12.13. MenuBar.__cascadeViewing()

This cascade is slightly more involved because it has a second-level cascade in it.

```python
def __cascadeViewing ( self, menu ):
    """Add the help cascade for viewing colors

    [ menu is a Menu widget ->
      menu := menu with choices added for help topics
      about viewing colors ]
    ""

First we create a new cascade and add it to the parent menu.

```---
1
# [ menu := menu with a new cascade added
# select := that cascade ]
select = Menu ( menu )
menu.add_cascade ( menu=select,
                 label="Viewing colors and fonts" )
```

There are only two subcomponents: help on the color swatch area, and the subcascade for font help. See Section 12.14, “MenuBar.__helpSwatch()” (p. 29) and Section 12.15, “MenuBar.__cascadeFonts()” (p. 30).

```---
2
# [ select := select with a pop-up added about the
#    color swatch ]
select.add_command ( command=self.__helpSwatch,
                    label="The color swatch" )
```

```---
3
# [ select := select with a cascade added for font help ]
self.__cascadeFonts ( select )
```


```---
--- MenuBar.__helpSwatch

def __helpSwatch ( self ):
    """Help for the color swatch.
    ""
    self.__dialog ( "Help: The color swatch",
                    "The top right corner of the window displays your "
```
12.15. MenuBar.__cascadeFonts()

This cascade contains subtopics about font selection.

```python
# - - - M e n u B a r . __ c a s c a d e F o n t s
def __cascadeFonts ( self, menu ):
    """Add the font help cascade.
    
    [ menu is a Menu widget ->
    menu := menu with help topics added on fonts ]
    """
    #-- 1 --
    # [ menu := menu with a new cascade added
    #   select := that cascade ]
    select = Menu ( menu )
    menu.add_cascade ( menu=select,
            label="Selecting a font" )

    For subtopics, see Section 12.16, "MenuBar.__helpFontFamily()" (p. 30),
    Section 12.17, "MenuBar.__helpFontSize()" (p.31), and Section 12.18,
    "MenuBar.__helpFontStyle()" (p.31).

    #-- 2 --
    # [ select := select with a pop-up added about font
    #   families ]
    select.add_command ( command=self.__helpFontFamily,
            label="Selecting a font family" )

    #-- 3 --
    # [ simile ]
    select.add_command ( command=self.__helpFontSize,
            label="Changing the font size" )

    #-- 4 --
    select.add_command ( command=self.__helpFontStyle,
            label="Changing font weight and slant" )
```

12.16. MenuBar.__helpFontFamily()

```python
# - - - M e n u B a r . __ h e l p F o n t F a m i l y
def __helpFontFamily ( self ):
    """Help for font family selection.
    """
    self.__dialog ( "Help: Selecting a font family",
            "Under 'Click to select font family:' is a scrollable "
```
“list containing the names of all the locally installed "
"font families. Click on any of these names to select "
"that family." )

12.17. MenuBar.__helpFontSize()

```python
# --- MenuBar._helpFontSize

def __helpFontSize ( self ):
    ""
        Help for changing font size.
        ""
    self.__dialog ( "Help: Changing the font size",
       "To change the size of the text, enter a number in the "
       "field labeled 'Size:'. This number gives the font size "
       "in points. You can also enter a negative number to "
       "specify a size in pixels. After changing the size, "
       "either press the Enter key or click on the 'Set size' "
       "button. " )
```

12.18. MenuBar.__helpFontStyle()

```python
# --- MenuBar._helpFontStyle

def __helpFontStyle ( self ):
    ""
        Popup for changing font weight and slant.
        ""
    self.__dialog ( "Help: Changing the font weight and slant",
       "To change from normal to boldface type or back to normal, "
       "click the 'Bold' button."
       "\n\tTo change from normal to italic type or back to "
       "normal, click the 'Italic' button." )
```

12.19. MenuBar.__helpImporting()

```python
# --- MenuBar._helpImporting

def __helpImporting ( self ):
    ""
        Pop-up help for importing color names.
        ""
    self.__dialog ( "Help: Using colors in other applications",
       "To use one of the standard color names in an application, "
       "you can just use the name as it appears in the list of "
       "standard colors."
       "\n\tYou can also use the hexadecimal form of the color "
       "name that appears under #RRGGBB for either the background "
       "or text (foreground) color." )
```
12.20. MenuBar.__helpAuthor()

```python
# - - - M e n u B a r . __ h e l p A u t h o r

def __helpAuthor ( self ):
    """Pop-up for author credit.
    """
    self.__dialog ( "Help: Who made this tool?",
    "%s was written by John W. Shipman (john@nmt.edu) "
    "of the New Mexico Tech Computer Center, Socorro, NM "
    "87801."
    "\n\tThis is version %s."
    "\n\tFor documentation, see:
    %s"
    (PROGRAM_NAME, EXTERNAL_VERSION,
    "http://www.nmt.edu/tcc/help/lang/python/examples/huey/ims/" )
```

12.21. MenuBar.__dialog(): Pop up a dialog widget

This method creates a pop-up Dialog widget. The structure is the same except for the frame title (the title argument) and the help text (the text argument).

```python
# - - - M e n u B a r . __ d i a l o g

def __dialog ( self, title, text):
    """Pop up a Dialog widget.

    [ (title is the frame title as a string) and
    (text is the help text as a multiline string containing
    newlines and tab characters) ->
    desktop := desktop with a Dialog widget displaying
    title and text ]
    """
```

The Dialog widget for pop-up menus comes from the Dialog.py module, part of Tkinter. Unfortunately, this widget is poorly documented; the author learned to use it by reading examples and reading its source code.

- The title argument appears as the window title.
- The default argument specifies which button is the default choice; in our case, it's the only button, button number zero.
- The strings argument is a tuple of button titles. We need only one button, a Dismiss button that gets rid of the pop-up.
- The bitmap argument specifies which pictograph appears on the pop-up; the "info" bitmap is a big gray question mark.
- The text argument is the body text for the pop-up. This text may include newlines ("\n") and tabs ("\t"). Long lines will be folded automatically. Note that Python automatically merges adjacent string constants.

```python
dialog ( self, default=0, strings=('Dismiss',), bitmap='info',
title=title, text=text )
```
One drawback of these prefabricated Dialog widgets is that you can’t override the ugly font. This is a candidate for replacement someday. Note that the widget grabs the focus, so the user can’t do anything back in the main window until they dismiss the pop-up.

13. **class NamePicker: Select colors by name**

This class is a compound widget that gives the user two different ways to select a color by its name. They may type the name directly into an Entry widget. Most of the vertical space below this Entry contains a ScrolledList that displays a prefabricated list of color names; the user may instead click on any of these.

```python
# ------------
class NamePicker

class NamePicker(Frame):
    """A compound widget for selecting a color by its name.

    Exports:
    NamePicker ( parent, callback=None ):
        [ (parent is a Frame) and
          (callback is a function or None) ->
          parent := parent with a new NamePicker widget
          added but not gridded, that calls callback
          whenever a color is selected
          return that NamePicker widget ]

    Contained widgets:
    ___entryLabel: [ a Label widget that labels self.__entry ]
    ___entry: [ an Entry widget for entering a color name ]
    ___pickLabel: [ a Label widget that labels self.__pickList ]
    ___pickList: [ a ScrolledList widget with color names ]

    Grid plan: Vertical stacking.

    Private attributes include ___callback, the callback function, and a StringVar control variable linked to the ___entry widget.

    State/Invariants:
    ___parent: [ as passed to constructor ]
    ___callback: [ as passed to constructor ]
    ___entryVar:
        [ a StringVar control variable for self.__entry ]

    """
```

The width of the Entry field, a manifest constant, is a class variable.

```python
NAME_WIDTH = 20
```
13.1. NamePicker.__init__(): Constructor

The constructor initializes the parent Frame, stores its argument, sets up a control variable for the Entry, and then creates all the widgets. See Section 13.2, “NamePicker.__createWidgets__(): Create all widgets” (p. 34).

```python
# - - - NamePicker.__init__

def __init__( self, parent, callback=None):
    """Constructor for the NamePicker compound widget."
    
    #-- 1 --
    # [ self := self as a Frame ]
    Frame.__init__ ( self, parent )

    #-- 2 --
    self.__parent = parent
    self.__callback = None

    #-- 3 --
    # [ self := self with all contained widgets created and
    #   gridded ]
    self.__createWidgets()

    #-- 4 --
    self.__callback = callback
```

13.2. NamePicker.__createWidgets__(): Create all widgets

For gridding information, see Section 13, “class NamePicker: Select colors by name” (p. 33).

```python
# - - - NamePicker.__createWidgets__

def __createWidgets(self):
    """Create all widgets.""
    
    #-- 1 --
    # [ self := self with a new Label to label the color
    #   name entry field
    #   Label := that Label ]
    self.__entryLabel = Label ( self,
                               font=BUTTON_FONT, text="Type a color name:" )
    rowx = 0
    self.__entryLabel.grid ( row=rowx, column=0, sticky=W )

    #-- 2 --
    # [ self := self with a new Entry whose control variable
    #   is self.__entryVar, and which calls self.__callback
```
Placing the label for the color pick list is straightforward. However, creating the color pick list is pretty involved, so that logic is located in Section 14, “class PickList: The color names pick list” (p. 37).

```
13.3. NamePicker.__entryHandler(): Handler for the Enter key

This method is called when the user presses the Enter key in the color name field. Assuming the color name is valid, we then call the class’s callback to notify observers that the color has changed.
```

```
def __entryHandler ( self, event ):
    """Handle the Enter key in the color name field.

    [ event is an Event from self._entry ->
      if self.__entryVar is a valid color name ->
        call self.__callback with a Color representing that name
      else ->
        pop up a Dialog informing the user that the color name is not valid ]
    ""
```
The .get() method on self.__entryVar retrieves the text from the color name field. We then pass the name on to Section 13.4, “NamePicker.__setName(): Try to convert a color name to a color” (p. 36).

```python
#-- 1 --
# [ colorName := text from self.__entryVar ]
colorName = self.__entryVar.get()

#-- 2 --
# [ if self.__entryVar is a valid color name ->
#   call self.__callback with a Color representing
#   that name
# else ->
#   pop up a Dialog informing the user that the
#   color name is not valid ]
self.__setName ( colorName )
```

13.4. NamePicker.__setName(): Try to convert a color name to a color

The purpose of this method is to convert a color name into a numeric value, and thence into a Color instance.

```python
#--- NamePicker.__setName ---

def __setName ( self, colorName ):
    """Convert a color name to RGB values as a Color instance.
    
    [ colorName is a string ->
    if colorName is defined in either Tkinter or our
    color pick list ->
    call self.__callback and pass it that color as
    a Color instance
    else ->
    pop up a Dialog window complaining about the
    undefined color name ]
    ""
```

In theory, a properly installed X Window system will have a file named rgb.txt that lists all the standard color names. In theory, any application should be able to use a color name from that file, and the window system will recognize it. In practice, the presence of that file is not guaranteed, and even if does exist, we have no assurance that the set of color names accepted by the window system is the same as the contents of the file.

In case the rgb.txt file is missing, this program contains a data structure which represents the contents of one version of that file.

However, there is another source for translating color names to RGB values. Every Tkinter widget has a method named .winfo_rgb() which takes a color name and returns a triple \((R, G, B)\) where each value is in the range \([0, \text{MAX\_PARAM}]\).

So, our liberal policy will be to try the latter technique first. If that fails, we can still use the list that appears in the pick list to convert names to RGB values. That way, if the two sources differ, we implement
the union of their values. There is a discussion of the Dialog pop-up in Section 12.4, "MenuBar.__helpTyping()" (p. 25). See also Section 14.3, “PickList.lookupName(): Convert a color name to a color value” (p. 45).

```python
#-- 1 --
# [ if (Tkinter recognizes (colorName) as a color) or
#   (colorName) is defined in self.__pickList) ->
#   rgb := that color as a tuple (r, g, b) of values
#     in [0,MAX_PARAM]
# else ->
#   pop up a Dialog complaining about the undefined
#   color name
#   return ]
try:
    rgb = self.winfo_rgb ( colorName )
except:
    rgb = self.__pickList.lookupName ( colorName )
if not rgb:
    Dialog ( self, title="Message", bitmap="info",
            default=0, strings=("Dismiss"),
            text="Color '%s' is undefined." % colorName )
    return

All that remains is to convert rgb to a Color and call the callback.

```

```python
#-- 2 --
# [ if self.__callback is not None ->
#   call self.__callback with a Color made from rgb
# else -> ]
if self.__callback is not None:
    self.__callback ( Color ( *rgb ) )
```

13.5. NamePicker.__pickListHandler(): Somebody clicked on a color

This method is called when the user clicks on a color name in the PickList. That object passes us that color as a Color instance, and we pass it on to our callback.

```python
#-- NamePicker.__pickListHandler --

def __pickListHandler ( self, color ):
    """Handler for self.__pickList.
    """
    if self.__callback is not None:
        self.__callback ( Color ( *color ) )
```

14. class PickList: The color names pick list

This class is a compound widget containing the list of standard color names. It responds to user clicks by calling its callback function with a Color instance. It also contains a translation table for converting
its color names to color values. For an important discussion of the two methods of converting color names, see Section 13.4, “NamePicker.__setName()”: Try to convert a color name to a color” (p. 36).

In GUI terms, it is a frame containing only a single widget, a ScrolledList. This widget is a compound widget from the author's widget library. It is a combination of a Tkinter Listbox widget with a vertical Scrollbar. For particulars of its call and implementation, see Section 4.1, “Imports” (p. 6).

To populate this list with standard color names, there are two possible sources. On most Unix boxen, there is a file named rgb.txt that defines the names and values. Depending on the age of the install, it may be in one of these paths; the first one reflects the more recent practice:

```
/usr/share/X11
/usr/lib/X11
```

However, just so this program isn’t useless in the absence of the file, this class contains an internal version of rgb.txt as a fallback measure.

Here is the class interface:

```python
# - - - - - class PickList
class PickList(Frame):
    """Compound widget for the color pick list.
    Exports:
    PickList ( parent, callback=None ):
        (parent is a Frame) and
        (callback is a function or None) ->
        parent := parent with a new PickList widget added
        but not gridded
        return that new widget
    .lookupName ( colorName ):
        [ colorName is a string ->
            if colorName matches a color in self's list,
            case-insensitive ->
            return the corresponding value as a Color instance
            else -> return None ]
    Contained widgets:
    .scrolledList:
        [ a scrolledlist.ScrolledList containing self's color
          names ]
    Grid plan: Only one widget
    
    State/Invariants:
    .callback: [ as passed to the constructor ]
```

This method allows the caller to translate a color name to a Color instance; see Section 14.3, “PickList.lookupName()”: Convert a color name to a color value” (p. 45).
For internal data structures, we need to be able to look up colors both by name and by their position in the pick list. The lists `.colorList` and `.nameList` are by position: the n-th element in `self.colorList` is the color displayed in the n-th line of the pick list, and the n-th element of `self.nameList` is the name of the color displayed in that line.

```python
__colorList:
    [ a list of Color instances such that self.colorList[i]
      is the value of the color displayed in the (i)th line
      of self.scrolledList, as a Color instance ]
__nameList:
    [ a list of color names such that self.nameList[i]
      is the name of the color displayed in the (i)th line
      of self.scrolledList ]
```

The obvious data structure for looking up color names is a dictionary. To make the lookup case-insensitive, the keys of this dictionary are the color names, uppercased. Each value in the dictionary is a tuple (index, color, name), where:

- The `index` is an integer that sorts colors into the original order in their source, whether that source is the rgb.txt file or the built-in default color list.
  This index is important in the process of removing redundant colors from the list while retaining their original order; see Section 14.11, “PickList.__cleanColorMap(): Remove redundant colors” (p. 53).
- The `color` is the color value as an instance of the Color class.
- The `name` is the color name as a string.

```python
__colorMap:
    [ a dictionary whose keys are the color names in self,
      uppercased, and each value is a tuple (index, color,
      name) where index orders the colors in their original
      source order, color is a Color instance, and name is
      the color name as a string ]
```

### 14.1. PickList class constants

The class has a few manifest constants. First is the name of the standard color names file.

```python
COLOR_NAMES_FILE = "rgb.txt"
```

Next, a list of all the pathnames in which to look for that file.

```python
PATHS_LIST = [ '/usr/share/X11', '/usr/lib/X11' ]
```

The physical width and height of the list box are given in characters and lines, respectively.

```python
NAME_WIDTH = 20
NAME_LINES = 22
```
14.2. PickList: Default color list

The default color list is also a class variable. It’s rather long, but it must be declared here.

This list was generated by reformating a 2001-era version of the stock rgb.txt file. Each line of that file is represented here as one string. The first three bytes of the string are the red, green, and blue parameters, each in the range [0,255], and the balance of the string is the actual color name.

```
DEFAULT_COLORS = [
    '\xFF\xFA\xFAsnow',  # snow
    '\xF5\xF5\xF5WhiteSmoke',  # WhiteSmoke
    '\xFF\xFA\x00FloralWhite',  # FloralWhite
    '\xF8\xF8\xFFGhostWhite',  # GhostWhite
    '\xF5\xF5\xFCWhiteSmoke',  # WhiteSmoke
    '\xFF\xF4\xF4Gainsboro',  # Gainsboro
    '\xF0\xF0\xF0White',  # White
    '\xFF\xFF\xFFWhite',  # White
    '\xF4\xE9\xF4Linens',  # Linen
    '\xFF\xFF\x80LightGreen',  # LightGreen
    '\xFF\xFF\xFFWhite',  # White
    '\xF8\xFF\xFFWhite',  # White
    '\xF2\xF8\xFFLightYellow',  # LightYellow
    '\xF0\xFF\xFFSkyBlue',  # SkyBlue
    '\x87\x8B\xFFDeepSkyBlue',  # DeepSkyBlue
    '\x75\x70\x75SeaGreen',  # SeaGreen
    '\x22\x8B\x22ForestGreen',  # ForestGreen
    '\xBD\xB7\x6BDarkKhaki',  # DarkKhaki
    '\xEE\xE8\xAAPaleGoldenrod',  # PaleGoldenrod
    '\xFA\xFA\xD2LightGoldenrodYellow',  # LightGoldenrodYellow
    '\xFF\xFF\xE0LightYellow',  # LightYellow
    '\xFF\xD7\x00Gold',  # Gold
]
```
<table>
<thead>
<tr>
<th>Name</th>
<th>Hex Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>goldenrod</td>
<td>#DAA520</td>
</tr>
<tr>
<td>DarkGoldenrod</td>
<td>#B8860B</td>
</tr>
<tr>
<td>RosyBrown</td>
<td>#BC8F8F</td>
</tr>
<tr>
<td>IndianRed</td>
<td>#CD5C5C</td>
</tr>
<tr>
<td>Sienna</td>
<td>#A0522D</td>
</tr>
<tr>
<td>Burlywood</td>
<td>#DEB887</td>
</tr>
<tr>
<td>B3wheat</td>
<td>#5563E0</td>
</tr>
<tr>
<td>C8tan</td>
<td>#B22222</td>
</tr>
<tr>
<td>Firebrick</td>
<td>#E9967A</td>
</tr>
<tr>
<td>AlightSalmon</td>
<td>#FFA07A</td>
</tr>
<tr>
<td>DarkOrange</td>
<td>#FF8C00</td>
</tr>
<tr>
<td>LightCoral</td>
<td>#F08080</td>
</tr>
<tr>
<td>OrangeRed</td>
<td>#FF4500</td>
</tr>
<tr>
<td>Salmon</td>
<td>#FFA07A</td>
</tr>
<tr>
<td>LightSalmon</td>
<td>#FF8C00</td>
</tr>
<tr>
<td>DarkSalmon</td>
<td>#FFA07A</td>
</tr>
<tr>
<td>DarkViolet</td>
<td>#A020F0</td>
</tr>
<tr>
<td>Pink</td>
<td>#FFC0CB</td>
</tr>
<tr>
<td>LightPink</td>
<td>#FFE4C4</td>
</tr>
<tr>
<td>PaleVioletRed</td>
<td>#DB7093</td>
</tr>
<tr>
<td>Maroon</td>
<td>#80000D</td>
</tr>
<tr>
<td>MediumVioletRed</td>
<td>#C71585</td>
</tr>
<tr>
<td>BlueViolet</td>
<td>#8A2BFF</td>
</tr>
<tr>
<td>MediumPurple</td>
<td>#9370DB</td>
</tr>
<tr>
<td>Violet</td>
<td>#C040F0</td>
</tr>
<tr>
<td>Purple</td>
<td>#FF00FF</td>
</tr>
<tr>
<td>Magenta</td>
<td>FF0080</td>
</tr>
<tr>
<td>SaddleBrown</td>
<td>#A52A2A</td>
</tr>
<tr>
<td>Tomato</td>
<td>#FF6900</td>
</tr>
<tr>
<td>DarkOrangeRed</td>
<td>#FF4500</td>
</tr>
<tr>
<td>LightCoral</td>
<td>#F08080</td>
</tr>
<tr>
<td>OrangeRed</td>
<td>#FF4500</td>
</tr>
<tr>
<td>PaleGoldenrod</td>
<td>#B8860B</td>
</tr>
<tr>
<td>BlanchedAlmond</td>
<td>#F2E0C4</td>
</tr>
<tr>
<td>Wheat</td>
<td>#F5DCB3</td>
</tr>
<tr>
<td>LemonChiffon</td>
<td>#FFEBEE</td>
</tr>
<tr>
<td>PaleGoldenrod</td>
<td>#B8860B</td>
</tr>
<tr>
<td>Gainsboro</td>
<td>#B8860B</td>
</tr>
<tr>
<td>LightSandyBrown</td>
<td>#FFFAF0</td>
</tr>
<tr>
<td>Sand</td>
<td>#FFEC82</td>
</tr>
<tr>
<td>LightGoldenrod</td>
<td>#FFD700</td>
</tr>
<tr>
<td>Honey</td>
<td>#F0E68C</td>
</tr>
<tr>
<td>Khaki</td>
<td>#BDB769</td>
</tr>
<tr>
<td>LightSkyBlue</td>
<td>#87CEEB</td>
</tr>
<tr>
<td>MediumSpringGreen</td>
<td>#00CED1</td>
</tr>
<tr>
<td>MediumSeaGreen</td>
<td>#66CDAA</td>
</tr>
<tr>
<td>Dolomite</td>
<td>#48D1CC</td>
</tr>
<tr>
<td>DarkKhaki</td>
<td>#8B4513</td>
</tr>
<tr>
<td>LightSkyBlue</td>
<td>#87CEEB</td>
</tr>
<tr>
<td>MediumSpringGreen</td>
<td>#00CED1</td>
</tr>
<tr>
<td>MediumSeaGreen</td>
<td>#66CDAA</td>
</tr>
<tr>
<td>DodgerBlue</td>
<td>#AFB424</td>
</tr>
<tr>
<td>MediumSpringGreen</td>
<td>#00CED1</td>
</tr>
<tr>
<td>MediumSeaGreen</td>
<td>#66CDAA</td>
</tr>
<tr>
<td>MediumOrchid</td>
<td>#BA55D3</td>
</tr>
<tr>
<td>DarkOrchid</td>
<td>#8D4F8B</td>
</tr>
<tr>
<td>DarkVioletBlue</td>
<td>#9400D3</td>
</tr>
<tr>
<td>MediumPurple</td>
<td>#9370DB</td>
</tr>
<tr>
<td>SteelBlue</td>
<td>#5870D0</td>
</tr>
<tr>
<td>SteelBlue</td>
<td>#5870D0</td>
</tr>
<tr>
<td>Moccasin</td>
<td>#B000D5</td>
</tr>
<tr>
<td>HotPink</td>
<td>#FF69B4</td>
</tr>
<tr>
<td>MediumOrchid</td>
<td>#BA55D3</td>
</tr>
<tr>
<td>MediumPurple</td>
<td>#9370DB</td>
</tr>
<tr>
<td>Azure</td>
<td>#4682B4</td>
</tr>
<tr>
<td>MediumBlue</td>
<td>#0000CD</td>
</tr>
<tr>
<td>RoyalBlue</td>
<td>#4169E1</td>
</tr>
<tr>
<td>MediumBlue</td>
<td>#0000CD</td>
</tr>
<tr>
<td>RoyalBlue</td>
<td>#4169E1</td>
</tr>
<tr>
<td>MediumBlue</td>
<td>#0000CD</td>
</tr>
<tr>
<td>RoyalBlue</td>
<td>#4169E1</td>
</tr>
<tr>
<td>MediumBlue</td>
<td>#0000CD</td>
</tr>
<tr>
<td>RoyalBlue</td>
<td>#4169E1</td>
</tr>
<tr>
<td>MediumBlue</td>
<td>#0000CD</td>
</tr>
<tr>
<td>RoyalBlue</td>
<td>#4169E1</td>
</tr>
<tr>
<td>MediumBlue</td>
<td>#0000CD</td>
</tr>
<tr>
<td>RoyalBlue</td>
<td>#4169E1</td>
</tr>
<tr>
<td>MediumBlue</td>
<td>#0000CD</td>
</tr>
<tr>
<td>RoyalBlue</td>
<td>#4169E1</td>
</tr>
<tr>
<td>MediumBlue</td>
<td>#0000CD</td>
</tr>
<tr>
<td>RoyalBlue</td>
<td>#4169E1</td>
</tr>
<tr>
<td>MediumBlue</td>
<td>#0000CD</td>
</tr>
<tr>
<td>RoyalBlue</td>
<td>#4169E1</td>
</tr>
<tr>
<td>MediumBlue</td>
<td>#0000CD</td>
</tr>
<tr>
<td>RoyalBlue</td>
<td>#4169E1</td>
</tr>
<tr>
<td>MediumBlue</td>
<td>#0000CD</td>
</tr>
<tr>
<td>RoyalBlue</td>
<td>#4169E1</td>
</tr>
<tr>
<td>MediumBlue</td>
<td>#0000CD</td>
</tr>
<tr>
<td>RoyalBlue</td>
<td>#4169E1</td>
</tr>
<tr>
<td>MediumBlue</td>
<td>#0000CD</td>
</tr>
<tr>
<td>RoyalBlue</td>
<td>#4169E1</td>
</tr>
<tr>
<td>MediumBlue</td>
<td>#0000CD</td>
</tr>
<tr>
<td>RoyalBlue</td>
<td>#4169E1</td>
</tr>
<tr>
<td>MediumBlue</td>
<td>#0000CD</td>
</tr>
<tr>
<td>RoyalBlue</td>
<td>#4169E1</td>
</tr>
<tr>
<td>MediumBlue</td>
<td>#0000CD</td>
</tr>
<tr>
<td>RoyalBlue</td>
<td>#4169E1</td>
</tr>
<tr>
<td>MediumBlue</td>
<td>#0000CD</td>
</tr>
<tr>
<td>RoyalBlue</td>
<td>#4169E1</td>
</tr>
<tr>
<td>MediumBlue</td>
<td>#0000CD</td>
</tr>
<tr>
<td>RoyalBlue</td>
<td>#4169E1</td>
</tr>
<tr>
<td>MediumBlue</td>
<td>#0000CD</td>
</tr>
<tr>
<td>RoyalBlue</td>
<td>#4169E1</td>
</tr>
<tr>
<td>MediumBlue</td>
<td>#0000CD</td>
</tr>
<tr>
<td>RoyalBlue</td>
<td>#4169E1</td>
</tr>
<tr>
<td>MediumBlue</td>
<td>#0000CD</td>
</tr>
<tr>
<td>RoyalBlue</td>
<td>#4169E1</td>
</tr>
<tr>
<td>MediumBlue</td>
<td>#0000CD</td>
</tr>
<tr>
<td>RoyalBlue</td>
<td>#4169E1</td>
</tr>
<tr>
<td>MediumBlue</td>
<td>#0000CD</td>
</tr>
<tr>
<td>RoyalBlue</td>
<td>#4169E1</td>
</tr>
<tr>
<td>MediumBlue</td>
<td>#0000CD</td>
</tr>
<tr>
<td>RoyalBlue</td>
<td>#4169E1</td>
</tr>
<tr>
<td>MediumBlue</td>
<td>#0000CD</td>
</tr>
<tr>
<td>RoyalBlue</td>
<td>#4169E1</td>
</tr>
<tr>
<td>MediumBlue</td>
<td>#0000CD</td>
</tr>
<tr>
<td>RoyalBlue</td>
<td>#4169E1</td>
</tr>
<tr>
<td>MediumBlue</td>
<td>#0000CD</td>
</tr>
<tr>
<td>RoyalBlue</td>
<td>#4169E1</td>
</tr>
<tr>
<td>Color Name</td>
<td>Hex Code</td>
</tr>
<tr>
<td>----------------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>Blue3</td>
<td>\x00\x00\xCD</td>
</tr>
<tr>
<td>DodgerBlue1</td>
<td>\x1E\x90\xFF</td>
</tr>
<tr>
<td>SteelBlue1</td>
<td>\x63\x88\xFF</td>
</tr>
<tr>
<td>LightSkyBlue1</td>
<td>\x00\x0F\xFF</td>
</tr>
<tr>
<td>SlateGray1</td>
<td>\xB0\xE2\xFF</td>
</tr>
<tr>
<td>LightBlue1</td>
<td>\xE0\xFF\xFF</td>
</tr>
<tr>
<td>PaleTurquoise1</td>
<td>\xBB\xFF\xFF</td>
</tr>
<tr>
<td>turquoise1</td>
<td>\x00\x0F\xFF</td>
</tr>
<tr>
<td>LightCyan1</td>
<td>\x97\xFF\xFF</td>
</tr>
<tr>
<td>OliveDrab1</td>
<td>\xC0\xFF\x3E</td>
</tr>
<tr>
<td>khaki1</td>
<td>\xFF\xFF\x8F</td>
</tr>
<tr>
<td>LightGoldenrod1</td>
<td>\xFF\xFF\xB4</td>
</tr>
<tr>
<td>LightYellow1</td>
<td>\xFF\xFF\xCD</td>
</tr>
<tr>
<td>DarkSlateGray1</td>
<td>\x9A\xFF\x9A</td>
</tr>
<tr>
<td>SpringGreen1</td>
<td>\x00\xFF\x00</td>
</tr>
<tr>
<td>chartreuse1</td>
<td>\x7F\xFF\x00</td>
</tr>
<tr>
<td>OliveDrab2</td>
<td>\x8D\xFF\x8D</td>
</tr>
<tr>
<td>khaki2</td>
<td>\xCD\xFF\x70</td>
</tr>
<tr>
<td>LightGoldenrod2</td>
<td>\xFF\xFF\xB4</td>
</tr>
<tr>
<td>LightYellow2</td>
<td>\xFF\xFF\xCD</td>
</tr>
<tr>
<td>DarkSlateGray2</td>
<td>\x9A\xFF\x9A</td>
</tr>
<tr>
<td>SpringGreen2</td>
<td>\x00\xFF\x00</td>
</tr>
<tr>
<td>chartreuse2</td>
<td>\x7F\xFF\x00</td>
</tr>
<tr>
<td>OliveDrab3</td>
<td>\x8D\xFF\x8D</td>
</tr>
<tr>
<td>khaki3</td>
<td>\xCD\xFF\x70</td>
</tr>
<tr>
<td>LightGoldenrod3</td>
<td>\xFF\xFF\xB4</td>
</tr>
<tr>
<td>LightYellow3</td>
<td>\xFF\xFF\xCD</td>
</tr>
<tr>
<td>DarkSlateGray3</td>
<td>\x9A\xFF\x9A</td>
</tr>
<tr>
<td>SpringGreen3</td>
<td>\x00\xFF\x00</td>
</tr>
<tr>
<td>chartreuse3</td>
<td>\x7F\xFF\x00</td>
</tr>
<tr>
<td>OliveDrab4</td>
<td>\x8D\xFF\x8D</td>
</tr>
<tr>
<td>khaki4</td>
<td>\xCD\xFF\x70</td>
</tr>
<tr>
<td>LightGoldenrod4</td>
<td>\xFF\xFF\xB4</td>
</tr>
<tr>
<td>LightYellow4</td>
<td>\xFF\xFF\xCD</td>
</tr>
<tr>
<td>DarkSlateGray4</td>
<td>\x9A\xFF\x9A</td>
</tr>
<tr>
<td>SpringGreen4</td>
<td>\x00\xFF\x00</td>
</tr>
<tr>
<td>chartreuse4</td>
<td>\x7F\xFF\x00</td>
</tr>
<tr>
<td>OliveDrab5</td>
<td>\x8D\xFF\x8D</td>
</tr>
<tr>
<td>khaki5</td>
<td>\xCD\xFF\x70</td>
</tr>
<tr>
<td>LightGoldenrod5</td>
<td>\xFF\xFF\xB4</td>
</tr>
<tr>
<td>LightYellow5</td>
<td>\xFF\xFF\xCD</td>
</tr>
<tr>
<td>DarkSlateGray5</td>
<td>\x9A\xFF\x9A</td>
</tr>
<tr>
<td>SpringGreen5</td>
<td>\x00\xFF\x00</td>
</tr>
<tr>
<td>chartreuse5</td>
<td>\x7F\xFF\x00</td>
</tr>
<tr>
<td>OliveDrab6</td>
<td>\x8D\xFF\x8D</td>
</tr>
<tr>
<td>khaki6</td>
<td>\xCD\xFF\x70</td>
</tr>
<tr>
<td>LightGoldenrod6</td>
<td>\xFF\xFF\xB4</td>
</tr>
<tr>
<td>LightYellow6</td>
<td>\xFF\xFF\xCD</td>
</tr>
<tr>
<td>DarkSlateGray6</td>
<td>\x9A\xFF\x9A</td>
</tr>
<tr>
<td>SpringGreen6</td>
<td>\x00\xFF\x00</td>
</tr>
<tr>
<td>chartreuse6</td>
<td>\x7F\xFF\x00</td>
</tr>
</tbody>
</table>
14.3. **PickList.lookupName()**: Convert a color name to a color value

This method uses the internal `self.__colorMap` dictionary to lookup color names and, if found, return the corresponding color value as a `Color` instance.

```
# - - -  P i c k L i s t . l o o k u p N a m e

def lookupName ( self, colorName ):
    
    """Look up a color name."
    """
    #-- 1 --
    # [ colorKey := colorName, uppercased ]
    colorKey = colorName.upper()

    #-- 2 --
    # [ if colorKey is a key in self.__colorMap ->
    #   return the corresponding value
    #  else ->
    #   return None ]
    try:
        index, color, name = self.__colorMap[colorKey]
        return color
    except KeyError:
        return None
```

14.4. **PickList.__init__()**: Constructor

First the constructor calls the parent constructor to make it into a Frame. Then it creates and grids the PickList widget.
```python
# - - - P i c k L i s t . _ _ i n i t _ _

def __init__ ( self, parent, callback=None):
    """Constructor for the color name PickList.""
    #-- 1 --
    # [ parent := parent with a new Frame added
    #  self := that Frame ]
    Frame.__init__ ( self, parent )

Next we initialize the internal data structures.

    #-- 2 --
    self.__colorList = []
    self.__nameList = []
    self.__colorMap = {}
    self.__callback = None

Now create and grid the sole widget. See Section 4.1, “Imports” (p. 6) for documentation on the ScrolledList widget.

    #-- 3 --
    # [ self := self with a new ScrolledList widget added
    #  and gridded
    #  self.__scrolledList := that widget ]
    self.__scrolledList = ScrolledList ( self,
        width=self.NAME_WIDTH, height=self.NAME_LINES,
        callback=self.__pickHandler )
    self.__scrolledList.listbox["font"] = BUTTON_FONT
    self.__scrolledList.grid ( row=0, column=0 )

The logic that populates the widget with color names, and sets up the corresponding internal tables, is Section 14.5, “PickList.__addColors(): Populate the color list” (p. 47).

    #-- 4 --
    # [ self.__scrolledList := self.__scrolledList populated
    #   with non-redundant color names from the standard
    #   file if found, defaulting to an internal color list
    # self.__colorMap += as invariant
    # self.__colorList += as invariant
    # self.__nameList += as invariant ]
    self.__addColors()

Now that everything is set up, enable the callback mechanism.

    #-- 5 --
    self.__callback = callback
```
14.5. `PickList.__addColors()`: Populate the color list

The purpose of this method is to place a standard set of color names into `self.__scrolledList` and the internal data structures `self.__colorMap`, `self.__colorMap` and `self.__colorList`, that describe that same set of colors.

```python
# -- - - P i c k L i s t _ _ a d d C o l o r s

def __addColors ( self ):
    """Populate the color set

    [ if a readable, valid self.COLOR_NAMES_FILE exists in
    one of the directories named in self.PATHS_LIST ->
    self.__scrolledList := self.pickList with the color
    names added from that file in the same order
    self.__colorMap := as invariant from that file
    self.__colorList := as invariant from that file
    self.__nameList := as invariant from that file
    else ->
    self.__scrolledList := self.pickList with the color
    names added from the internal default list
    self.__colorMap := as invariant from that file
    self.__colorList := as invariant from that file
    self.__nameList := as invariant from that file ]
    """

We will try to find the file first; see Section 14.6, “PickList.__findColorsFile(): Read the rgb.txt file” (p. 48).

```python
# -- 1 --
# [ if self.COLOR_NAMES_FILE names a readable, valid
# rgb.txt file in one of the directories named in
# self.PATHS_LIST ->
#    self.__colorMap := as invariant from that file
# else ->
#    self.__colorMap := an empty dictionary ]
self.__findColorsFile()
```

If that didn’t work, set up `self.__colorList` from `self.DEFAULT_COLORS`; see Section 14.10, “PickList.__useDefaultColors(): Set up a default color list” (p. 52).

```python
# -- 2 --
# [ if self.__colorMap is empty ->
#    self.__colorMap += as invariant from self.DEFAULT_COLORS
# else -> I ]
if len(self.__colorMap) == 0:
    self.__useDefaultColors()
```

Because of historical differences in language and naming conventions, the color files usually have a large number of redundant colors. For example, in our local rgb.txt, colors “LightSlateGray”, “LightSlateGrey”, “light slate gray”, and “light slate grey” all refer to exactly the same color. So, to make it easier for the user to scroll through the list, we’ll remove redundant colors from `self.__colorMap`; see Section 14.11, “PickList.__cleanColorMap(): Remove redundant colors” (p. 53).
Now that the color set is finalized, populate the ScrolledList widget and build the self.__nameList and self.__colorList lists. Because the values in self.__colorMap are tuples starting the original color index, sorting these values will put them back in their original order.

```python
#-- 4 --
# [ valueList := values from self.__colorMap, sorted ]
valueList = self.__colorMap.values()
valueList.sort()

#-- 5 --
# [ valueList is a list of tuples (index, color, name) ->
#   self.__scrolledList += names from valueList in the
#   same order
#   self.__colorList += colors from valueList in the
#   same order ]
for index, color, name in valueList:
    self.__scrolledList.append( name )
    self.__nameList.append( name )
    self.__colorList.append( color )
```

### 14.6. PickList.__findColorsFile(): Read the rgb.txt file

This method searches various directories in hopes that there is a standard colors file there. If so, it sets up self.__colorList with the color values and self.__nameList with the corresponding names.

```python
#--- PickList.__findColorsFile ---

def __findColorsFile ( self ):
    """Try to find and read the standard colors file.

    [ if self.COLOR_NAMES_FILE names a readable, valid
    rgb.txt file in one of the directories named in
    self.PATHS_LIST ->
    self.__colorMap := as invariant from that file
    else ->
    self.__colorMap := an empty dictionary ]
    ""

    #-- 1 --
inFile = None

    #-- 2 --
    # [ if self.PATHS_LIST names a directory that contains
    #   a readable file named self._COLOR_NAMES_FILE ->
    #   inFile := that file opened for reading
    #   else -> I ]
```
for dir in self.PATHS_LIST:
    fileName = os.path.join ( dir, self.COLOR_NAMES_FILE )
    try:
        inFile = open ( fileName )
        break
    except IOError:
        pass

If we couldn’t find the file, we set self.__colorMap to an empty list to signal failure, and return to the caller.

```python
#-- 2 --
if inFile is None:
    self.__colorMap = {}
    return
```

We’re not guaranteed success yet—the file might not be in the correct format. See Section 14.7, “PickList.__readColorsFile(): Process the rgb.txt file” (p. 49) for details of the file format and the process of reading it.

```python
#-- 3 --
# [ if inFile is a valid colors file ->
#   self.__colorMap := as invariant from that file
# else ->
#   self.__colorMap := an empty dictionary ]
self.__readColorsFile ( inFile )
```

### 14.7. PickList.__readColorsFile(): Process the rgb.txt file

The purpose of this method is to attempt to read the rgb.txt file and use it to set up the list of standard colors.

```python
#-- P i c k L i s t . _ _ r e a d C o l o r s F i l e

def __readColorsFile ( self, inFile ):
    """Try to read the file of standard colors.
    
    [ inFile is a readable file ->
    if inFile is a valid colors file ->
    self.__colorMap := as invariant from that file
    else ->
    self.__colorMap := an dictionary ]
    ""
```

The format of the colors file is fairly simple:

- Ignore comment lines beginning with a bang (!).
- Here is a typical line defining a color:

```
255 239 213 papaya whip
```

The first three fields are numbers in [0,255] describing the red, green, and blue color values, respectively. The rest of the line contains the color name.
The color files aren’t huge, so we use `.readlines()` to return the contents of the file as a list of strings. This makes it easier to number the lines in the next step.

```python
#-- 1 --
# [ lineList := the lines from inFile as a list of strings ]
lineList = inFile.readlines()
```

Later steps need to know the original order of the colors, so we use the index in `lineList` as the induction variable of the loop to process the lines. If even so much as one line is not valid, we clear `self.__colorMap` and return. See Section 14.8, “`PickList.__readColorLine()`: Process one line from rgb.txt” (p. 50).

```python
#-- 2 --
# [ if all the non-comment lines in inFile are valid ->
#   self.__colorMap := as invariant from those lines
#   else ->
#   self.__colorMap := an empty list ]
try:
    for index in range ( len ( lineList ) ):
        #-- 2 body --
        # [ if lineList[index] is a comment -> I
        #   else if lineList[index] is a valid color line ->
        #     self.__colorMap += an entry mapping the
        #     upperscased color name |-> a tuple (index,
        #     the color from that line as a Color instance),
        #     the name from that line)
        #   else ->
        #     self.__colorMap := {}
        #     return ]
        self.__colorMap = {}
        self.__readColorLine ( index, lineList[index] )
except IOError:
    self.__colorMap = {}
```

### 14.8. `PickList.__readColorLine()`: Process one line from rgb.txt

This method reads and processes one line from the `rgb.txt` file. For a description of the format, see Section 14.7, “`PickList.__readColorsFile()`: Process the rgb.txt file” (p. 49).

```python
#-- PickList.__readColorLine

def __readColorLine ( self, index, rawLine ):
    """Process one line from the rgb.txt file.

    [ (index is a nonnegative integer) and
      (rawLine is a string) ->
      if rawLine is a comment -> I
      else if rawLine is valid ->
      self.__colorMap += an entry mapping the
      upperscased color name |-> a tuple (index,
      the color from that line as a Color instance),
      the name from that line)
```

50  huey: Internal maintenance specification  New Mexico Tech Computer Center
First we remove any trailing whitespace. This will take care of any newline character.

```python
else -> raise IOError
```

If the line is a comment, return.

```python
#-- 2 --
if line.startswith('!'):
    return
```

Python’s string `.split()` method does a nice job of breaking the line into fields. In this call, there are two arguments. The first argument is the delimiter; passing `None` to this argument has the effect of using any contiguous string of whitespace characters as a delimiter. The second argument specifies an upper limit to the number of fields removed from the front of the string. By passing a value of 3 to this argument, we never get more than four fields; this will avoid splitting up color names that contain embedded spaces.

```python
#-- 3 --
# [ fieldList := line split on whitespace regions, but never
#   more than the first four ]
fieldList = line.split( None, 3 )
```

Now for some validity tests. There should be exactly four fields. Also, the first three must contain strings that can be converted to integers. For conversion of the integer values, see Section 14.9, “PickList.__convert8()” (p. 52).

```python
#-- 4 --
if len(fieldList) < 4:
    raise IOError, "rgb.txt lines must have four fields"
else:
    colorName = fieldList[-1]
```

```python
#-- 5 --
# [ if the first three elements of fieldList can be
#   converted to integers in [0,255] ->
#   red8 := int(fieldList[0])
#   green8 := int(fieldList[1])
#   blue8 := int(fieldList[2])
#   else -> raise IOError ]
red8 = self.__convert8( fieldList[0] )
green8 = self.__convert8( fieldList[1] )
blue8 = self.__convert8( fieldList[2] )
```

The `Color` class constructor expects values in the range `[0,MAX_PARAM]`. One possibility is just to multiply the values by 256, and that’s probably sufficiently accurate. A rejected possibility was to duplicate the value, e.g., `0x80` becomes `0x8000`, and `0xff` becomes `0xffff`. If anyone ever convinces me that it makes a difference, I’ll change it.
Finally, add a new entry to `self.__colorMap` for this color.

```python
#-- 7 --
colorKey = colorName.upper()
colorTuple = (index, color, colorName)
self.__colorMap [ colorKey ] = colorTuple
```

### 14.9. PickList.__convert8(): Convert an 8-bit number

This method is used to process one of the 8-bit numbers from the `rgb.txt` file.

```python
#--- PickList.__convert8

def __convert8 ( self, s ):
    """Convert a string to an 8-bit number.
    
    [ if s is the string representation of a number in [0,255] ->
      return s as an integer
    else -> raise IOError ]
    """
    #-- 1 --
    # [ if s can be converted to an integer ->
    #   result := s converted to an integer
    # else -> raise IOError ]
    try:
        result = int ( s )
    except ValueError:
        raise IOError, "Bad color value: '%s'" % s
    #-- 2 --
    return result
```

### 14.10. PickList.__useDefaultColors(): Set up a default color list

This method is called when the `rgb.txt` file is missing or invalid. It sets up the class's color list from the default color set in Section 14.2, “PickList: Default color list” (p. 40).

```python
#--- PickList.__useDefaultColors

def __useDefaultColors ( self):
    """Set up self's colors from the default list.
    
    [ self.__colorMap += as invariant from
```
This method sets up the color pick list from our internal, default list of colors.

The `self.DEFAULT_COLORS` list is a list of strings with the same values as lines from `rgb.txt`: 8-bit values for red, green, and blue, followed by the color name.

As we set up `self.__colorMap`, the 8-bit values are shifted left eight bits to scale them to the `[0, MAX_PARAM]` range expected by the `Color()` constructor.

```python
#-- 1 --
for index in range(len(self.DEFAULT_COLORS)):
    # [ self.DEFAULT_COLORS[index] is a string with the
    #   8-bit red, green, and blue values followed by the
    #   color name ->
    #   self.__colorMap += an entry mapping the uppercased
    #   color name -> a tuple (index, a Color instance
    #   made from those color values, the color name) ]
    #-- 1.1 --
    colorString = self.DEFAULT_COLORS[index]
    red = ord(colorString[0]) << 8
    green = ord(colorString[1]) << 8
    blue = ord(colorString[2]) << 8
    color = Color(red, green, blue)
    colorName = colorString[3:]
    #-- 1.2 --
    colorKey = colorName.upper()
    colorTuple = (index, color, colorName)
    #-- 1.3 --
    self.__colorMap[colorKey] = colorTuple
```

### 14.11. PickList.__cleanColorMap(): Remove redundant colors

This method removes duplicate color names. There are two mechanisms that create these:

- Spacing conventions: “light slate gray” is the same color as “LightSlateGray”.
- Spelling conventions: “LightSlateGray” is the same color as “LightSlateGrey”.

Because of these variations, there may be as many as four names for the same color: “LightSlateGray” is also the same color as “light slate grey”.

We'll arbitrary prefer the intercapitalized forms such as “LightSlateGray” to the versions with embedded spaces such as “light slate gray”. Equally arbitrarily, because the author happened to be born in the USA, we'll cast out the “grey” variants.

```python
#-- -- PickList.__cleanColorMap

def __cleanColorMap(self):
    """Remove redundant colors from the color map.
    [ self.__colorMap := self.__colorMap with redundant colors
```
The logic for removing less preferred names is in Section 14.12, “PickList.__nameCleaner(): Remove redundant color names” (p. 54).

### 14.12. PickList.__nameCleaner(): Remove redundant color names

This method takes as an argument one of the keys to `self.__colorMap`, an uppercased color name. Its purpose is to find any other entries in `self.__colorMap` whose keys are less preferred names, and remove them.

```
def __nameCleaner ( self, colorName ):
    """Remove any redundant names dominated by colorName."
    [ colorName is a key in self.__colorMap ->
      self.__colorMap := self.__colorMap with any entries
      removed whose keys are less-preferred alternatives
to colorName ]
    """
```

For general remarks on the preference rules for color names, see Section 14.11, “PickList.__cleanColorMap(): Remove redundant colors” (p. 53).

Because intercapitalized color names such as "DodgerBlue" are stored under their uppercase equivalents such as "DODGERBLUE", we need to pull out the original color name, or we won't know where spaces are to be inserted. If this lookup fails, that's okay, because the name may already have been purged. For example, the name “DarkGray” might purge “dark grey”, and then later when “dark gray” comes along it will also try to purge “dark grey”, but find that it's already gone.
There are two transformations on `colorName` to convert to the key for a less-preferred version. The `__lowerize()` method converts intercapitalized names like “DodgerBlue” to their space-embedded equivalents such as “dodger blue”. The `__anglicize()` method converts the American spelling “gray” to the English spelling “grey”. However, we must also purge the list of names to which both transformations are applied.

```
#-- 2 --
# [ anglican := colorName with 'gray' -> 'grey' ]
anglican = self.__anglicize ( origName )

#-- 2 --
# [ if self.__colorMap has a key (anglican) ->
#   self.__colorMap := self.__colorMap with that key's
#   entry removed
# else -> I ]
self.__purgeName ( origName, anglican )

#-- 3 --
# [ if self.__colorMap has a key self.__lowerize(colorName) ->
#   self.__colorMap := self.__colorMap with that key's
#   entry removed
# else -> I ]
self.__purgeName ( origName, self.__lowerize ( origName ) )

#-- 4 --
# [ if self.__colorMap has a key self.__lowerize(anglican) ->
#   self.__colorMap := self.__colorMap with that key's
#   entry removed
# else -> I ]
self.__purgeName ( origName, self.__lowerize ( anglican ) )
```

### 14.13. PickList.__anglicize(): Convert “gray” to “grey”

This method returns its string argument with the first occurrence of "gray" replaced by "grey", if there is one.

```
#--- PickList.__anglicize

def __anglicize ( self, name ):
    """Anglicize a name.
    
    [ name is a string ->
      if name contains "gray" ->
        return name with the first occurrence of "gray"
        replaced by "grey"
      else -> return name ]
    """
    where = name.find ( "gray" )
    if where >= 0:
        return name[:where] + "grey" + name[where+4:]
    else:
```
14.14. PickList.__lowerize(): Convert intercapitalized names to lowercase

This method converts an intercapitalized name such as “DarkSlateGray” to its lowercased, space-separated form such as “dark slate gray”.

```python
# - - - P i c k L i s t . _ _ l o w e r i z e

def __lowerize ( self, name ):
    """Convert an intercapitalized color name to lowercase form.
    [ name is a string -> return name with all letters lowercased, and single
      spaces inserted at each lowercase->uppercase transition ]
    ""

    First we break the name up into a list of individual characters. Then we work through the gaps between characters, adding a space after that gap if there was a lowercase-uppercase transition there. Finally, the str.join() method reassembles the pieces into a string.

    #-- 1 --
    # [ letters := a list of the characters in name ]
    letters = list ( name )

    #-- 2 --
    # [ letters := letters with a space prefixed to any
    #   uppercase letter preceded by a lowercase letter ]
    for i in range ( 1, len ( letters ) ):
        if ( letters[i-1].islower() and
            letters[i].isupper() ):
            letters[i] = " " + letters[i]

    #-- 3 --
    return "".join ( letters )
```

14.15. PickList.__purgeName(): Remove one redundant color name

This method takes two arguments. The goodName argument is a color name; badName is a less-preferred alternative.

```python
# - - - P i c k L i s t . _ _ p u r g e N a m e

def __purgeName ( self, goodName, badName ):
    """If badName is redundant for goodName, remove it.
```
First we test to see if `goodName` and `badName` are the same. For example, the `__lowerize()` transformation on "red" yields "red". In this case we return.

```python
#-- 1 --
if goodName == badName: return
```

Next, test to see if `badName` is even in the color map. If not, return. If it is there, retrieve the colors for both names.

```python
#-- 2 --
# [ if badName is a key in self.__colorMap -> 
#   goodColor := self's color for goodName
#   badColor := self's color for badName
# else -> 
#   return ]
badColor = self.lookupName ( badName )
if badColor is None: return
else:
  goodColor = self.lookupName ( goodName )
```

If the colors are the same, we can safely remove the less preferred one. If they don’t match, just for safety, retain the bad color.

```python
#-- 3 --
# [ if badColor == goodColor -> 
#   self.__colorMap := self.__colorMap with the entry 
# for badColor removed 
# else -> I ]
if badColor == goodColor:
  badKey = badName.upper()
  del self.__colorMap [ badKey ]
```

### 14.16. PickList.__pickHandler(): Someone clicked on a color name

This method is called when the user clicks on a color name. The argument is the line number within the pick list. It passes the corresponding color through to the class’s callback function.
15. class Adjuster: Color adjustment and color model selection

This class includes all the widgets down the center third of the application:

- A ColorReadout widget shows the text and background colors in #RRGGBB form, and has a pair of radiobuttons for selecting whether the rest of this widget displays the text (foreground) color or the background color.

- A ModelSelector widget has radiobuttons that let the user select HSV, RGB, or CMY color space.

- A ColorSliders widget has three Tkinter Scale widgets the user can slide in order to make fine adjustments to a color.

This Adjuster widget class, then, is really the central “brain” of the application. Inside this object are stored the current background and text colors.

Here is the class interface. As with all compound widgets, it inherits its widgetness from the Frame class.
The next two methods return the current colors; see Section 15.3, “Adjuster.textColor(): Return the current text color” (p. 60) and Section 15.4, “Adjuster.bgColor(): Return the current text color” (p. 60).

```
.textColor():
    [ return the current text color as a Color instance ]
.bgColor():
    [ return the current background color as a Color instance ]
```

The .set() method changes the currently displayed color; see Section 15.2, “Adjuster.set(): Change the color” (p. 60).

```
.set ( color ):
    [ color is a Color instance ->
      if self is displaying the text color ->
        self's text color := color
      else ->
        self's background color := color ]
```

The .isText() method tests whether the sliders are displaying the text color or the background color; see Section 15.5, “Adjuster.isText(): Which color is being displayed?” (p. 61).

```
isText():
    [ if self is displaying the text color ->
      return True
    else -> return False ]
```

One might ask, instead of making isText() a method call, why not make it simply an exported read-only Boolean? It is made a method strictly for the convenience of the implementer: this way, the actual value can be kept in the same Tkinter IntVar control variable that is linked to the two radiobuttons that control that switch.

Here is the layout of the widgets inside.

```
Contained widgets:
    .__colorReadout:
        [ a ColorReadout widget that displays self's text and
          background colors, and a pair of radiobuttons to
          switch between them ]
    .__modelSelector:
        [ a ModelSelector widget that lets the user select
          which color model to display ]
    .__colorSliders:
        [ a ColorSliders widget that lets the user adjust
          the color model's parameters ]

    Grid plan: Vertically stacked in a single column.
```

### 15.1. Manifest constants

Two class variables hold the initial colors, black text and red background.
15.2. Adjuster.set(): Change the color

This method changes the color to the given value. The actual color is stored inside the ColorReadout widget. See Section 16.2, “ColorReadout.set(): Change the displayed color” (p. 67).

```python
# - - - A d j u s t e r . s e t

def set ( self, color ):
    """Change the currently displayed color.
    """
    #-- 1 --
    # [ self.__colorReadout := self.__colorReadout
    #    displaying color ]
    self.__colorReadout.set ( color )
```

Also change the slider positions to reflect the new color; see Section 18.2, “ColorSliders.setColor(): Change the displayed color” (p. 78).

```python
#-- 2 --
# [ self.__colorSliders := self.__colorSliders
#    displaying color ]
self.__colorSliders.setColor ( color )
```

15.3. Adjuster.textColor(): Return the current text color

This is a pass-through method that extracts the current text color from the internal ColorReadout widget. See Section 16.4, “ColorReadout.textColor(): Return the current text color” (p. 68).

```python
# - - - A d j u s t e r . t e x t C o l o r

def textColor ( self ):
    """Return self's current text color.
    """
    return self.__colorReadout.textColor()
```

15.4. Adjuster.bgColor(): Return the current text color

This is another pass-through method. The reference source for the current background color is in the ColorReadout widget. See Section 16.5, “ColorReadout.bgColor(): Return the current background color” (p. 68).

```python
# - - - A d j u s t e r . b g C o l o r

def bgColor ( self ):
    """Return self's current background color.
    """
```
15.5. Adjuster.isText(): Which color is being displayed?

This method is a predicate that returns True if the text color is currently being displayed in the ColorSliders widget, False if the background color is being displayed.

The actual state lives inside the ColorReadout widget, so this method interrogates that widget for the value. See Section 16.1, “ColorReadout.isText(): Which color are we displaying?” (p. 66).

```python
# -- - - A d j u s t e r . i s T e x t

def isText(self):
    """Is self displaying the text color?
    ""
    return self.__colorReadout.isText()
```

15.6. Adjuster.__init__(): Constructor

The constructor starts out by calling the parent’s constructor to initialize this instance’s frame-nature. Then it stores the name of the callback, and calls Section 15.7, “Adjuster.__createWidgets(): Create internal widgets” (p. 62) to set up the contained widgets.

```python
# -- - - A d j u s t e r . _ _ i n i t _ _

def __init__(self, parent, callback):
    """Constructor for the Adjuster compound widget.
    ""
    #-- 1 --
    # [ parent := parent with self added as a new Frame ]
    Frame.__init__(self, parent)

    To prevent callbacks during initialization, we temporarily set the callback pointer to None.

    #-- 2 --
    self.__callback = None

    Section 15.7, “Adjuster.__createWidgets(): Create internal widgets” (p. 62) takes care not only
    of widget creation, but also sets up the initial colors.

    #-- 3 --
    # [ self := self with all contained widgets created and
    #     gridded ]
    self.__createWidgets()
```

One more initialization remains. The ModelSelector widget, just created, selects one of the models initially. We must now retrieve the current model from that widget and pass it down to the ColorSliders widget so that it can display the correct labels on the sliders. See Section 17.2, “ModelSelector.getModel(): Return the current model” (p. 73) and Section 18.1, “ColorSliders.setModel(): Display a new color model” (p. 77).
Now that everything is initialized, we can arm the callback mechanism.

```python
#-- 5 --
self.__callback = callback
```

### 15.7. `Adjuster.__createWidgets()`: Create internal widgets

There are three contained widgets, stacked vertically in column 0 of the grid.

```python
# -- A d j u s t e r . _ _ c r e a t e W i d g e t s

# def __createWidgets ( self ):
# 
#   """Create and grid all internal widgets
#   """

First is the `ColorReadout` widget that displays the selected colors; see Section 16, “class ColorReadout: Text and background color readouts” (p. 65). The initial colors are defined as manifest class constants; see Section 15.1, “Manifest constants” (p. 59). The `ColorReadout` class expects a callback function; we pass it a reference to Section 15.8, “Adjuster.__readoutHandler(): Change between text and background colors” (p. 63).

```python
#-- 1 --
# [ self := self with a new ColorReadout added and gridded
#    that will call self.__callback() when the user
#    changes the text/background choice
# self.__colorReadout := that new ColorReadout ]
self.__colorReadout = ColorReadout ( self,
    self.DEFAULT_BG_COLOR, self.DEFAULT_TEXT_COLOR,
    self.__readoutHandler )
rowx = 0
self.__colorReadout.grid ( row=rowx, column=0, sticky=E+W )
```

Next is the `ModelSelector` widget. It needs a callback function that will be called when the user changes the displayed color model. See Section 17, “class ModelSelector: Widgets to select a color model” (p. 72).

```python
#-- 2 --
# [ self := self with a new ModelSelector added and gridded
#    that will select the color model displayed in
#    self.__colorSliders, and call self.__modelHandler
#    when the color model is changed ]
self.__modelSelector = ModelSelector ( self,
    self.__modelHandler )
rowx += 1
```
Last is the ColorSliders widget that displays the current color model parameters. This too has a callback function, called when the user changes any of the current color model parameters. See Section 18, “class ColorSliders: Color model parameter sliders” (p. 76).

```
#-- 3 --
# [ self := self with a new ColorSliders widget added
#   that displays the current color model parameters
#   and calls self.__modelHandler whenever the user
#   changes a color model parameter
#   self.__colorSliders := that ColorSliders widget ]
self.__colorSliders = ColorSliders ( self, self.bgColor(),
                                  self.__sliderHandler )
rowx += 1
self.__colorSliders.grid ( row=rowx, column=0, sticky=W )
```

15.8. Adjuster.__readoutHandler(): Change between text and background colors

This method is called by the ColorReadout widget when the user changes between displaying the text color and displaying the background color. It has two functions:

1. Internally, it tells the ColorSliders to display whichever color is now shown in the ColorReadout widget.
2. Externally, it calls the Adjuster instance’s callback function.

```
# --- Adjuster.__readoutHandler ---

def __readoutHandler ( self):

    """Change the internal and external colors.

    [ if self.isText() ->
      self := self displaying the text color
      call self.__callback(True, self.textColor())
    else ->
      self := self displaying the background color
      call self.__callback(False, self.bgColor()) ]
    ""

The first step is to determine which color the sliders display: the text color or the background color. Then order the ColorSliders widget to display that color using the current color model.

```
#-- 1 --
if self.isText():
    sliderColor = self.textColor()
else:
    sliderColor = self.bgColor()

#-- 2 --
```
# [ self.__colorSliders := self.__colorSliders displaying sliderColor ]

We must also notify self's callback.

```python
#-- 3 --
if self.__callback is not None:
    self.__callback ( self.isText(), sliderColor )
```

15.9. Adjuster.__modelHandler(): Change the current color model

This method is called when the user changes the currently selected color model. The internal text and background colors do not change, but the ColorSliders widget must change the labels and values shown on the three sliders for the color model's three parameters.

Python's ability to pass classes as objects really simplifies life here. We define an abstract class in Section 7, “class ColorModel: Base class for color models” (p. 11), with concrete subclasses for each color model. Then, when the user selects a new color model, we can pass the actual Python class to the ColorSliders object, where it is needed to convert between parameter values and colors. See also Section 18.1, “ColorSliders.setModel(): Display a new color model” (p. 77).

```python
#--- Adjuster.__modelHandler

def __modelHandler ( self, model ):
    """Change the color model used in the color sliders.

    [ model is a concrete subclass of ColorModel ->
    self.__colorSliders := self.__colorSliders
    displaying its current color using (model) ]
    """

    self.__colorSliders.setModel ( model )
```

15.10. Adjuster.__sliderHandler(): Handler for slider changes

This method is called when the user changes the position of any of the color sliders in the ColorSliders widget. It updates the ColorReadout to show the new color. Then it calls this class’s callback to inform external observers that the displayed color has changed. See Section 16.3, “ColorReadout.internalSet()” (p. 67).

```python
#--- Adjuster.__sliderHandler

def __sliderHandler ( self, color ):
    """Notify observers of a change in the color sliders.
    """

    #-- 1 --
    # [ self.__colorReadout := self.__colorReadout displaying color
    self.__colorReadout.internalSet ( color )

    #-- 2 --
```
16. class  **ColorReadout**: Text and background color readouts

This class is a compound widget that has two functions:

- It displays the current text and background colors in #RRGGBB form.
- It has radiobuttons that allow the user to select whether the color sliders are displaying the text color or the background color.

Here is the class interface:

```python
# - - - - - c l a s s  C o l o r R e a d o u t
class ColorReadout(Frame):
    """Displays text and background colors, and switches between them.

    Exports:
    ColorReadout ( parent, bg, fg, callback=None ):  # (parent is a Frame) and
    (bg is a background color as a Color) and
    (fg is a foreground color as a Color) and
    (callback is a function or None) ->
    parent := parent with a new ColorReadout widget
    added but not gridded, initially displaying
    the background color, which will call
    callback() whenever the user changes
    between text to bg color

    The next two methods return the current text and background colors, respectively; see Section 16.4, "ColorReadout.textColor()": Return the current text color (p. 68) and Section 16.5, "ColorReadout.bgColor()": Return the current background color (p. 68).
```
To determine whether the user has selected text or background color, call this method; see Section 16.1, “ColorReadout.isText(): Which color are we displaying?” (p. 66).

```python
    .isText:
        [ if self is currently displaying the text color ->
          return True
        else -> return False ]
```

Here is the layout of the internal widgets, and the internal attributes that control them.

```plaintext
    Contained widgets:
        __rgbLabel: [ text label '#RRGGBB' ]
        __textColorName:
            [ an Entry widget displaying self.__textColorVar ]
        __bgColorName:
            [ an Entry widget displaying self.__bgColorVar ]
        __textRadio: [ a Radiobutton selecting text color ]
        __bgRadio: [ a Radiobutton selecting background color ]

    Grid plan:
        0 1
        +--------------+-------------------+
        0 |             | __rgbLabel  |
        +--------------+-------------------+
        1 | __bgRadio  | __bgColorName |
        +--------------+-------------------+
        2 | __textRadio| __textColorName |
        +--------------+-------------------+

    State/Internals:
        __callback: [ as passed to constructor, read-only ]
        __isTextVar:
            [ an IntVar that is 1 if displaying text color,
              0 if displaying background color ]
        __bgColor:
            [ self's current background color as a Color instance ]
        __bgColorVar:
            [ a StringVar displaying self.__bgColor as #RRGGBB ]
        __textColor:
            [ self's current text color as a Color instance ]
        __textColorVar:
            [ a StringVar displaying self.__textColor as #RRGGBB ]
```

16.1. ColorReadout.isText(): Which color are we displaying?

This method returns True if the text color is selected, False otherwise. The actual state item that contains this choice is self.__isTextVar, a Tkinter IntVar control variable that is also linked to the Radiobutton widgets.

```python
    # - - - C o l o r R e a d o u t . i s T e x t
```
def isText ( self ):
    
    return self.__isTextVar.get()

16.2. ColorReadout.set(): Change the displayed color

When the user changes the current color, either by clicking on it in the color name pick list, or by adjusting
one of the color parameter sliders, this method is called to display the new color’s name.

# - - - C o l o r R e a d o u t . s e t

def set ( self, color ):
    
    return self.__colorVar.get()

Translating a color into "#RRGGBB" is done by the Color.__str__() method. The widgets that display
this string are linked to control variables, so a call to the control variable’s .set() method changes the
displayed label. See Section 16.3, “ColorReadout.internalSet()” (p. 67).

#-- 1 --
# [ if self.isText() ->
#    self's text color := color
# else ->
#    self's background color := color ]

self.internalSet ( color )

External observers must also be notified that the current color has changed.

#-- 2 --
if self.__callback is not None:
    self.__callback ( )

16.3. ColorReadout.internalSet()

This method actually displays the new color. It does not call the user's callback; it is used when the
color change happens internal to the Adjuster widget. Originally, a change to the color sliders called
the .set() method, but this lead to an infinite loop: the sliders called ColorReadout.set() which
called its callback which told the sliders about the new color, and around and around we go until we
reach the Python recursion limit.

# - - - C o l o r R e a d o u t . i n t e r n a l S e t

def internalSet ( self, color ):
    
    return self.__colorVar.get()

    [ color is a Color instance ->
    if self.isText() ->
        self's text color := color
    else ->
        self's background color := color ]
16.4. `ColorReadout.textColor()`: Return the current text color

This method returns the current text color as a `Color` instance. The actual value is stored in `self.__textColor`, but this is a private attribute.

```python
# ColorReadout.textColor

def textColor(self):
    """Return self's current text color."
    return self.__textColor
```

16.5. `ColorReadout.bgColor()`: Return the current background color

Another pass-through method that protects the internal background color in `self.__bgColor`.

```python
# ColorReadout.bgColor

def bgColor(self):
    """Return self's current background color."
    return self.__bgColor
```

16.6. `ColorReadout.__init__()`: Constructor

The constructor first calls it parent class constructor to make it into a `Frame`. Then it creates the control variable `__isTextVar`, and set it initially to indicate the background color. Finally it calls Section 16.7, “ColorReadout.__createWidgets()” (p. 69) to create and position its contained widgets.

```python
# ColorReadout.__init__

def __init__(self, parent, bg, fg, callback=None):
    """Constructor for ColorReadout."
    ""
    #-- 1 --
    # [ parent := parent with self added as a new Frame ]
    Frame.__init__(self, parent, relief=SUNKEN, bd=4)
    #-- 2 --
    # [ self.__isTextVar := a new IntVar control variable
```
# initialized to 0
# self.__bgColor := bg
# self.__bgColorVar := a new StringVar control
# variable set to str(bg)
# self.__textColor := fg
# self.__textColorVar := a new StringVar control
# variable set to str(fg)
self.__isTextVar = IntVar()
self.__isTextVar.set(0)
self.__bgColor = bg
self.__bgColorVar = StringVar()
self.__bgColorVar.set(str(bg))
self.__textColor = fg
self.__textColorVar = StringVar()
self.__textColorVar.set(str(fg))

#-- 3 --
# [ self := self with all internal widgets created and
#   gridded ]
self.__createWidgets()

#-- 4 --
self.__callback = callback

16.7. ColorReadout.__createWidgets()

This methods creates and grids all the widgets. For the grid plan, refer to Section 16, “class ColorReadout: Text and background color readouts” (p. 65).

```python
# --- ColorReadout.__createWidgets

def __createWidgets(self):
    """Create and grid all internal widgets."
```

The first row contains only a label "#RRGGBB" that helps the user interpret the background and text colors that appear aligned below it.

```python
#-- 1 --
# [ self.__rgbLabel := a new Label added and gridded ]
self.__rgbLabel = Label(self, font=MONO_FONT,
                         text="#RRGGBB")
rowx = 0
self.__rgbLabel.grid(row=rowx, column=1, sticky=E+W)
```

The second row contains the radiobutton and readout for the background color.

```python
#-- 2 --
# [ self := self with a new Radiobutton widget that sets
#   self.__isTextVar to 0 and calls self.__radioHandler
#   when the radiobutton is changed
#   self.__bgRadio := that Radiobutton widget ]
```
It might seem logical to use a `Label` widget to display color names, but they have one drawback: the user can't use the mouse to cut the color name for pasting elsewhere. Consequently, we use an `Entry` widget. Here are some of the less obvious options, and why we use them:

- **relief=RAISED, bd=4**
  Make the labels look like they are sticking up. Use a four-pixel border to give some space for this 3-d effect.

- **exportselection=1**
  Allow the user to cut text from this widget.

- **takefocus=0**
  We don't want the user to be able to change the text displayed here; this option prevents the `tab` key from moving the input focus through this widget.

The next row is quite similar to the previous row, but it is for the text color.

```python
self.__textRadio = Radiobutton ( self,
  command=self.__radioHandler,
  font=BUTTON_FONT, text="Text color",
  value=1, variable=self.__isTextVar )
rowx += 1
self.__textRadio.grid ( row=rowx, column=0, sticky=W )
```

```python
self.__textColorName = Entry ( self,
  exportselection=1, takefocus=0, width=7,
  relief=RAISED, bd=4, bg="white",
  font=MONO_FONT, textvariable=self.__textColorVar )
self.__textColorName.grid ( row=rowx, column=1, padx=4, sticky=W )
```
There is a drawback to using an Entry widget to display the color values: the user can then edit the displayed values, which would mean they no longer reflect the actual color values. One trick we tried is to set state=DISABLED in both Entry widgets. This does a fine job of preventing the user from changing the value, and in an older Tkinter version, the fix worked. However, in a newer Tkinter install, it is now impossible to use cut-and-paste on a disabled widget.

The solution is to bind all keyboard events to a little handler called Section 16.9, “ColorReadout.__fixNames()::Prevent user modification of the Entry widgets” (p. 71). This event handler rewrites the values of both the names to keep them consistent with the internal colors. Here is the code that sets up that event binding.

```python
#-- 6 --
# [ self.__bgColorName := self.__bgColorName set up so
# that any user keypresses cause self.__bgColorVar
# to be set to str(self.__bgColor)
# self.__textColorName := self.__textColorName set up so
# that any user keypresses cause self.__textColorVar
# to be set to str(self.__textColor) ]
self.__bgColorName.bind ( "<Any-KeyRelease>",
                       self.__fixNames )
self.__textColorName.bind ( "<Any-KeyRelease>",
                           self.__fixNames )
```

16.8. ColorReadout.__radioHandler(): Radiobutton state change handler

When the user changes the state of the radiobuttons, we need to call the external callback function.

```python
# - - - ColorReadout.__radioHandler

def __radioHandler ( self):
    """Handler for a change between text and background color."
    if self.__callback is not None:
        self.__callback ( )
```

16.9. ColorReadout.__fixNames(): Prevent user modification of the Entry widgets

For an explanation of what this handler does and why it does it, see Section 16.7, “ColorReadout.__createWidgets()” (p. 69). It is called with the event handler protocol, so it expects an Event object as its argument.

```python
# - - - ColorReadout.__fixNames

def __fixNames ( self, event):
```
These lines re-assert the two class invariants: that the `self.__bgColorVar` control variable reflects the background color in `self.__bgColor`, and the same invariant for `self.__textColorVar` and `self.__textColor`.

```python
self.__bgColorVar.set(str(self.__bgColor))
self.__textColorVar.set(str(self.__textColor))
```

**17. class ModelSelector: Widgets to select a color model**

This class is a compound widget that allows the user to select among different color models. There are a number of different ways to map color space onto coordinate systems, but three numbers are required in any case.

Although originally designed to support three models (HSV, RGB, and CMY), there is no reason why other models cannot be added later.

```python
# - - - - - c l a s s M o d e l S e l e c t o r

class ModelSelector(Frame):
    """Compound widget for selecting color models.
    Exports:
    ModelSelector (parent, callback=None):
        [ (parent is a Frame) and
        (callback is a function or None) ->
        parent := parent with a new ModelSelector widget
        added but not gridded, that lets the user select
        a color model (initially RGB), and calls
        callback(model) when a selection is made, where
        model is a concrete class that inherits from ColorModel
        return that new ModelSelector widget ]
```

This method returns the currently selected model; see Section 17.2, “ModelSelector.getModel(): Return the current model” (p. 73).

```python
.getModel():
    [ returns self's model as a concrete class of ColorModel ]
```

**Internal widgets:**

- `__topLabel`: [ a Label that labels the radiobuttons ]
- `__radioList`: [ a list of Radiobutton widgets, one per model ]

**Grid plan:**

```
0 1 2...
+-----------------+-----------------+-----+
0 | __topLabel   |
+-----------------+-----------------+-----+
```

**New Mexico Tech Computer Center**
Here are some private attributes of the class. First, self.__callback is the callback function. The control variable that links the radiobuttons together into a group is self.__radioVar.

State/Invariants:

- __callback: [as passed to constructor, read-only]
- __radioVar: [an IntVar control variable for the radiobuttons whose value is the model's index in self.__modelList]

There is also a class variable that enumerates the supported color models, as instances of concrete classes implementing the ColorModel abstract class.

ModelSelector.__modelList: [a list of instances of ColorModel concrete classes, in the same order as self.__radioList]

17.1. ModelSelector: Constants

The set of concrete classes of ColorModel, representing the different color models that the user can select, is a class attribute. See Section 8, “class HSVModel: Hue-saturation-value color model” (p. 13), Section 9, “class RGBModel: Red-green-blue color model” (p. 16), and Section 10, “class CMYModel: Cyan-magenta-yellow color model” (p. 17).

__modelList = [HSVModel(), RGBModel(), CMYModel()]

To add a new model, write its class inheriting from ColorModel, implement its virtual methods, and add a call to its constructor to the line above.

17.2. ModelSelector.getModel(): Return the current model

This method returns the currently selected color model as a concrete class implementing ColorModel. The self.__radioVar control variable contains the index of the currently selected model.

```python
# -- ModelSelector.getModel --
def getModel(self):
    """Returns self's current color model."
    """

    #-- 1 --
    # [ modelx := index in self.__radioList of the currently selected color model ]
    modelx = self.__radioVar.get()

    #-- 2 --
    return self.__modelList[modelx]
```
17.3. ModelSelector.__init__(): Constructor

The constructor first calls the parent class constructor, which makes self a Frame.

```python
# - - - M o d e l S e l e c t o r . _ _ i n i t _ _

def __init__ ( self, parent, callback ):
    """Constructor for the ModelSelector compound widget."
    
    #-- 1 --
    # [ parent := parent with a new Frame widget added but
    #     not gridded
    #     self := that new Frame widget ]
    Frame.__init__ ( self, parent, relief=SUNKEN, bd=4 )

Next we store a reference to the callback function, and create the list that will contain the radiobuttons,
and then create and initialize their associated control variable. Finally, we create all the widgets—see
Section 17.4, “ModelSelector.__createWidgets()” (p. 74).

```
The radiobuttons can be arranged either horizontally or vertically. We chose horizontal under the assumption that space may be at a premium, and also because the labels for the color models are short (all are three letters). First we create the associated control variable, then we populate `self.__radioList` with Radiobutton widgets, each displaying the name from the corresponding model as its label.

17.5. `ModelSelector.__radioHandler()`: Handler for radiobutton action

This method is called when the user selects one of the color model radiobuttons.

When the model is changed, we must call our callback function to notify outside observers of the new model. The argument to the callback is a `ColorModel` instance.
modelx = self.__radioVar.get()
self.__callback ( self.__modelList[modelx] )

18. class ColorSliders: Color model parameter sliders

This class is a compound widget containing three sliders (Tkinter Scale widgets) that the user can drag to change the value of one of the color model's parameters.

```python
# - - - - - c l a s s C o l o r S l i d e r s

class ColorSliders(Frame):
    """Compound widget for displaying and adjusting color parameters

Exports:
    ColorSliders ( parent, color, callback=None ): [ (parent is a Frame) and
    (color is the initial color as a Color instance) and
    (callback is a function or None) ->
    parent := parent with a new ColorSliders widget
    added but not gridded, using the HSV model,
    initially displaying white, that will call
    callback(color) whenever the user changes a
    color parameter
    return that new ColorSliders widget ]

The .setModel() changes the model used to compute the slider positions; see Section 18.1, “ColorSliders.setModel()” (p. 77).

    .setModel ( model ):
    [ model is a concrete subclass of ColorModel ->
    self := self with its current color displayed
    using model ]

The .setColor() method changes the currently displayed color; see Section 18.2, “ColorSliders.setColor()” (p. 78).

    .setColor ( color ):
    [ color is a Color instance ->
    self := self displaying color using its current
    model ]

The label for this widget spans all the columns in row 0. The three ParamSlider widgets are gridded horizontally across row 1.

Internal widgets:
    .__topLabel: [ a Label that describes this widget ]
    .__sliderList:
        [ a list containing three ParamSlider widgets
        corresponding to the three parameters of self.__model
        in the same order ]
```
Internal state includes the current color and the current color model.

State/Invariants:

```python
__callback: [as passed to the constructor]
__color: [the currently displayed color as a Color]
__model:
    [the current color model as a concrete subclass of ColorModel]
```

18.1. ColorSliders.setModel(): Display a new color model

This method is called to changed the color model displayed in these sliders.

```python
# -- ColorSliders.setModel

def setModel(self, model):
    """Change the displayed color model."
    
    It is not necessary to display the name of the currently selected color model here. The user can look at the ModelSelector widget to find that out, or they can look at the labels displayed on the parameter sliders.

    The primary work of this method, then, is to tell the ParamSlider widgets to change their labels to those of the parameters of the new models. Once they are relabeled, we call Section 18.2, "ColorSliders.setColor(): Change the displayed color" (p. 78) with the current color, which will tell the parameter sliders to redisplay that color using the newly selected model.

    #-- 1 --
    self.__model = model

    #-- 2 --
    # [ self.__sliderList := self.__sliderList with its
    #     contained widgets relabeled for model (model) ]
    for paramx in range(N_PARAMS):
        self.__sliderList[paramx].setModel(self.__model)

    #-- 3 --
    # [ self := self displaying the parameters self.__color
    #     using model self.__model ]
    self.setColor(self.__color)
```

New Mexico Tech Computer Center
18.2. ColorSliders.setColor(): Change the displayed color

This method is called to change the color currently being displayed in this widget.

```python
#-- 1 --
def setColor( self, color ):
    """Change the displayed color."
    ""

Because each ParamSlider widget knows which parameter of the model is displaying, all we have to do is tell each one the new color; see Section 19.3, “ParamSlider.setColor(): Change the current color” (p. 82).

```python
#-- 1 --
self.__color = color
#-- 2 --
for paramx in range ( N_PARAMS ):
    #-- 1 body --
    # [ self.__sliderList[paramx] := self.__sliderList[paramx]
    # displaying the (paramx)th parameter of (color)
    # using its current color model ]
    self.__sliderList[paramx].setColor( color )
```

18.3. ColorSliders.__init__(): Constructor

The constructor starts by calling its parent class constructor. The calls to .columnconfigure() distribute the width of the included columns equally (weight) and assign a minimum column size (minsize).

```python
#-- 1 --
def __init__( self, parent, color, callback ):
    """Constructor for the ColorSliders widget."
    ""
    #-- 1 --
    # [ parent := parent with a new Frame added but not
    #    gridded
    #    self := that Frame ]
    Frame.__init__( self, parent )
    self.columnconfigure( 0, weight=1, minsize="80" )
    self.columnconfigure( 1, weight=1, minsize="80" )
    self.columnconfigure( 2, weight=1, minsize="80" )
```

Then it creates the initial color (red) and model (HSV), and calls Section 18.4, “ColorSliders.__createWidgets()” (p. 79) to create its internal widgets.

```python
#-- 2 --
# [ self.__color := red
#    self.__model := the HSV color model
#    self.__callback := callback ]
```
18.4. ColorSliders.__createWidgets()

For the widget layout, see Section 18, “class ColorSliders: Color model parameter sliders” (p. 76).

```python
#-- 3 --
# [ self := self with all internal widgets created and
#     gridded ]
self.__createWidgets()

#-- 4 --
self.__callback = callback
```

Next we add a horizontal group of parameter slider widgets: see Section 19, “class ParamSlider: Scale widget for a color model parameter” (p. 80).

```python
#-- 2 --
self.__sliderList = []

# [ self := self with N_PARAMS new ParamSlider widgets
#     added and gridded
# self.__sliderList += those widgets ]
rowx += 1
for paramx in range( N_PARAMS ):
    paramSlider = ParamSlider( self, self.__model,
                                self.__color, paramx, self.__sliderHandler )
    paramSlider.grid( row=rowx, column=paramx, sticky=E+W )
    self.__sliderList.append( paramSlider )
```
18.5. `ColorSliders.__sliderHandler(): Handler for a parameter change`

This method is called when the user changes the setting of any of the `ParamSlider` widgets. This method takes no arguments; when any slider changes, we have to look at all three and convert those settings into the new color.

```python
#--- ColorSliders.__sliderHandler

def __sliderHandler ( self ):
    """Read the ParamSliders and convert them to a new color
    ""

First we read all three sliders and build a list of their values, each value in [0,MAX_PARAM].

```python
#-- 1 --
paramList = []

#-- 2 --
# [ paramList += parameter values of the ParamSlider
#   widgets in self.__sliderList ]
for paramx in range(N_PARAMS):
    paramList.append (self.__sliderList[paramx].get() )

In classes that inherit from `ColorModel`, the `.paramsToColor()` method converts the list of parameters into a `Color` object. See Section 7, “class ColorModel: Base class for color models” (p. 11).

```python
#-- 3 --
# [ paramList is a list of N_PARAMS numbers, each in
#   [0,MAX_PARAM] ->
#   self.__color := paramList converted to a color using
#   the color model in self.__model ]
self.__color = self.__model.paramsToColor ( paramList )

We must call our callback now to notify external observers that the color has changed.

```python
#-- 4 --
if self.__callback is not None:
    self.__callback ( self.__color )
```

19. class `ParamSlider`: Scale widget for a color model parameter

This class is a compound widget containing the controls that let the user adjust one of the parameters of the current color.

Since colors in external form show only 8 bits per pixel, the length of the vertical `Scale` widget for the parameter is exactly 256 pixels, covering the range [0,255]. Because color model parameter values are in the range [0,65535], conversion from the scale value to the parameter value are a simple shift of 8 bits.
# --- --- class ParamSlider

class ParamSlider(Frame):
    """Compound widget with a Scale for adjusting one color parameter."

    Exports:
    ParamSlider ( parent, model, color, paramx, callback=None ):
        [ (parent is a Frame) and
          (model is a ColorModel) and
          (color is a Color) and
          (param is an int in [0,N_PARAMS]) and
          (callback is a function or None) ->
            parent := parent with a new ParamSlider widget added
            but not gridded, that displays the (paramx)th
            parameter of (color) using (model), and calls
            callback() whenever its scale is moved
            return that new ParamSlider widget ]

    The .get() method returns the 16-bit integer value of this parameter; see Section 19.1, "ParamSlider.get(): Retrieve the current parameter value" (p. 82).

    .get(): [ return self's slider value in [0,MAX_PARAM] ]

    The .setModel() and .setColor() methods change the current color model and color, respectively; see Section 19.2, "ParamSlider.setModel(): Change the color model" (p. 82) and Section 19.3, "ParamSlider.setColor(): Change the current color" (p. 82).

    .setModel ( model ):
        [ self := self displaying labels for the (paramx)th
            parameter of model ]
    .setColor ( color ):
        [ self := self displaying the (paramx)th parameter of
            self.model ]

    Each ParamSlider contains four widgets in a vertical row:

    Internal widgets:
    .__topLabel: [ Label showing the parameter's name ]
    .__plusButton: [ Button that increments the parameter ]
    .__scale: [ Scale for adjusting the parameter ]
    .__minusButton: [ Button that decrements the parameter ]

    Grid plan: One vertical column.

    Here are the private attributes.

    State/Invariants:
    .__paramx: [ as passed to the constructor ]
    .__callback: [ as passed to the constructor ]
    .__topLabelVar: [ StringVar for self.__topLabel ]
    .__scaleVar: [ IntVar for self.__scale ]
19.1. `ParamSlider.get()`: Retrieve the current parameter value

This method returns the parameter value in the range \([0, \text{MAX\_PARAM}]\). The actual parameter value lives in the `self.__scaleVar` control variable, but it is an 8-bit value, so we must convert it.

```python
# - - - P a r a m S l i d e r . g e t
def get(self):
    """Return self's parameter value in [0,MAX\_PARAM]."
    return self.__scaleVar.get() << 8
```

19.2. `ParamSlider.setModel()`: Change the color model

This method changes the color model being displayed.

```python
# - - - P a r a m S l i d e r . s e t M o d e l
def setModel(self, model):
    """Change the current color model."
    # - - 1 --
    self.__model = model

First we store the new color model in `self.__model`.

# - - 2 --
self.__topLabelVar = self.__topLabelVar with its
# value set to the (self.__paramx)th parameter name
# of self.__model ]
paramLabel = self.__model.labelList[self.__paramx]
self.__topLabelVar.set(paramLabel)
```

**Note**: Originally, there was logic here to change the displayed color. However, this led to a nasty defect. In the original version, the caller first sets the first slider’s model—but then the callback to change the color would calculate a new color based on the positions of sliders 2 and 3, which still used the old model!

The fix was to put the burden on the caller to redisplay the new color once all the sliders were moved to the new model.

19.3. `ParamSlider.setColor()`: Change the current color

This method changes the color whose parameter is currently displayed.
# --- ParamSlider.setColor

def setColor(self, color):
    """Change the currently displayed color."""
    #-- 1 --
    self.__color = color

We must readjust the position of the Scale widget. First, we translate the new color into its three parameters using the current color model. For this interface, see Section 7, “class ColorModel: Base class for color models” (p. 11).

    #-- 2 --
    # [ paramList := self.__color expressed as a list of
    #   N_PARAMS parameters using color model self.__model ]
    paramList = self.__model.colorToParams(color)

The parameter displayed is an element of paramList, with that element's position specified by self.__paramx. This value must be shifted right by 8 bits to fit into the range [0, MAX_BYTE]. Then the Scale widget is repositioned by calling the .set() method on its control variable.

    #-- 3 --
    scaleValue = paramList[self.__paramx] >> 8

    #-- 4 --
    # [ self.__scale := self.__scale repositioned to value
    #   (scaleValue) ]
    self.__scaleVar.set(scaleValue)

Finally, we must notify the external observers that the color has changed.

    #-- 5 --
    if self.__callback is not None:
        self.__callback()

19.4. ParamSlider.__init__(): Constructor

For internal widgets and their placement, see Section 19, “class ParamSlider: Scale widget for a color model parameter” (p. 80).

# --- ParamSlider.__init__

def __init__(self, parent, model, color, paramx, callback=None):
    """Constructor for ParamSlider."
    ""

First we call the parent constructor and store all the constructor's arguments.

    #-- 1 --
    # [ parent := parent with a new Frame widget added but
    #   not gridded
# self := that Frame widget ]
Frame.__init__ ( self, parent )

#-- 2 --
self.__model = model
self.__color = color
self.__paramx = paramx
self.__callback = None

Create the two control variables. self.__scaleVar is for the Scale widget, so its values vary in the range [0,MAX_BYTE].

The second control variable, self.__topLabelVar, is for the Label widget that displays the parameter name. We can’t create this Label using the normal text= parameter, because it can change any time the model is changed.

For example, if this is the third parameter of the RGB model, this Label would display “B”. The paramx value in that case would be 2 (since indices count from 0).

#-- 3 --
# [ self.__topLabelVar := a new StringVar control variable
#    set to the (paramxth) parameter name of self.__model
# self.__scaleVar := a new IntVar control variable ]
paramLabel = self.__model.labelList[self.__paramx]
self.__topLabelVar = StringVar()
self.__topLabelVar.set ( paramLabel )
self.__scaleVar = IntVar()

Now we’re ready to create the widgets; see Section 19.5, “ParamSlider.__createWidgets()” (p. 84).

#-- 4 --
# [ self := self with all internal widgets created and
#    gridded ]
self.__createWidgets()

#-- 5 --
self.__callback = callback

19.5. ParamSlider.__createWidgets()

def __createWidgets ( self ):
    """Create and grid all widgets."
    ""

The first widget is the Label that displays the parameter name.

#-- 1 --
# [ self := self with a new Label added and gridded,
#    with control variable self.__topLabelVar
#    self.topLabel := that new Label ]
Next is a button with “+” on it that increments the parameter value by 256, which is equivalent to an increment of 1 in the Scale widget’s control variable. For the associated handler, see Section 19.6, “ParamSlider.__plusHandler(): Increment the parameter” (p. 85).

The Scale widget comes next. Its handler, called when the user drags the slider, is Section 19.8, “ParamSlider.__scaleHandler(): Somebody dragged the scale” (p. 86).

The “-” button is pretty similar to the “+” button. For its handler, see Section 19.7, “ParamSlider.__minusHandler(): Decrement the parameter” (p. 86).

19.6. ParamSlider.__plusHandler(): Increment the parameter

This method is called when the user clicks the “+” button to increment the parameter.
19.7. `ParamSlider.__minusHandler()`: Decrement the parameter

This method is called when the user clicks the “-” button to decrement the parameter.

```python
#-- 1 --
def __minusHandler ( self):
    """Increment the parameter."
    """

    oldValue = self.__scaleVar.get()
    if oldValue > 0:
        self.__scaleVar.set ( oldValue - 1 )
        if self.__callback is not None:
            self.__callback()
```

19.8. `ParamSlider.__scaleHandler()`: Somebody dragged the scale

This method is called whenever the slider in the Scale widget gets moved.

```python
#-- 1 --
def __scaleHandler ( self, value):
    """Handler for self.__scale"
    """

    if self.__callback is not None:
        self.__callback()
```
20. class Swatch: Font and color samples

This class is a compound widget containing only two internal widgets.

• On top is a large Text widget that displays some text in the currently selected font using the currently selected foreground color, and its background color is the currently selected background color.

• Below that is a FontSelect widget that allows the user to select different text fonts. For a link to this compound widget’s documentation, see Section 4.1, “Imports” (p. 6).

Here is the class interface.

```python
# - - - - - c l a s s   S w a t c h

class Swatch(Frame):
    """Compound widget for text/background display and font selection.

    Exports:
    Swatch ( parent, bg, fg ):
        [ (parent is a Frame) and
          (bg is the initial background color as a Color instance) and
          (fg is the initial foreground color as a Color instance) ->
          parent := parent with a new Swatch widget added but not
          gridded
          return that new Swatch widget ]

    These methods set either the foreground or background color; see Section 20.2, “Swatch.setTextColor()” (p. 88) and Section 20.3, “Swatch.setBgColor()” (p. 88).

    .setTextColor ( color ):
        [ color is a color as a Color instance ->
          self := self with the text displayed in that color ]
    .setBgColor ( color ):
        [ color is a color as a Color instance ->
          self := self with the background set to color ]

    Internal widgets:
    .__text:      [ a Text widget ]
    .__fontSelect: [ a FontSelect widget ]

    Grid plan: One vertical column.

    The internal state includes the current foreground and background colors, and also the current font.

    State/Invariants:
    .__textColor: [ current text color as a Color ]
    .__bgColor:   [ current background color as a Color ]
    .__font:      [ current font as a tkFont.Font ]
```

New Mexico Tech Computer Center  huey: Internal maintenance specification  87
20.1. Class constants

These class variables contain constants used to configure the widget.

SWATCH_TEXT is the sample text that appears in the swatch initially.

```
SWATCH_TEXT = (  
    "The quick brown fox jumps over\n"  
    "the lazy dog.\n"  
    "ABCDEFGHIJKLMNOPQRSTUVWXYZ\n"  
    "abcdefghijklmnopqrstuvwxyz\n"  
    "0123456789 !"#$%&'()*+,-./\n"  
    ":;<=>?@[\]^_/{|}~" )
```

These two constants specify the width in characters of the Text widget, and its height in lines.

```
SWATCH_WIDE = 40 # Width of the color swatch in characters
SWATCH_HIGH = 8  # Height of the color swatch in lines
```

The next constant specifies the number of font family names that will be visible at once in the font selector widget.

```
FONT_FAMILIES = 15
```

20.2. Swatch.setTextColor()

This method sets the text (foreground) color in the Text widget. The text color of a Text widget is its "fg" (foreground) attribute. The str() function on a Color instance returns the color's name in the "#RRGGBB" form.

```
# - - - S w a t c h . s e t T e x t C o l o r  

def setTextColor ( self, color ):  
    """Sets the text color of self.__text.\n    """
    self.__text["fg"] = str(color)
```

20.3. Swatch.setBgColor()

Similar to Section 20.2, “Swatch.setTextColor()” (p. 88), but the background color is the "bg" attribute.

```
# - - - S w a t c h . s e t B g C o l o r  

def setBgColor ( self, color ):  
    """Sets the background color of self.__text.\n    """
    self.__text["bg"] = str(color)
```
20.4. Swatch.__init__(): Constructor

Call the parent constructor, then create the widgets; see Section 20.5, “Swatch.__createWidgets()” (p. 89).

```python
# - - - Swatch.__init__

def __init__ ( self, parent, bg, fg ):
    """Constructor for Swatch."""
    #-- 1 --
    # [ parent := parent with a new Frame added
    #   self := that Frame ]
    Frame.__init__ ( self, parent )

    #-- 2 --
    self.__textColor = fg
    self.__bgColor = bg

    #-- 3 --
    # [ self := self with all internal widgets added and gridded ]
    self.__createWidgets()

At this point the FontSelect widget contains an initial font, so we can set up the Text widget to use that font. Then we place some sample text into the Text widget.

```
Next is the FontSelect widget. For the documentation on this widget, see Section 4.1, “Imports” (p. 6). Its callback, Section 20.6, “Swatch.__fontHandler(): Handle a font change” (p. 90), is called whenever the user changes the font in any way.

```
#-- 2 --
# [ self := self with a new FontSelect widget added that
#   calls self.__fontHandler when the font is changed
# self.__fontSelect := that widget ]
self.__fontSelect = FontSelect ( self,
  font=BUTTON_FONT,
  listCount=self.FONT_FAMILIES,
  observer=self.__fontHandler )
rowx += 1
self.__fontSelect.grid ( row=rowx, column=0, sticky=W )
```

## 20.6. Swatch.__fontHandler(): Handle a font change

This method is called when the user changes the font displayed in the FontSelect widget. It changes the Text widget to use that font.

```
# - - - S w a t c h . _ _ f o n t H a n d l e r
def __fontHandler ( self, font ):
  """Handler for font changes.
  ""
  self.__text["font"] = font
```

## 21. Code epilogue

The last lines of the script will actually run the application—provided it has been run as a script, in which case Python will set the built-in __name__ variable to "__main__". If another script wishes to import this script in order to reuse some of its pieces, name will not be set to "__main__" while the script is being read, and in that case main() will not be called.

```
#== Epilogue
#--
# [ if this file is being run as a script ->
#   run an instance of the Application class
# else -> I ]
if __name__ == '__main__':
  main()
```