Windows Eiffel Library Tutorial

Interactive Software Engineering, Inc.
The Windows Eiffel Library (WEL) has been designed to make Windows programming easier, more reliable, more convenient, and more powerful by using Eiffel principles. The most obvious definition of WEL is that it is an encapsulation of Windows primitives, making it possible for users of ISE’s Graphical Eiffel for Windows to have direct access to the Windows graphical API.

This tutorial\(^1\) requires some knowledge of Eiffel 3 (the language), EiffelBase (ISE's basic libraries) and EiffelBench (ISE's Programming Environment). Your knowledge of Windows Software Developers Kit will be helpful but not necessary.

In the upcoming chapters, you will build a graphical, interactive Windows program, complete with menus, file saving and loading, graphic and text drawing. On the way, you will be introduced to the major principles of Windows application design, such as message processing, managing a device context, using dialog boxes, and automatic graphics redrawing.

This walk-through consists of eight steps:

- Step 1: Creating an application
- Step 2: Defining a main window class
- Step 3: Drawing text in a window
- Step 4: Drawing lines in a window
- Step 5: Changing line thickness
- Step 6: Repainting a window
- Step 7: Adding a menu
- Step 8: Storing the drawing in a file

Figure 1 shows the application you will have created at the end of this manual.

**Figure 1**

![My application window](image)

The source code for the application is provided at various stages on the distribution disk. The directories in `$EIFFEL3\EXAMPLES\WEL\TUTORIAL` are named STEP1, STEP2 and so on, corresponding to the steps in the tutorial.

\(^1\) This tutorial is also available on our WEB site at [http://www.eiffel.com/doc/manuals/technology/wel/tutorial/index.html](http://www.eiffel.com/doc/manuals/technology/wel/tutorial/index.html).
Step 1: Creating an application

All Windows programs have a window called main window that appears when the user starts the program. In WEL, this window is owned by the application which is responsible for creating and displaying the main window, processing Windows messages and terminating the application.

Every WEL application must define its own descendant of WEL_APPLICATION in order to define the deferred function main_window as follows:

class APPLICATION

  inherit WEL_APPLICATION

  creation
    make
  
  feature

    main_window: WEL_FRAME_WINDOW is
      -- Create the application's main window.
      once
      !! Result.make_top ("My application")
    
  end -- class APPLICATION

This class is the minimal WEL application, if you run it you will get an empty frame window that can be moved, resized, maximized, minimized and closed. Figure 2 shows the appearance of the application.

Figure 2
Step 2: Defining a main window class

In this step, you will learn how to define your window type for the main window, including application-specific behavior and appearance. You will create a more specialized main window class which is descending from WEL_FRAME_WINDOW.

The quickest way to make a window useful is to explain it how to respond to Windows messages. For example, when the user clicks the left mouse button in the main window of My application, the corresponding window object receives a Wm_lbuttondown message from Windows. This tells the window object that the user clicked the mouse in it. It also passes the coordinates of the point where the user clicked. To intercept and respond to Windows messages, WEL defines several procedures corresponding to the most common Windows messages. For instance, WEL_WINDOW has a procedure named on_left_button_down corresponding to the Wm_lbuttondown message. To respond to the message, you just need to redefine this procedure as follows:

```plaintext
on_left_button_down (keys, x_pos, y_pos: INTEGER) is
  do
    information_message_box ("You have pressed the left mouse button.",
                            "Message received")
  end ;
```

Since you have defined a customized main window class, you have to change the main_window function in APPLICATION in order to return an instance of MAIN_WINDOW. Class APPLICATION must be defined as follows:

```plaintext
class APPLICATION
  inherit WEL_APPLICATION

  creation
    make

  feature
    main_window: MAIN_WINDOW is
      do
        !! Result.make
      end ;

end -- class APPLICATION
```

Figure 3 shows the message box displayed when the user clicks in the main window.
The program created here closes when the user clicks on the system close box. But in serious applications, you may want to change this default behavior, for instance, to ask the user if he wants to save his work. WEL provides an easy way to do that, you just need to redefine the boolean function closeable to perform tests and actions needed. If you return True (which is the default value) the user will be able to close the window, otherwise the user will be unable to close it. A possible implementation could be the following:

\[
\text{closeable} : \text{BOOLEAN is} \\
\text{do} \\
\quad \text{Result} := \text{question_message_box} ("Do you want to quit?", "Quit") \\
\text{end ;}
\]
Here is the full source code of `MAIN_WINDOW`:

```plaintext
class
    MAIN_WINDOW

inherit
    WEL_FRAME_WINDOW
redefine
    on_left_button_down, closeable
end

creation
    make

feature {NONE} -- Initialization
    make is
        -- Make the main window.
        do
            make_top ("My application")
        end;

feature {NONE} -- Implementation
    on_left_button_down (keys, x_pos, y_pos: INTEGER) is
        -- Display a message box when the user presses the
        -- the left mouse button.
        do
            information_message_box ("You have pressed the left mouse button.",
                                    "Message received")
        end;

closeable: BOOLEAN is
    -- Does the user want to quit?
    do
        Result := question_message_box ("Do you want to quit?", "Quit")
    end;
end -- class MAIN_WINDOW
```
**Step 3: Drawing text in a window**

In the next sections, you will learn how to draw lines in the window, change the thickness of the lines, and finally save the contents of the window into a file for reloading later on. But first and to make things simple, you will learn to draw text in a window.

To provide applications with graphic functionality, Windows has a set of functions called the Graphic Device Interface, or GDI. The GDI is the graphic engine that Windows applications use to display and manipulate graphics. To draw text, lines, or figures in a window, you need to use a device context given by the GDI. A device context is a virtual surface with associated attributes, such as a pen, brush, font, background color, text color and current position. When you call GDI functions to draw on a device context, the device driver associated with that device context translates that drawing action into appropriate commands. These commands reproduce the drawing action as accurately as possible on the device context, regardless of the display's capabilities. The display might be a low-resolution monochrome screen, a two-million color screen or a printer. In other words, all devices supported by Windows.

To make things more interesting in your program, instead of bringing up a message box, you will respond by drawing text that shows the coordinates of the point where you clicked on the window. The new version of `on_left_button_down` will be defined as follows:

```plaintext
on_left_button_down (keys, x_pos, y_pos: INTEGER) is
    -- Write x_pos and y_pos when the user presses
    -- the left mouse button.
    local
        position: STRING
    do
        position := "(";
        position.append_integer (x_pos);
        position.append (", ");
        position.append_integer (y_pos);
        position.extend (')');
        dc.get;
        dc.text_out (x_pos, y_pos, position);
        dc.release
    end;
```

You also need to add a new attribute in class `MAIN_WINDOW` which is:

```plaintext
dc: WEL_CLIENT_DC;
    -- Device context associated to the current
    -- client window
```

The `make` routine needs to be modified in order to create `dc` as follows:

```plaintext
make is
    do
        make_top ("My application");
        !! dc.make (Current)
    end;
```

Figure 4 shows the result after several clicks.
One more function you can add to the application is clearing the window. The window will be cleared when a right mouse button is clicked. To implement this, redefine `on_right_button` to call `invalidate` which causes the whole window to be repainted. Since your window does not yet know how to repaint itself, it just clears its client area.

```plaintext
on_right_button_down (keys, x_pos, y_pos: INTEGER) is
    -- Invalidate window.
    do
        invalidate
    end :
```
The full code of MAIN_WINDOW is:

```plaintext
class MAIN_WINDOW

inherit WEL_FRAME_WINDOW
redefine on_left_button_down, on_right_button_down, closeable
end

creation
make

feature {NONE} -- Initialization

make is
  -- Make the main window.
  do
    make_top ("My application");
    dc.make (Current)
  end;

feature -- Access

dc: WEL_CLIENT_DC;
  -- Device context associated to the current
  -- client window

feature {NONE} -- Implementation

on_left_button_down (keys, x_pos, y_pos: INTEGER) is
  -- Write x_pos and y_pos when the user presses
  -- the left mouse button.
  local
    position: STRING
  do
    position := "(";
    position.append_integer (x_pos);
    position.append (", ");
    position.append_integer (y_pos);
    position.extend (')');
    dc.get;
    dc.text_out (x_pos, y_pos, position);
    dc.release
  end;

on_right_button_down (keys, x_pos, y_pos: INTEGER) is
  -- Invalidate window.
  do
    invalidate
  end;

closeable: BOOLEAN is
```

8
-- Does the user want to quit?

    do
        Result := question_message_box ("Do you want to quit?", "Quit")
    end ;

end -- class MAIN_WINDOW
Step 4: Drawing lines in a window

In the next few steps, you will build a simple painting program that lets the user draw on the main window. You will do the following steps:

1. Respond to left button clicks and drag by connecting the dots, resulting drawn lines.
2. Respond to the right click by bringing up an input dialog, allowing the user to change the thickness of the line.
3. Automatically redraw the window's contents by storing the points and redrawing them in response to a paint message.

Typically, a window will receive one left button down message, followed by a series of mouse move messages (one for each point dragged over), then followed by a single left button up message. The drawing model will be implemented as follows:

```vhdl
on_left_button_down (keys, x_pos, y_pos: INTEGER) is
    -- Initiate the drawing process.
    do
        if not button_down then
            button_down := true;
            dc.get;
            dc.move_to (x_pos, y_pos)
        end
    end;

on_mouse_move (keys, x_pos, y_pos: INTEGER) is
    -- Connect the points to make lines.
    do
        if button_down then
            dc.line_to (x_pos, y_pos)
        end
    end;

on_left_button_up (keys, x_pos, y_pos: INTEGER) is
    -- Terminate the drawing process.
    do
        if button_down then
            button_down := false;
            dc.release
        end
    end;
```

Attribute `button_down` has been introduced to draw the lines only when the user moves the mouse while the button is down.

```vhdl
button_down: BOOLEAN;
    -- Is the left mouse button down?
```
Figure 5 shows what the application can do.

**Figure 5**
Here is the full text of MAIN_WINDOW:

class
   MAIN_WINDOW

inherit
   WEL_FRAME_WINDOW

redefine
   on_left_button_down, on_left_button_up,
   on_right_button_down, on_mouse_move,
   closeable

creation
   make

feature {NONE} -- Initialization

   make is
      -- Make the main window.
      do
         make_top ("My application");
         !! dc.make (Current)
      end ;

feature -- Access

   dc: WEL_CLIENT_DC;
      -- Device context associated to the current
      -- client window

   button_down: BOOLEAN;
      -- Is the left mouse button down?

feature {NONE} -- Implementation

   on_left_button_down (keys, x_pos, y_pos: INTEGER) is
      -- Initiate the drawing process.
      do
         if not button_down then
            button_down := true ;
            dc.get;
            dc.move_to (x_pos, y_pos)
         end
      end ;

   on_mouse_move (keys, x_pos, y_pos: INTEGER) is
      -- Connect the points to make lines.
      do
         if button_down then
            dc.line_to (x_pos, y_pos)
         end
      end ;
on_left_button_up (keys, x_pos, y_pos: INTEGER) is
  -- Terminate the drawing process.
  do
    if button_down then
      button_down := false;
      dc.release
  end
end;

on_right_button_down (keys, x_pos, y_pos: INTEGER) is
  -- Invalidate window.
  do
    invalidate
  end;

closeable: BOOLEAN is
  -- Does the user want to quit?
  do
    Result := question_message_box ("Do you want to quit?", "Quit")
  end;
end -- class MAIN_WINDOW
**Step 5: Changing line thickness**

At this point, you can draw only thin lines. In order to change line thickness, you have to change the thickness of the pen the application use to draw lines. In this step, you will learn how to set new tools in a display context and how to create a dialog box.

You will use a dialog box to provide a mechanism for the user to change the line thickness. Figure 6 shows the dialog box.

**Figure 6**

There are 2 methods to include a dialog box in your application:

- Using resources
- Writing Eiffel code

The first method consists of using Windows resources capabilities works only with the Professional version of ISE Eiffel since resources are linked to the program's executable file. This solution needs a linker, a resource compiler and a resource editor which are shipped with your C-compiler. Resources compilation process is completely integrated to Professional ISE Eiffel 3 for Windows and fully transparent for the user. You just need to put a resource file in the project directory, start a Freeze (or Finalize) operation, and EiffelBench does the rest. In a nutshell, the freeze or finalization process will automatically copy the resource file from the project directory into the EIFGEN\[W|F]\_CODE directory and will link it to the executable file. Once the resource file is linked to your executable you can just Melt your project. You will need to Freeze again if you change or add any resources. This method is much easier and quicker than the second because it is usually easier to use a visual tool to design a dialog box or a menu instead of writing Eiffel code. One of the most important advantages of resources is that you can change them without touching Eiffel code. It is clear that this is the preferred method if you are a Professional ISE Eiffel 3 for Windows user. You will see later in details how to create and use a resource file in WEL.

The second method consists of writing an Eiffel class which creates each visual component of the dialog box with the right positions and dimensions. Typically, if you use Personal ISE Eiffel 3 for Windows you will choose this method since the Personal version does not allow to produce a stand-alone executable file.

Both methods will be explain in details in the following sections.

**Method 1: Using resources**

Resources are data stored in a program's executable file, but stored separately from the program's normal data segment. Typically, Windows leaves resources on disk when it loads an application into memory, and loads individual resources as it needs them during execution. You've probably noticed dynamic loading of resources when working with Windows programs. When you invoke a program's dialog box for the first time, Windows usually accesses the disk to copy the dialog box resource from the program's .EXE file into memory. Of course, if you want the resource to be loaded when the program is loaded, or if you don't want Windows to be able to discard the resource from memory, you can change its attributes (for more details, see your resource editor manual). These are the resources you will create and use most often:
You can create resources visually using a resource editor as Borland Resource Workshop, Microsoft Developer Studio, or Watcom resource editor. See your resource editor manual to learn how to create a resource file.

For instance, Figure 7 shows Borland Resource Workshop after the dialog box thickness has been created and Figure 8 shows the same with Microsoft Developer Studio.

Figure 7
No matter what resource editor tool you want to use to design the line thickness dialog box, you should obtain a resource file like the following (app.rc).

```c
#include <windows.h>
#define DLG_LINE_THICKNESS 1
#define IDC_EDIT_WIDTH 101
DLG_LINE_THICKNESS DIALOG 98, 38, 124, 49
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "Line thickness"
FONT 8, "MS Sans Serif"
{
    LTEXT "&Width", -1, 9, 12, 22, 8
    EDITTEXT IDC_EDIT_WIDTH, 9, 22, 30, 12, WS_BORDER | WS_GROUP | WS_TABSTOP
    DEFPUSHBUTTON "OK", IDOK, 65, 8, 50, 14, BS_DEFPUSHBUTTON | WS_GROUP | WS_TABSTOP
    PUSHDOWN "Cancel", IDCANCEL, 65, 26, 50, 14, WS_GROUP | WS_TABSTOP
}
```

The **DIALOG** statement defines a window that an application can use to create dialog boxes. The statement defines the position and dimensions of the dialog box on the screen as well as the dialog box style. **control-statement** defines the controls of the dialog box.
nameID DIALOG [ load-mem] x, y, width, height
[optional-statements]
BEGIN
  control-statement
  . . .
END

Important note: The resource file needs to be present in the project directory and its name must be the same as the system name specified in the Ace file with the .RC extension.

As you see at the beginning of the resource file, two identifiers (DLG_LINE_THICKNESS and IDC_EDIT_WIDTH) are declared. You will use them to identify the dialog box and the edit control since a resource file may contain several dialog boxes and one dialog box may contains several controls. WEL provides a simple tool to extract identifiers from a resource file (.RC) or a header file (.H) to make an Eiffel class which contains a set of constants. If you change or add any identifiers in the resource file, make sure to update your Eiffel class as well. After running this tool named H2E (available in $EIFFEL3\UTIL) with your resource file, you will obtain the following class:

indexing
description: "Generated by h2e from the file C:\Eiffel3\examples\we\tutorial\step5\app.rc."

class
APPLICATION_IDS

feature  -- Access
  Dlg_line_thickness: INTEGER is 1;

  Idc_edit_width: INTEGER is 101;

end  -- class APPLICATION_IDS

Figure 9 shows the values entered in H2E in order to obtain the above file.

Figure 9

Note: If you use Microsoft Developer Studio, identifiers are saved in file `resource.h'. In this case, specify this file and not the resource file.
Now, you will learn how to make the connection between this dialog box specified in the resource file and your application. You will create a new class inherited from the class `WEL_MODAL_DIALOG` to load and use the line thickness dialog box. This class will do the following tasks:

- Load the dialog box from the resource and create an edit control. See *make* routine.
- Set the single line edit control Width with the value previously entered by the user (The first time, this value will be equal to 1). See *setup_dialog* routine.
- If the new width entered by the user is a valid integer, save the value in `pen_width` and terminate the dialog box. See *on_ok* routine.
The full text of this class is: (Note that the class APPLICATION_IDS generated by H2E is added in the inheritance clause.)

class

LINE_THICKNESS_DIALOG

inherit

WEL_MODAL_DIALOG

redefine

on_ok, setup_dialog

end ;

APPLICATION_IDS

export

{NONE} all

end

creation

make

feature {NONE} -- Initialization

make (a_parent: WEL_COMPOSITE_WINDOW) is

-- Make the dialog box and create edit.

do

make_by_id (a_parent, Dlg_line_thickness);

!! edit.make_by_id (Current, idc_edit_width);

pen_width := 1

end ;

feature -- Access

edit: WEL_SINGLE_LINE_EDIT;

-- Edit control to enter pen width

pen_width: INTEGER;

-- Pen width entered

feature {NONE} -- Implementation

setup_dialog is

-- Set the width previously entered.

local

s: STRING

do

!! s.make (0);

s.append_integer (pen_width);

edit.set_text (s)

end ;

on_ok is

-- Ensure edit value is an integer, save it in

-- pen_width and close the dialog box.

do

if edit.text.is_integer then
pen_width := edit.text.to_integer,
terminate (idok)
end

end -- class LINE_THICKNESS_DIALOG
Method 2: Writing Eiffel code

If you use Personal ISE Eiffel, you need to create the following class:

class
  LINE_THICKNESS_WINDOW

inherit
  WEL_FRAME_WINDOW
redefine
  on_control_command
end

creation
  make

feature {NONE} -- Initialization

  make (a_parent: WEL_COMPOSITE_WINDOW) is
    -- Make the line thickness window.
    do
      make_child (a_parent, "Line thickness");
      move_and_resize (20, 20, 210, 104, true);
      !! static.make (Current, "Width", 14, 18, 38, 13, -1);
      !! edit.make (Current, ",", 14, 33, 52, 19, -1);
      !! ok_button.make (Current, "OK", 98, 12, 87, 23, -1);
      !! cancel_button.make (Current, "Cancel", 98, 45, 87, 23, -1);
      pen_width := 1
    end;

feature -- Access

  edit: WEL_SINGLE_LINE_EDIT;
    -- Edit control to enter pen width

  ok_button: WEL_PUSH_BUTTON;
    -- Button to validate the value

  cancel_button: WEL_PUSH_BUTTON;
    -- Button to cancel the value

  static: WEL_STATIC;
    -- "Width" static text

  pen_width: INTEGER;
    -- Pen width entered

feature -- Basic operations

  activate is
    -- Activate the window

  local
    s: STRING
do
  !! s.make (0);
  s.append_integer (pen_width);
  edit.set_text (s);
  show
end :

feature {NONE} -- Implementation

  on_control_command (control: WEL_CONTROL) is
    -- Process ok_button and cancel_button selection.
    local
      p: MAIN_WINDOW
    do
      if control = ok_button then
        if edit.text.is_integer then
          pen_width := edit.text.to_integer;
          p ?= parent;
          if p /= void then
            p.set_pen_width (pen_width)
          end :
          hide
        end
      elseif control = cancel_button then
        hide
      end
    end :
end -- class LINE_THICKNESS_WINDOW

In order to be able to change line thickness in the device context, you need to add the following attribute:

  pen: WEL_PEN;
    -- Pen currently selected in dc

You also have to set the default pen width in the make routine:

  make is
    -- Make the main window.
    do
      make_top ("My application");
      !! dc.make (Current);
      set_pen_width (1)
    end :

The implementation of set_pen_width is defined as follows:

  set_pen_width (new_width: INTEGER) is
    -- Set pen width with new_width.
    do
      !! pen.make_solid (new_width, black)
    end ;
The black color can be retrieved from the class WEL_STANDARD_COLORS which must be added in the inheritance clause.

You have to insert a call to select_pen in the routine on_left_button_down as follows. Selecting a pen for a device context will allow you to use a different pen than the default one.

```eiffel
on_left_button_down (keys, x_pos, y_pos: INTEGER) is
  -- Initiate the drawing process.
  do
    if not button_down then
      button_down := true;
      dc.get;
      dc.move_to (x_pos, y_pos);
      dc.select_pen (pen)
    end
  end;
```

The new version of on_right_button will now bring up the dialog box:

According the version you use, you need to update the MAIN_WINDOW class in order to bring up the line thickness dialog box or window when the user pushes on the right button. If you use Professional ISE Eiffel 3 for Windows, you need to add a new attribute line_thickness_dialog and change the body of on_right_button_down as follows:

```eiffel
line_thickness_dialog: LINE_THICKNESS_DIALOG;
  -- Dialog box to change line thickness

on_right_button_down (keys, x_pos, y_pos: INTEGER) is
  -- Bring up line_thickness_dialog and set the
  -- new pen width.
  do
    if line_thickness_dialog = void then
      !! line_thickness_dialog.make (Current)
    end;
    line_thickness_dialog.activate;
    if line_thickness_dialog.ok_pushed then
      set_pen_width (line_thickness_dialog.pen_width)
    end
  end;
```

On the other hand, if you use Personal ISE Eiffel for Windows, you need to add a new attribute line_thickness_window and change the body of on_right_button_down as follows:

```eiffel
line_thickness_window: LINE_THICKNESS_WINDOW;
  -- Window to change line thickness

on_right_button_down (keys, x_pos, y_pos: INTEGER) is
  -- Bring up line_thickness_window and set the
  -- new pen width.
  do
    if line_thickness_window = void then
      !! line_thickness_window.make (Current)
    end;
    line_thickness_window.activate
  end;
```
Figure 10 shows the new capabilities of your application:

Figure 10
This is the full text of `MAIN_WINDOW` (Professional version):

class
    MAIN_WINDOW

inherit
    WEL_FRAME_WINDOW

redefine
    on_left_button_down, on_left_button_up,
    on_right_button_down, on_mouse_move,
    closeable
end;

WEL_STANDARD_COLORS

export
    {NONE} all
end

creation

make

feature {NONE} -- Initialization

    make is
        -- Make the main window.
        do
            make_top ("My application");
            !! dc.make (Current);
            set_pen_width (1)
        end;

feature -- Access

    dc: WELCLIENT_DC;
        -- Device context associated to the current
        -- client window

    button_down: BOOLEAN;
        -- Is the left mouse button down?

    pen: WEL_PEN;
        -- Pen currently selected in dc

    line_thickness_dialog: LINE_THICKNESS_DIALOG;
        -- Dialog box to change line thickness

feature -- Element change

    set_pen_width (new_width: INTEGER) is
        -- Set pen width with new_width.
        do
            !! pen.make_solid (new_width, black)
        end;

feature {NONE} -- Implementation
on_left_button_down (keys, x_pos, y_pos: INTEGER) is
   -- Initiate the drawing process.
   do
      if not button_down then
         button_down := true ;
         dc.get;
         dc.move_to (x_pos, y_pos);
         dc.select_pen (pen)
      end
   end :

on_mouse_move (keys, x_pos, y_pos: INTEGER) is
   -- Connect the points to make lines.
   do
      if button_down then
         dc.line_to (x_pos, y_pos)
      end
   end :

on_left_button_up (keys, x_pos, y_pos: INTEGER) is
   -- Terminate the drawing process.
   do
      if button_down then
         button_down := false ;
         dc.release
      end
   end :

on_right_button_down (keys, x_pos, y_pos: INTEGER) is
   -- Bring up line_thickness_dialog and set the
   -- new pen width.
   do
      if line_thickness_dialog = void then
         !! line_thickness_dialog.make (Current)
      end :
      line_thickness_dialog.activate;
      if line_thickness_dialog.ok_pushed then
         set_pen_width (line_thickness_dialog.pen_width)
      end
   end :

closeable: BOOLEAN is
   -- Does the user want to quit?
   do
      Result := question_message_box ("Do you want to quit?", "Quit")
   end :

end -- class MAIN_WINDOW
This is the full text of *MAIN_WINDOW* (Personal version):

```c
class
    MAIN_WINDOW

inherit
    WEL_FRAME_WINDOW
redefine
    on_left_button_down, on_left_button_up,
    on_right_button_down, on_mouse_move,
    closeable
end ;
WEL_STANDARD_COLORS
export
    {NONE} all
end

creation
make

feature   {NONE} -- Initialization

    make is
        -- Make the main window.
        do
            make_top ("My application");
            !! dc.make (Current);
            set_pen_width (1)
        end ;

feature   -- Access

    dc: WEL_CLIENT_DC;
        -- Device context associated to the current
        -- client window.

    button_down: BOOLEAN;
        -- Is the left mouse button down?

    pen: WEL_PEN;
        -- Pen currently selected in dc.

    line_thickness_window: LINE_THICKNESS_WINDOW;
        -- Window to change line thickness.

feature   -- Element change

    set_pen_width (new_width: INTEGER) is
        -- Set pen width with new_width.
        do
            !! pen.make_solid (new_width, black)
        end ;

feature   {NONE} -- Implementation
```
on_left_button_down (keys, x_pos, y_pos: INTEGER) is
  -- Initiate the drawing process.
  do
    if not button_down then
      button_down := true;
      dc.get;
      dc.move_to (x_pos, y_pos);
      dc.select_pen (pen)
  end
end ;

on_mouse_move (keys, x_pos, y_pos: INTEGER) is
  -- Connect the points to make lines.
  do
    if button_down then
      dc.line_to (x_pos, y_pos)
  end
end ;

on_left_button_up (keys, x_pos, y_pos: INTEGER) is
  -- Terminate the drawing process.
  do
    if button_down then
      button_down := false;
      dc.release
  end
end ;

on_right_button_down (keys, x_pos, y_pos: INTEGER) is
  -- Bring up line_thickness_window and set the
  -- new pen width.
  do
    if line_thickness_window = void then
      !! line_thickness_window.make (Current)
    end ;
    line_thickness_window.activate
  end ;

closeable: BOOLEAN is
  -- Does the user want to quit?
  do
    Result := question_message_box ("Do you want to quit?", "Quit")
  end ;
end -- class MAIN_WINDOW
Step 6: Repainting a window

As you have probably noticed, the graphics and text you draw in a window using device context functions (like `line_to` or `text_out`) disappear when you resize or uncover the window. Windows does not save the graphics that you draw in the device context, the application is in charge to refresh the window when it is necessary. In this step you will learn how to do that.

When the user of your application resizes or uncovers a window, it requires updating, or painting. WEL automatically calls the `on_paint` procedure (from `WEL_COMPOSITE_WINDOW`) when the window needs to be painted. Procedure `on_paint` is where you write the code to paint the contents of the window. There is one major difference between drawing graphics in the `on_paint` procedure and at other times, such as in response to mouse actions. The device context to be used for painting is passed in the `paint_dc` parameter, so your program does not need to get and release it. You will, however, need to select your drawing tools into the `paint_dc`.

To paint a window's contents, you are going to replay the actions that led to the original drawing on `dc`, but use `paint_dc` instead. But first, you need to store the graphic as objects, so you can paint them in the `on_paint` procedure.

Let's say that the window's contents is a set of lines, and each line is a set of points with a width. Then, you can simply define a line as follows:
class
  LINE

inherit
  LINKED_LIST [POINT]

creation
  make

feature  -- Access
  width: INTEGER;
          -- Width of the line

feature  -- Element change
  set_width (a_width: INTEGER) is
    require
      positive_width: a_width >= 0
    do
      width := a_width
    ensure
      width_set: width = a_width
    end ;

  add (x, y: INTEGER) is
    -- Add a point specified by x and y.
    local
      p: POINT
    do
      !! p.make (x, y);
      extend (p)
    end ;

invariant
  positive_width: width >= 0;

end  -- class LINE
Class `POINT` is simply defined as follows:

```
class POINT

creation
make

feature -- Initialization
make (a_x, a_y: INTEGER) is
  do
    x := a_x;
    y := a_y
  ensure
    x_set: x = a_x;
    y_set: y = a_y
  end;

feature -- Access
x: INTEGER; -- x position
y: INTEGER; -- y position

end -- class POINT
```

Using class `LINE`, the basic idea consists of saving mouse movements while the user draws. Then, you will use these data in the `on_paint` procedure to redraw window's contents. First, you need to add the following attributes in class `MAIN_WINDOW`.

```
lines: LINKED_LIST [LINE];
  -- All lines drawn by the user

current_line: LINE;
  -- Line currently drawn by the user
```

Attribute `lines` needs to be created in the `make` routine as follows:

```
make is
  do
    make_top ("My application");
    !! dc.make (Current);
    set_pen_width (1);
    !! lines.make
  end;
```

And finally, you have to change `on_left_button_down` and `on_mouse_move` to store the points in `lines`. 
on_left_button_down (keys, x_pos, y_pos: INTEGER) is
  -- Initiate the drawing process.
do
  if not button_down then
    button_down := true;
dc.get;
dc.move_to (x_pos, y_pos);
dc.select_pen (pen);
  !! current_line.make;
current_line.set_width (pen.width);
lines.extend (current_line);
current_line.add (x_pos, y_pos)
end
end;

on_mouse_move (keys, x_pos, y_pos: INTEGER) is
  -- Connect the points to make lines.
do
  if button_down then
    dc.line_to (x_pos, y_pos);
current_line.add (x_pos, y_pos)
end
end;
At this point, *lines* has all the information needed to redraw the window's contents. Basically, you just need to redefine *on_paint* and iterate over the list to draw the lines as follows:

```
    on_paint (paint_dc: WEL_PAINT_DC; invalid_rect: WEL_RECT) is
       -- Paint the lines.
       local
          a_line: LINE;
          a_pen: WEL_PEN;
          first_point: BOOLEAN
       do
          from
          lines.start
          until
          lines.off
          loop
             from
               first_point := true;
               a_line := lines.item;
               a_line.start;
               !! a_pen.make_solid (a_line.width, black);
               paint_dc.select_pen (a_pen)
             until
               a_line.off
             loop
               if first_point then
                  first_point := false;
                  paint_dc.move_to (a_line.item.x, a_line.item.y)
               else
                  paint_dc.line_to (a_line.item.x, a_line.item.y)
               end
               a_line.forth
            end
          lines.forth
       end
    end ;
```

Now, if you minimize and restore the window, you will see that window's contents is restored.
Here is the full text of `MAIN_WINDOW` (Professional version):

```plaintext
class
    MAIN_WINDOW

inherit
    WEL_FRAME_WINDOW

redefine
    on_left_button_down, on_left_button_up,
    on_right_button_down, on_mouse_move,
    on_paint, closeable
end;

WEL_STANDARD_COLORS

export
    {NONE} all
end

creation
make

feature  {NONE} -- Initialization
make is
    make_top ("My application");
!! dc.make (Current);
set_pen_width (1);
!! lines.make
end ;

feature  -- Access

dc: WEL_CLIENT_DC;
    -- Device context associated to the current
    -- client window

button_down: BOOLEAN;
    -- Is the left mouse button down?

pen: WEL_PEN;
    -- Pen currently selected in dc

line_thickness_dialog: LINE_THICKNESS_DIALOG;
    -- Dialog box to change line thickness

lines: LINKED_LIST [LINE];
    -- All lines drawn by the user

current_line: LINE;
    -- Line currently drawn by the user

feature  -- Element change
```

34
set_pen_width (new_width: INTEGER) is
   -- Set pen width with new_width.
   do
      !! pen.make_solid (new_width, black)
   end;

feature {NONE} -- Implementation

on_left_button_down (keys, x_pos, y_pos: INTEGER) is
   -- Initiate the drawing process.
   do
      if not button_down then
         button_down := true ;
         dc.get;
         dc.move_to (x_pos, y_pos);
         dc.select_pen (pen);
         !! current_line.make;
         current_line.set_width (pen.width);
         lines.extend (current_line);
         current_line.add (x_pos, y_pos)
      end
   end

on_mouse_move (keys, x_pos, y_pos: INTEGER) is
   -- Connect the points to make lines.
   do
      if button_down then
         dc.line_to (x_pos, y_pos);
         current_line.add (x_pos, y_pos)
      end
   end

on_left_button_up (keys, x_pos, y_pos: INTEGER) is
   -- Terminate the drawing process.
   do
      if button_down then
         button_down := false ;
         dc.release
      end
   end

on_right_button_down (keys, x_pos, y_pos: INTEGER) is
   -- Bring up line_thickness_dialog and set the
   -- new pen width.
   do
      if line_thickness_dialog = void then
         !! line_thickness_dialog.make (Current)
      end:
      line_thickness_dialog.activate;
      if line_thickness_dialog.ok_pushed then
         set_pen_width (line_thickness_dialog.pen_width)
      end
   end

end ;
on_paint (paint_dc: WEL_PAINT_DC; invalid_rect: WEL_RECT) is
-- Paint the lines.
local
  a_line: LINE;
a_pen: WEL_PEN;
first_point: BOOLEAN
do
  from lines.start until lines.off loop
    from first_point := true ;
    a_line := lines.item;
    a_line.start;
    !! a_pen.make_solid (a_line.width, black);
    paint_dc.select_pen (a_pen)
    until a_line.off
  loop
   if first_point then
      first_point := false ;
      paint_dc.move_to (a_line.item.x, a_line.item.y)
    else
      paint_dc.line_to (a_line.item.x, a_line.item.y)
    end ;
  end ;
  a_line.forth
  end ;
  lines.forth
end ;
closeable: BOOLEAN is
  -- Does the user want to quit?
  do
    Result := question_message_box ("Do you want to quit?", "Quit"
  end ;
end -- class MAIN_WINDOW
Here is the full text of `MAIN_WINDOW` (Personal version):

```plaintext
class
  MAIN_WINDOW

inherit
  WEL_FRAME_WINDOW
redefine
    on_left_button_down, on_left_button_up,
    on_right_button_down, on_mouse_move,
    on_paint, closeable
end;

WEL_STANDARD_COLORS
export
  {NONE} all
end

creation
  make

feature
  {NONE} -- Initialization

  make is
    -- Make the main window.
    do
      make_top ("My application");
      !! dc.make (Current);
      set_pen_width (1);
      !! lines.make
    end;

feature -- Access

  dc: WEL_CLIENT_DC;
    -- Device context associated to the current
    -- client window

  button_down: BOOLEAN;
    -- Is the left mouse button down?

  pen: WEL_PEN;
    -- Pen currently selected in `dc`

  line_thickness_window: LINE_THICKNESS_WINDOW;
    -- Window to change line thickness

  lines: LINKED_LIST [LINE];
    -- All lines drawn by the user

  current_line: LINE;
    -- Line currently drawn by the user

feature -- Element change
```

37
set_pen_width (new_width: INTEGER) is
  -- Set pen width with new_width.
  do
    !! pen.make_solid (new_width, black)
  end;

feature {NONE} -- Implementation

on_left_button_down (keys, x_pos, y_pos: INTEGER) is
  -- Initiate the drawing process.
  do
    if not button_down then
      button_down := true;
      dc.get;
      dc.move_to (x_pos, y_pos);
      dc.select_pen (pen);
      !! current_line.make;
      current_line.set_width (pen.width);
      lines.extend (current_line);
      current_line.add (x_pos, y_pos)
    end ;
  end ;

on_mouse_move (keys, x_pos, y_pos: INTEGER) is
  -- Connect the points to make lines.
  do
    if button_down then
      dc.line_to (x_pos, y_pos);
      current_line.add (x_pos, y_pos)
    end ;
  end ;

on_left_button_up (keys, x_pos, y_pos: INTEGER) is
  -- Terminate the drawing process.
  do
    if button_down then
      button_down := false;
      dc.release
    end ;
  end ;

on_right_button_down (keys, x_pos, y_pos: INTEGER) is
  -- Bring up line_thickness_window and set the
  -- new pen width.
  do
    if line_thickness_window = void then
      !! line_thickness_window.make (Current)
      end ;
      line_thickness_window.activate
    end ;

on_paint (paint_dc: WEL_PAINT_DC; invalid_rect: WEL_RECT) is
  -- Paint the lines.
  local
a_line: LINE;
a_pen: WEL_PEN;
first_point: BOOLEAN

do
  from
    lines.start
  until
    lines.off
  loop
    from
      first_point := true;
      a_line := lines.item;
      a_line.start;
      !! a_pen.make_solid (a_line.width, black);
      paint_dc.select_pen (a_pen)
    until
      a_line.off
  loop
    if first_point then
      first_point := false;
      paint_dc.move_to (a_line.item.x, a_line.item.y)
    else
      paint_dc.line_to (a_line.item.x, a_line.item.y)
    end;
    a_line.forth
  end;
lines.forth
end;

closeable: BOOLEAN is
  -- Does the user want to quit?
  do
    Result := question_message_box ("Do you want to quit?", "Quit")
  end;
end -- class MAIN_WINDOW
Step 7: Adding a menu

Most Windows applications have a menu on their main window to provide a variety of selections for the user. In this section, you will add a menu to the program. Figure 11 shows the menus that you will add.

Figure 11

Like dialog boxes, you have two ways to add a menu: using resources or writing Eiffel code. Once again, both solutions will be presented for the Professional and Personal users.

Method 1: Using resources

Resource editors provide a very easy way to design menus visually. For instance, Figure 12 shows Borland Resource Workshop's menu module.
After adding the menus in app.rc, the following text will be added to the file.

```c
#define MAIN_MENU_ID 1
#define CMD_NEW 101
#define CMD_OPEN 102
#define CMD_SAVE 103
#define CMD_EXIT 104
#define CMD_LINE_THICKNESS 105

MAIN_MENU_ID MENU
{  
    POPUP "&File"
    {
        MENUITEM "&New", CMD_NEW
        MENUITEM "&Open...", CMD_OPEN
        MENUITEM "&Save...", CMD_SAVE
        MENUITEM SEPARATOR
        MENUITEM "E&xit", CMD_EXIT
    }

    POPUP "&Line"
    {
        MENUITEM "Line &thickness...", CMD_LINE_THICKNESS
    }
}
```

The value of MAIN_MENU_ID, which identifies the menu, will be used in Eiffel to load the menu. Menu item identifiers (as CMD_NEW, CMD_OPEN) will be used to identify options selected by the user. Don't forget to update class APPLICATION_IDS using H2E since new identifiers have been added in the resource file.

Now, you need to add the following once function to load the menu:

```eiffel
main_menu: WEL_MENU is
    -- Window's menu
    once
    Result.make_by_id (Main_menu_id)
end;
```

Class APPLICATION_IDS needs to be added in the inheritance clause of MAIN_WINDOW to use Main_menu_id.
Method 2: Writing Eiffel code

If you can't use a resource editor, you have to create the menu using WEL_MENU's procedures as follows:

```eiffel
main_menu: WEL_MENU is
    -- Window's menu
    local
        file, line: WEL_MENU
    once
        !! file.make;
        file.append_string ("&New", Cmd_new);
        file.append_string ("&Open...", Cmd_open);
        file.append_string ("&Save...", Cmd_save);
        file.append_separator;
        file.append_string ("E&xit", Cmd_exit);
        !! line.make;
        line.append_string ("Line &thickness...", Cmd_line_thickness);
        !! Result.make;
        Result.append_popup (file, "&File");
        Result.append_popup (line, "&Line")
end;

Cmd_new: INTEGER is 101;
Cmd_open: INTEGER is 102;
Cmd_save: INTEGER is 103;
Cmd_exit: INTEGER is 104;
Cmd_line_thickness: INTEGER is 105;
```

Basically, the function creates two popup menus (File and Line) and add them to the main menu. Procedure `append_string` expects two arguments which are an item name (`STRING`) and an unique identifier (`INTEGER`). Optional character "&" identifies the hot key of the option.

No matter which method you use, now you need to set the menu for the window in the `MAIN_WINDOW`'s make routine as follows:

```eiffel
set_menu (main_menu)
```

Figure 13 shows the menu how it appears in the menu window.
At this point, choosing an option in the menu does not perform any task. In order to process menu commands, you have to redefine on_menu_command from WEL_COMPOSITE_WINDOW. This procedure has an INTEGER argument which identifies the option selected by the user (the same as you have specified in the resources or in main_menu function).

A very basic implementation of on_menu_command could be the following:

```vas
on_menu_command (menu_id: INTEGER) is
    -- menu_id has been selected.
    do
        inspect menu_id
        when Cmd_new then
            warning_message_box ("Feature not implemented.", "New")
        when Cmd_open then
            warning_message_box ("Feature not implemented.", "Open")
        when Cmd_save then
            warning_message_box ("Feature not implemented.", "Save")
        when Cmd_exit then
            warning_message_box ("Feature not implemented.", "Exit")
        when Cmd_line_thickness then
            warning_message_box ("Feature not implemented.", "Line thickness")
    end
end ;
```

For instance, if you choose option New in the File menu, Figure 14 shows what you will get.
To make things more interesting, the application will process New option as it should be: Delete all the lines and forces a repainting of the screen. Since there are no lines to redraw, the screen becomes blank. Replace the call to `warning_message_box` by the following code:

```plaintext
when Cmd_new then
    lines.wipe_out;
    invalidate
```

It is easy to respond to the Exit selection by destroying the main window as follows:

```plaintext
when Cmd_exit then
    if closeable then
        destroy
    end
```

You can also move the code from `on_right_button_down` which brings up line thickness dialog box to `on_menu_command` to respond to the Line thickness selection (identifier `Cmd_line_thickness`).
Step 8: Storing the drawing in a file

Since that you have got a data representation of the drawing, you should be able to transfer that data into a file and read it back. You will also use standard dialog boxes to get file names from the user.

You are going to use class `STORABLE` from EiffelBase to save and read data of the drawing. But first, you need to create the following abstraction in order to be able to save the linked list of points.

```eiffel
class
    LINES
inherit
    LINKED_LIST [LINE];
    STORABLE
creation
    make
end -- class LINES
```

In `MAIN_WINDOW` you need to change the definition of `lines` from:

```eiffel
lines: LINKED_LIST [LINE];
```

into:

```eiffel
lines: LINES;
```

The application will show standard dialog boxes in response to the user's selection of Open and Save to get the file name which will be used for the operation. The following attributes need to be added in `MAIN_WINDOW`:

```eiffel
open_dialog: WEL_OPEN_FILE_DIALOG;
    -- Standard dialog box to open a file.

save_dialog: WEL_SAVE_FILE_DIALOG;
    -- Standard dialog box to save a file.
```
Now, you are ready to implement Open and Save options in on_menu_command. Basically, the file name is retrieved from the standard dialog box and used to save (store_by_name) or read (retrieve_by_name) the data. Extension .DRW is adopted for the files (procedures set_filter and set_default_extension).

\[
\begin{align*}
on_menu_command (\text{menu_id}: \text{INTEGER}) \text{ is} \\
& \quad \text{-- menu_id has been selected.} \\
& \text{do} \\
& \quad \text{inspect} \\
& \quad \quad \text{menu_id} \\
& \quad \ldots \\
& \text{when Cmd_open then} \\
& \quad \text{if open_dialog = void then} \\
& \quad \quad \text{!! open_dialog.make;} \\
& \quad \quad \quad \text{open_dialog.set_filter ("Draw file", "}*.*.drw")}; \\
& \quad \quad \text{open_dialog.set_default_extension ("drw")} \\
& \quad \text{end;} \\
& \quad \text{open_dialog.activate (Current);} \\
& \quad \text{if open_dialog.selected then} \\
& \quad \quad \text{lines ?=} \text{lines.retrieve_by_name (open_dialog.file_name);} \\
& \quad \quad \text{invalidate} \\
& \text{end} \\
& \text{when Cmd_save then} \\
& \quad \text{if save_dialog = void then} \\
& \quad \quad \text{!! save_dialog.make;} \\
& \quad \quad \quad \text{save_dialog.set_filter ("Draw file", "}*.*.drw")}; \\
& \quad \quad \text{save_dialog.set_default_extension ("drw")} \\
& \quad \text{end;} \\
& \quad \text{save_dialog.activate (Current);} \\
& \quad \text{if save_dialog.selected then} \\
& \quad \quad \text{lines.store_by_name (save_dialog.file_name)} \\
& \text{end} \\
& \ldots \\
& \text{end}\; ;
\end{align*}
\]

Figure 15 shows the standard open dialog box as it appears when the user chooses the Open option.
Here is the end of the tutorial, we hope that you have plenty of ideas to improve your application!