Lesson 4

Java GUI

1. Introduction

This chapter gives you a short introduction in Java Foundation Classes (JFC) and Swing.

JFC encompass a group of features for building graphical user interfaces (GUIs) and adding rich graphics functionality and interactivity to Java applications.

The Swing API is only one of five libraries that make up the JFC. The Java Foundation Classes also consist of the Abstract Window Toolkit (AWT), the Accessibility API, the 2D API, and enhanced support for drag-and-drop capabilities.

- AWT - basic GUI Toolkit shipped with all versions of JDK. Swing does not reuse any AWT components but does build off some lightweight component facilities;
- Accessibility - enables assistive technologies, such as screen readers and Braille displays, to get information from the user interface. All Swing Components support accessibility;
- 2D API - Various painting styles, drawing styles, fonts and colours. Not Part of SWING;
- Drag and Drop - Click and hold of GUI objects, moving windows or objects *etc*. Not Part of SWING ;

Swing , which provides enhanced versions of all AWT components and augments them with capabilities like pluggable look and feel for custom interfaces and accessibility support to help users with special needs. In the

early days of Java, Sun introduced AWT as a platform-independent user interface builder that addressed the dilemna of producing code for specific operating environments. The theory suggested that a developer could design complex user interfaces on a Sun Solaris platform and easily move them to Microsoft Windows or the Apple Macintosh without having to recompile source code. In reality, AWT failed to meet this promise because the GUIs of these platforms are so different from one another.

In June 1998, Sun presented a solution to the limitations of AWT, including improved and consistent components as well as a much more solid foundation. This new design, based on the Model-View-Controller (MVC) concept, originated in the Smalltalk world and was later exploited by Erich Gamma et al in their book *Design Patterns* (Addison-Wesley, 1995). This technology, known as Swing, is one part of a much larger effort, known as Java Foundation Classes (JFC), that promises to pave a standard road into the future.

The MVC architecture offers some distinct advantages for AWT, including the capability to replace the data model with one that is custom designed for an apllication, or to redesign the user interface for individual components as required. This second point is especially important since it also accomodates nontraditional user interfaces such as speech or braille output devices for users with special needs. This capability is not possible with AWT components.

MVC breaks GUI components into 3 elements:

- ***Model***

  - *State* data for each component;
  - Different for each component;

E.g. Scrollbar - Max and minimum Values, Current Position, width of *thumb* position relative to values

E.g. Menu - Simple List of items.

  - Model data is *always* independent of visual representation;

- ***View***

  - How component is painted on the screen;
  - Can vary between platforms/look and feel;

- ***Controller***

  - dictates how component interacts with events ;
  - Many forms of events - mouse clicks, keyboard, focus, repaint *etc.*;

The three elements interact as follows (Figure 1):

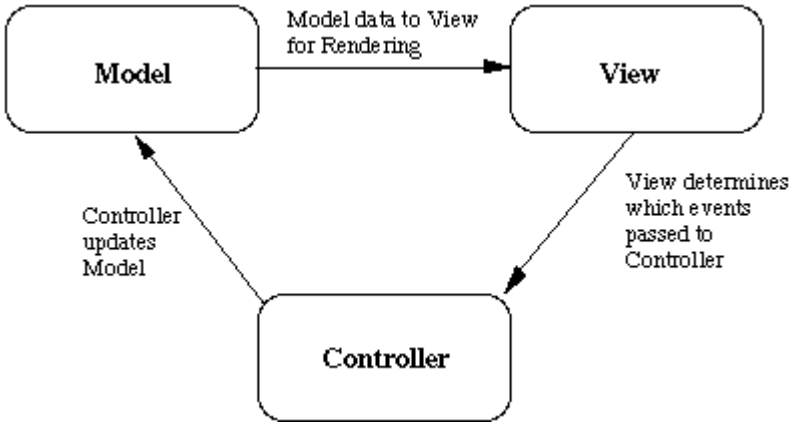

Figure 12: The MVC Communication Model

Swing actually makes use of a simplified variant of the MVC design called the *model-delegate* .This design combines the view and the controller object into a single element that draws the component to the screen and handles GUI events known as the *UI delegate* . Bundling graphics capabilities and event handling is somewhat easy in Java, since much of the event handling is taken care of in AWT. As you might expect, the communication between the model and the UI delegate then becomes a two-way street, as shown in Figure 2.

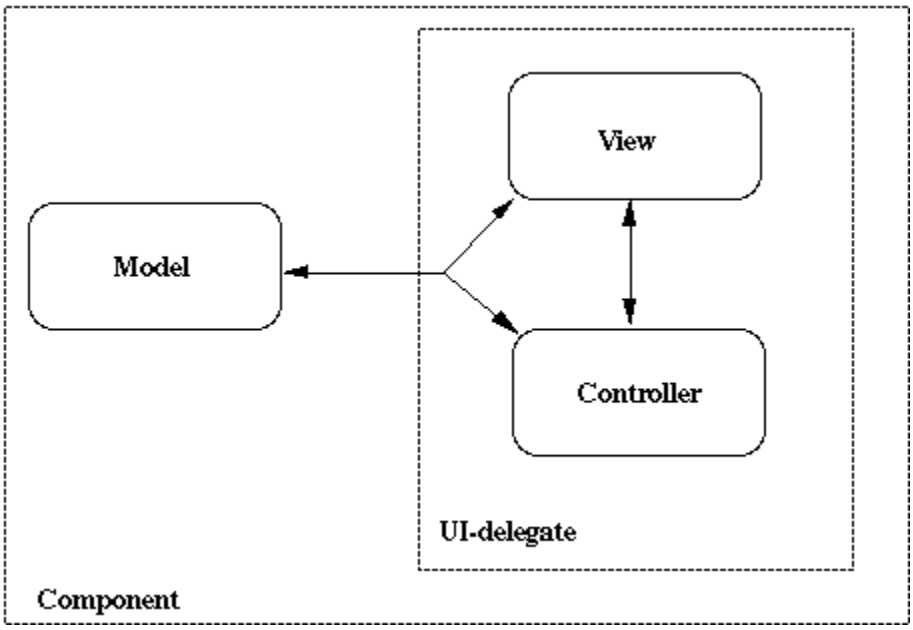

Figure 13: MVC design -> model-delegate

So each Swing component contains a model and a UI delegate. The model is responsible for maintaining information about the component's state. The UI delegate is responsible for maintaining information about how to draw the component on the screen. In addition, the UI delegate (in conjunction with AWT) reacts to various events that propagate through the component.

Architecturally, the Swing class library is similar to AWT. This is a great benefit to those migrating from the older technology to the new. Most Swing component APIs are similar and often the same as their AWT counterparts, so developers can quickly grasp Swing's basics and port older applications with relative ease. Figure 3 shows the user interface class hierarchy supported by Swing. Note that with the addition of a "J" prefix,

the class names closely resemble those currently offered in AWT, though Swing offers user interface classes not found in AWT.

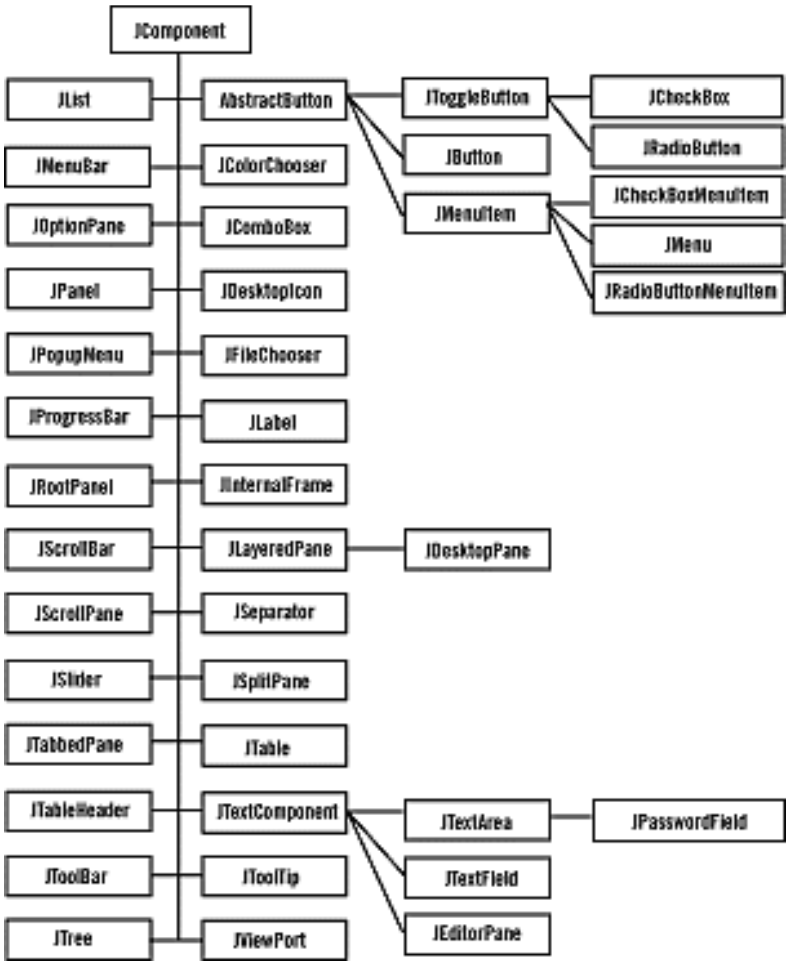

Figure 14: The Swing User Interface Classes

Figure 3 shows that all Swing components extend the JComponent class. Since JComponent extends the AWT container class, this implies that all

Swing user interface components are containers. This offers some avenues for developers to create interesting applications by building up components inside components. For example, inserting a checkbox or graphic into a list element is a relatively simple operation with Swing, but is impossible with AWT. Swing's capabilities generally start where AWT leaves off, letting developers create more advanced applications faster than they could before.

2. Using Swing Components

This chapter gives you the background information you need to use Swing components.

The Swing package defines two types of components:

- top-level containers (JFrame, JApplet, JWindow, JDialog) ;
- lightweight components (*everything-else*, such as JButton, JPanel, and JMenu) ;

- ***JFrame***

A Frame (Figure 4) is a top-level window with a title and a border. Here is a picture of the extremely plain window created by the FrameDemo demonstration application:

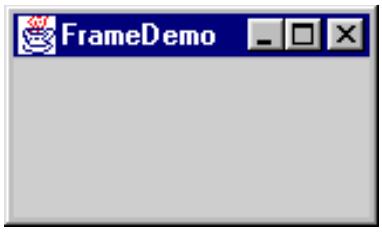

Figure 15: JFrame

The following FrameDemo code shows how to create and set up a frame:

```
//1. Create the frame.
JFrame frame = new JFrame("FrameDemo");

//2. Optional: What happens when the frame closes?
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

//3. Create components and put them in the frame.
//...create emptyLabel...
frame.getContentPane().add(emptyLabel, BorderLayout.CENTER);

//4. Size the frame.
frame.pack();

//5. Show it.
frame.setVisible(true);
```

- ***JDialog***

A Dialog window is an independent subwindow meant to carry temporary notice apart from the main Swing Application Window. Most Dialogs present an error message or warning to a user, but Dialogs can present images, directory trees, or just about anything compatible with the main Swing Application that manages them.

For example, here is an informational dialog (Figure 5):

Figure 16: Jdialog

Swing uses the *JOptionPane* class to provide predefined methods for each type of dialog.

Here is the code that creates and shows it:

```
JOptionPane.showMessageDialog(frame, "Eggs are not supposed to be green.");
```

- ***JPanel***

JPanel is the most commonly used content pane. An instance of the pane is created and then added to a frame. The *add(widgetName)* method allows widgets (ie GUI controls) to be added to the pane. The way they are added is controlled by the current layout manager. JPanel defaults to FlowLayout. All other content panes default to BorderLayout.

The following is a simple template that creates a JFrame container class using inheritance. The created subclass then adds a JPanel. This custom class will form the basis of many of our GUI examples:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
```

```
public class Frame1 extends JFrame
{
        JPanel pane=new JPanel();
         Frame1() // the frame constructor method
        {
                super("My Simple Frame");
                setBounds(100,100,300,100);
                setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                Container con=this.getContentPane(); // inherit main
frame
                con.add(pane); // add the panel to frame
                // customize panel here
                // pane.add(someWidget);
                setVisible(true); // display this frame
        }
         public static void main(String args[]) {
                new Frame1();
        }
}
```

- http://java.sun.com/docs/books/tutorial/uiswing/components/
- http://java.sun.com/j2se/1.5.0/docs/api/

3. Class work

Ex1: Create a window with a title and some text in it.

Package	Class	Method	Description
javax.swing	JFrame	Constructor: JFrame(String title)	Constructs a frame (= a window with a border) with a title.
		getContentPane()	Returns a Component in which content can be placed.
		pack()	Resizes the frame to the preferred size of all its components.
		show()	Shows the frame on the screen and brings it to the front.
	JLabel	Constructor: JLabel(String text)	Constructs a label with initial text in it.

Ex2: Create a window with a button in it, and a counter which counts the number of times the user has clicked the button.

Package	Class	Method	Description

javax.swing	JPanel	setLayout(LayoutManager)	Sets a layout for the component. In this example the borderlayout is used.
		add(Component, int location)	This method can be used if a component has a BorderLayout. the location specifies the location: east, west, north, south or center.
	JButton	Constructor: JButton(String)	Creates a button with initial text in it
		addActionListener(ActionListener)	specifies a class which listens to events generated by clicks on this button
	JLabel	setText(String text)	Changes the text of the label.
	JFrame	setDefaultCloseOperation(int)	Specifies what happens if the close-button of the frame is pressed. Possible close operations are: EXIT_ON_CLOSE;HIDE_ON_CLOSE; DISPOSE_ON_CLOSE
java.awt	Border Layout	Constructor: BorderLayout()	A LayoutManager. This layout manager divides the panel into five subareas: east, west, north, south and center.
java.awt.event	ActionListener	actionPerformed(ActionEvent)	This method is called by the component which generates the events (in this case: clickButton).
	ActionEvent	getSource()	Returns the component which generates the

			events (in this case: clickButton)

Ex3: Create a window with an editable textfield, a button and a label in it. Whenever the button is pressed, the text from the textfield is copied to the label.

Package	**Class**	**Method**	**Description**
javax.swing	JTextField	Constructor: JTextField(int width)	Constructs an editable textfield which can hold a certain amount of characters.
		String getText()	Returns a String containing the text of the textfield.
		setText(String text)	Replaces the text of the textfield by the new String.
	JFrame	setSize(Dimension)	Sets the size of the frame in pixels.
java.awt	Dimension	Constructor: Dimension(int width, int height)	Creates a Dimension object with a certain width and height in pixels.
java.awt.event	ActionListener		
	ActionEvent		

EX4: Create a Frame with a menu in it. In the frame, a panel is placed which has two text fields and a label. The user can put numbers in the textfield. If he presses one of the menu items of the frame, the frame will ask the panel to perform the desired calculation.

Package	**Class**	**Method**	**Description**
javax.swing	JMenuBar	Constructor: JMenuBar()	Creates a menu bar.
		add(Menu)	Adds a menu to the menu bar.
		int getMenuCount()	Returns an integer which is the number of menus of the menu bar.
		Menu getMenu(int n)	Returns the n^{th} menu of the menu bar
	JMenu	Constructor: JMenu(String title)	Creates a menu with a certain title.
		add(MenuItem)	Adds a menu item to the menu.
		addSeparator()	Adds a separator to the menu.
		int getItemCount()	Returns an integer which is the number of menu items of the menu (a separator is a menu item too).
		MenuItem getItem(int n)	Returns the n^{th} menu-item.
	JMenuItem	Constructor: JMenuItem(String title)	Creates a menu item with a certain title.
		addActionListener(ActionListener)	Adds an ActionListener which listens to ActionEvents generated by the menu item.

	JFrame	setJMenuBar(JMenuBar)	Sets the MenuBar for the JFrame
java.lang	System	exit(int status)	Exits the program.
	Integer	static void parseInt(String n)	Tries to parse a string value (n) to an integer value. Throws a NumberFormatException if the String cannot be parsed.
java.awt	GridLayout	Constructor: GridLayout(int rows, int columns)	Creates a GridLayout with a specified number of rows and columns.

4. Home Work

Ex1: Create a simple calculator (**Figure 6**), according to the Model-View-Controller (MVC) pattern. The idea is to separate the application into: **UI** (The **View** and the **Controller**) and **Model.**

- **UI** will be implemented as a subclass of JFrame (View - creates the display, interacting with the Model as necessary and the Controller - responds to user requests, interacting with both the View and Controller as necessary).
- **Model** - where the actual calculations take place.

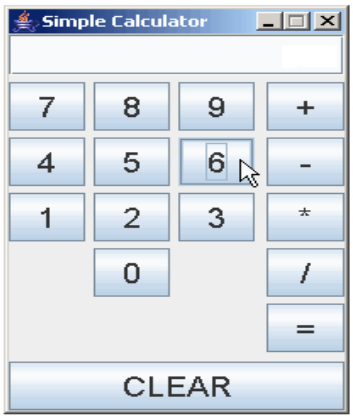

Figure 17: Simple Calculator

Ex2: Create a **X-0** game (**Figure 7**), according to the Model-View-Controller (MVC)pattern.

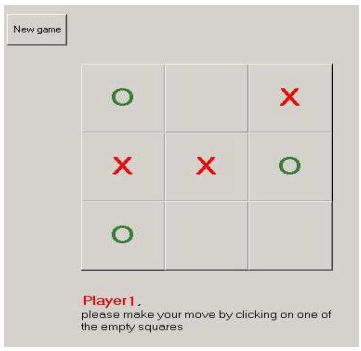

Figure 18: X-0 game

Lesson 5

Windows Forms

1. Introduction

In this paper, we will take a quick look at the following aspects of Windows Forms programming:

- The Form class;
- Controls: how each control is a distinct class, and how controls are added to a form;

The Form - is the most fundamental object used in applications. By itself, a form does nothing. Its main role is to host other objects that the user uses to interact with the computer.

There are various ways you can get a form to your application:

- If you create a Windows Forms Application, it creates a starting form for you;
- After starting an empty project or a Windows Forms Application, you can add a form to it. To do this, on the main menu, you can click Project -> Add New Item... Select Windows Form. Give it a name and click OK;

- http://www.functionx.com/vcsharp/form/Lesson01.htm

2. Windows controls

- ***Button***

A button is a control that establishes a specific state, typically some form of on or off. Buttons are used to perform immediate actions in an interface, define the behavior for a specific feature, or turn a setting on or off. More generally, the various types of buttons are as follows:

- A *push button* - is a button that performs some immediate action, such as displaying or deactivating a dialog, or modifying the values in the window. In Windows Forms, the Button class represents a push button;
- A *check box button* - allows a user to turn a specific option on or off, such as whether a file should be saved as read-only or not. In .NET, the CheckBox class can represent either a check box button or a *toggle button*. A toggle button appears as a normal button, but preserves an up or down state to represent a checked or unchecked mode, respectively;
- A *radio button* - sometimes called an *option button*, is used to select from a set of mutually exclusive options. When one of a group of radio buttons is selected, the other radio buttons in the group are automatically deselected. Radio buttons can be displayed normally or as toggle buttons. Windows Forms provides the RadioButton class for the creation of these objects. All radio buttons in the same container are automatically part of the same group. Use container classes such as GroupBox and Panel to support multiple groups of radio buttons on your forms;

Functions:

- *CheckDlgButton* - Changes the check state of a button control;
- *CheckRadioButton* - Adds a check mark to (checks) a specified radio button in a group and removes a check mark from (clears) all other radio buttons in the group;
- *IsDlgButtonChecked* - Determines whether a button control is checked or whether a three-state button control is checked, unchecked, or indeterminate;

Macros:

- *Button_Enable* - Enables or disables a button;
- *Button_GetCheck* - Gets the check state of a radio button or check box. You can use this macro or send the BM_GETCHECK message explicitly;
- *Button_GetIdealSize* - Gets the size of the button that best fits the text and image, if an image list is present. You can use this macro or send the BCM_GETIDEALSIZE message explicitly;
- *Button_GetImageList* - Gets the BUTTON_IMAGELIST structure that describes the image list that is set for a button control. You can use this macro or send the BCM_GETIMAGELIST message explicitly;
- *Button_GetNote* - Gets the text of the note associated with a command link button. You can use this macro or send the BCM_GETNOTE message explicitly;
- *Button_GetNoteLength* - Gets the length of the note text that may be displayed in the description for a command link. Use this macro or send the BCM_GETNOTELENGTH message explicitly;

- *Button_GetSplitInfo* - Gets information for a specified split button control. Use this macro or send the BCM_GETSPLITINFO message explicitly;
- *Button_GetState* - Gets the check state of a radio button or check box. You can use this macro or send the BM_GETSTATE message explicitly;
- *Button_GetText* - Gets the text of a button;
- *Button_GetTextLength* - Gets the number of characters in the text of a button;
- *Button_GetTextMargin* - Gets the margins used to draw text in a button control. You can use this macro or send the BCM_GETTEXTMARGIN message explicitly;
- *Button_SetCheck* - Sets the check state of a radio button or check box. You can use this macro or send the BM_SETCHECK message explicitly;
- *Button_SetDropDownState* - Sets the drop down state for a specified button with style of BS_SPLITBUTTON. Use this macro or send the BCM_SETDROPDOWNSTATE message explicitly;
- *Button_SetElevationRequiredState* - Sets the elevation required state for a specified button or command link to display an elevated icon. Use this macro or send the BCM_SETSHIELD message explicitly.
- *Button_SetImageList* - Assigns an image list to a button control. You can use this macro or send the BCM_SETIMAGELIST message explicitly.
- *Button_SetNote* - Sets the text of the note associated with a specified command link button. You can use this macro or send the BCM_SETNOTE message explicitly;

- *Button_SetSplitInfo* - Sets information for a specified split button control. Use this macro or send the BCM_SETSPLITINFO message explicitly;
- *Button_SetState* - Sets the highlight state of a button. The highlight state indicates whether the button is highlighted as if the user had pushed it. You can use this macro or send the BM_SETSTATE message explicitly;
- *Button_SetStyle* - Sets the style of a button. You can use this macro or send the BM_SETSTYLE message explicitly;
- *Button_SetText* - Sets the text of a button;
- *Button_SetTextMargin* - Sets the margins for drawing text in a button control. You can use this macro or send the BCM_SETTEXTMARGIN message explicitly;

- ***Combo Box***

A *combo box* is a unique type of control, defined by the COMBOBOX class, that combines much of the functionality of a list box and an edit control.

Functions:

- *DlgDirListComboBox* - Replaces the contents of a combo box with the names of the subdirectories and files in a specified directory. You can filter the list of names by specifying a set of file attributes. The list of names can include mapped drive letters;
- *DlgDirSelectComboBoxEx* - Retrieves the current selection from a combo box filled by using the DlgDirListComboBox function. The selection is interpreted as a drive letter, a file, or a directory name;

- *GetComboBoxInfo* - Retrieves information about the specified combo box;

Macros:

- *ComboBox_AddItemData* - Adds item data to the list in a combo box at the specified location. You can use this macro or send the CB_ADDSTRING message explicitly;
- *ComboBox_AddString* - Adds a string to a list in a combo box. If the combo box does not have the CBS_SORT style, the string is added to the end of the list. Otherwise, the string is inserted into the list and the list is sorted. You can use this macro or send the CB_ADDSTRING message explicitly;
- *ComboBox_DeleteString* - Deletes the item at the specified location in a list in a combo box. You can use this macro or send the CB_DELETESTRING message explicitly;
- *ComboBox_Dir* - Adds names to the list displayed by a combo box. The macro adds the names of directories and files that match a specified string and set of file attributes. It can also add mapped drive letters to the list in a combo box. You can use this macro or send the CB_DIR message explicitly;
- *ComboBox_Enable* - Enables or disables a combo box control;
- *ComboBox_FindItemData* - Finds the first item in a combo box list that has the specified item data. You can use this macro or send the CB_FINDSTRING message explicitly;
- *ComboBox_FindString* - Finds the first string in a combo box list that begins with the specified string. You can use this macro or send the CB_FINDSTRING message explicitly;

- *ComboBox_FindStringExact* - Finds the first string in a combo box list that exactly matches the specified string, except that the search is not case sensitive. You can use this macro or send the CB_FINDSTRINGEXACT message explicitly;
- *ComboBox_GetCount* - Gets the number of items in the list box of a combo box. You can use this macro or send the CB_GETCOUNT message explicitly;
- *ComboBox_GetCueBannerText* - Gets the cue banner text displayed in the edit control of a combo box. Use this macro or send the CB_GETCUEBANNER message explicitly;
- *ComboBox_GetCurSel* - Gets the index of the currently selected item in a combo box. You can use this macro or send the CB_GETCURSEL message explicitly;
- *ComboBox_GetDroppedControlRect* - Retrieves the screen coordinates of a combo box in its dropped-down state. You can use this macro or send the CB_GETDROPPEDCONTROLRECT message explicitly;
- *ComboBox_GetDroppedState* - Ascertains whether the drop list in a combo box control is visible. You can use this macro or send the CB_GETDROPPEDSTATE message explicitly;
- *ComboBox_GetExtendedUI* - Ascertains whether a combo box is using the default user interface (UI) or the extended UI. You can use this macro or send the CB_GETEXTENDEDUI message explicitly;
- *ComboBox_GetItemData* - Gets the application-defined value associated with the specified list item in a combo box. You can use this macro or send the CB_GETITEMDATA message explicitly;

- *ComboBox_GetItemHeight* - Retrieves the height of list items in a combo box. You can use this macro or send the CB_GETITEMHEIGHT message explicitly;
- *ComboBox_GetLBText* - Gets a string from a list in a combo box. You can use this macro or send the CB_GETLBTEXT message explicitly;
- *ComboBox_GetLBTextLen* - Gets the length of a string in the list in a combo box. You can use this macro or send the CB_GETLBTEXTLEN message explicitly;
- *ComboBox_GetMinVisible* - Gets the minimum number of visible items in the drop-down list of a combo box;
- *ComboBox_GetText* - Retrieves the text from a combo box control;
- *ComboBox_GetTextLength* - Gets the number of characters in the text of a combo box;
- *ComboBox_InsertItemData* - Inserts item data in a list in a combo box at the specified location. You can use this macro or send the CB_INSERTSTRING message explicitly;
- *ComboBox_InsertString* - Adds a string to a list in a combo box at the specified location. You can use this macro or send the CB_INSERTSTRING message explicitly;
- *ComboBox_LimitText* - Limits the length of the text the user may type into the edit control of a combo box. You can use this macro or send the CB_LIMITTEXT message explicitly;
- *ComboBox_ResetContent* - Removes all items from the list box and edit control of a combo box. You can use this macro or send the CB_RESETCONTENT message explicitly;

- *ComboBox_SelectItemData* - Searches a list in a combo box for an item that has the specified item data. If a matching item is found, the item is selected. You can use this macro or send the CB_SELECTSTRING message explicitly;
- *ComboBox_SelectString* - Searches a list in a combo box for an item that begins with the characters in a specified string. If a matching item is found, the item is selected. You can use this macro or send the CB_SELECTSTRING message explicitly;
- *ComboBox_SetCueBannerText* - Sets the cue banner text that is displayed for the edit control of a combo box;
- *ComboBox_SetCurSel* - Sets the currently selected item in a combo box. You can use this macro or send the CB_SETCURSEL message explicitly;
- *ComboBox_SetExtendedUI* - Selects either the default user interface (UI) or the extended UI for a combo box that has the CBS_DROPDOWN or CBS_DROPDOWNLIST style. You can use this macro or send the CB_SETEXTENDEDUI message explicitly;
- *ComboBox_SetItemData* - Sets the application-defined value associated with the specified list item in a combo box. You can use this macro or send the CB_SETITEMDATA message explicitly;
- *ComboBox_SetItemHeight* - Sets the height of list items or the selection field in a combo box. You can use this macro or send the CB_SETITEMHEIGHT message explicitly;
- *ComboBox_SetMinVisible* - Sets the minimum number of visible items in the drop-down list of a combo box;
- *SetText* - Sets the text of a combo box;

- *ComboBox_ShowDropdown* - Shows or hides the list in a combo box. You can use this macro or send the CB_RESETCONTENT message explicitly;
- *ListBox_AddItemData* - Adds item data to the list box at the specified location. You can use this macro or send the LB_ADDSTRING message explicitly;

- ***Combo BoxEx***

ComboBoxEx controls are combo box controls that provide native support for item images. To make item images easily accessible, the control provides image list support. By using this control, you can provide the functionality of a combo box without having to manually draw item graphics.

- ***Edit Control***

An *edit control* is a rectangular control window typically used in a dialog box to permit the user to enter and edit text by typing on the keyboard.

Functions:

- *EditWordBreakProc* - An application-defined callback function used with the EM_SETWORDBREAKPROC message. A multiline edit control or a rich edit control calls an EditWordBreakProc function to break a line of text;
- *EditWordBreakPro**c*** - is a placeholder for the application-defined function name;

Macros:

- *Edit_CanUndo* - Determines whether there are any actions in the undo queue of an edit or rich edit control. You can use this macro or send the EM_CANUNDO message explicitly;
- *Edit_EmptyUndoBuffer* - Resets the undo flag of an edit or rich edit control. The undo flag is set whenever an operation within the edit control can be undone. You can use this macro or send the EM_EMPTYUNDOBUFFER message explicitly;
- *Edit_Enable* - Enables or disables an edit control;
- *Edit_FormatLines* - Sets a flag that determines whether text retrieved from a multiline edit control includes soft line-break characters. A soft line break consists of two carriage returns and a line feed and is inserted at the end of a line that is broken because of wordwrapping. You can use this macro or send the EM_FMTLINES message explicitly;
- *Edit_GetCueBannerText* - Gets the text that is displayed as a textual cue, or tip, in an edit control. You can use this macro or send the EM_GETCUEBANNER message explicitly;
- *Edit_GetFirstVisibleLine* - Gets the index of the uppermost visible line in a multiline edit or rich edit control. You can use this macro or send the EM_GETFIRSTVISIBLELINE message explicitly;
- *Edit_GetHandle* - Gets a handle to the memory currently allocated for the text of a multiline edit control. You can use this macro or send the EM_GETHANDLE message explicitly;
- *Edit_GetHilite* - Not implemented;

- *Edit_GetLine* - Retrieves a line of text from an edit or rich edit control. You can use this macro or send the EM_GETLINE message explicitly;
- *Edit_GetLineCount* - Gets the number of lines in the text of an edit control. You can use this macro or send the EM_GETLINECOUNT message explicitly;
- *Edit_GetModify* - Gets the state of an edit or rich edit control's modification flag. The flag indicates whether the contents of the control have been modified. You can use this macro or send the EM_GETMODIFY message explicitly;
- *Edit_GetPasswordChar* - Gets the password character for an edit or rich edit control. You can use this macro or send the EM_GETPASSWORDCHAR message explicitly;
- *Edit_GetRect* - Gets the formatting rectangle of an edit control. You can use this macro or send the EM_GETRECT message explicitly.
- *Edit_GetSel* - Gets the starting and ending character positions of the current selection in an edit or rich edit control. You can use this macro or send the EM_GETSEL message explicitly;
- *Edit_GetText* - Gets the text of an edit control;
- *Edit_GetTextLength* - Gets the number of characters in the text of an edit control;
- *Edit_GetWordBreakProc* - Retrieves the address of an edit or rich edit control's Wordwrap function. You can use this macro or send the EM_GETWORDBREAKPROC message explicitly;
- *Edit_HideBalloonTip* - Hides any balloon tip associated with an edit control. You can use this macro or send the EM_HIDEBALLOONTIP message explicitly;

- *Edit_LimitText* - Limits the length of text that can be entered into an edit control. You can use this macro or send the EM_LIMITTEXT message explicitly;
- *Edit_LineFromChar* - Gets the index of the line that contains the specified character index in a multiline edit or rich edit control. You can use this macro or send the EM_LINEFROMCHAR message explicitly;
- *Edit_LineIndex* - Gets the character index of the first character of a specified line in a multiline edit or rich edit control. You can use this macro or send the EM_LINEINDEX message explicitly;
- *Edit_LineLength* - Retrieves the length, in characters, of a line in an edit or rich edit control. You can use this macro or send the EM_LINELENGTH message explicitly;
- *Edit_ReplaceSel* - Replaces the selected text in an edit control or a rich edit control with the specified text. You can use this macro or send the EM_REPLACESEL message explicitly;
- *Edit_Scroll* - Scrolls the text vertically in a multiline edit or rich edit control. You can use this macro or send the EM_SCROLL message explicitly;
- *Edit_ScrollCaret* - Scrolls the caret into view in an edit or rich edit control. You can use this macro or send the EM_SCROLLCARET message explicitly;
- *Edit_SetCueBannerText* - Sets the text that is displayed as the textual cue, or tip, for an edit control. You can use this macro or send the EM_SETCUEBANNER message explicitly;

- *Edit_SetCueBannerTextFocused* - Sets the text that is displayed as the textual cue, or tip, for an edit control. You can use this macro or send the EM_SETCUEBANNER message explicitly;
- *Edit_SetHandle* - Sets the handle of the memory that will be used by a multiline edit control. You can use this macro or send the EM_SETHANDLE message explicitly;
- *Edit_SetHilite* - Not implemented;
- *Edit_SetModify* - Sets or clears the modification flag for an edit control. The modification flag indicates whether the text within the edit control has been modified. You can use this macro or send the EM_SETMODIFY message explicitly;
- *Edit_SetPasswordChar* - Sets or removes the password character for an edit or rich edit control. When a password character is set, that character is displayed in place of the characters typed by the user. You can use this macro or send the EM_SETPASSWORDCHAR message explicitly;
- *Edit_SetReadOnly* - Sets or removes the read-only style (ES_READONLY) of an edit or rich edit control. You can use this macro or send the EM_SETREADONLY message explicitly;
- *Edit_SetRect* - Sets the formatting rectangle of an edit control. You can use this macro or send the EM_SETRECT message explicitly;
- *Edit_SetRectNoPaint* - Sets the formatting rectangle of a multiline edit control. This macro is equivalent to Edit_SetRect, except that it does not redraw the edit control window. You can use this macro or send the EM_SETRECTNP message explicitly;

- *Edit_SetSel* - Selects a range of characters in an edit or rich edit control. You can use this macro or send the EM_SETSEL message explicitly;
- *Edit_SetTabStops* - Sets the tab stops in a multiline edit or rich edit control. When text is copied to the control, any tab character in the text causes space to be generated up to the next tab stop. You can use this macro or send the EM_SETTABSTOPS message explicitly.
- *Edit_SetText* - Sets the text of an edit control;
- *Edit_SetWordBreakProc* - Replaces an edit control's default Wordwrap function with an application-defined Wordwrap function. You can use this macro or send the EM_SETWORDBREAKPROC message explicitly;
- *Edit_ShowBalloonTip* - Displays a balloon tip associated with an edit control. You can use this macro or send the EM_SHOWBALLOONTIP message explicitly;
- *Edit_Undo* - Undoes the last operation in the undo queue of an edit or rich edit control. You can use this macro or send the EM_UNDO message explicitly;

- ***List Box***

A list box is a control window that contains a simple list of items from which the user can choose.

Functions:

- *DlgDirList* - Replaces the contents of a list box with the names of the subdirectories and files in a specified directory. You can filter

the list of names by specifying a set of file attributes. The list can optionally include mapped drives;

- *DlgDirSelectEx* - Retrieves the current selection from a single-selection list box. It assumes that the list box has been filled by the DlgDirList function and that the selection is a drive letter, filename, or directory name;
- *GetListBoxInfo* - Retrieves information about the specified list box;

Messages:

- *LB_ADDFILE* - Adds the specified filename to a list box that contains a directory listing;
- *LB_ADDSTRING* - Adds a string to a list box. If the list box does not have the LBS_SORT style, the string is added to the end of the list. Otherwise, the string is inserted into the list and the list is sorted;
- *LB_DELETESTRING* - Deletes a string in a list box;
- *LB_DIR* - Adds names to the list displayed by a list box. The message adds the names of directories and files that match a specified string and set of file attributes. LB_DIR can also add mapped drive letters to the list box;
- *LB_FINDSTRING* - Finds the first string in a list box that begins with the specified string;
- *LB_FINDSTRINGEXACT* - Finds the first list box string that exactly matches the specified string, except that the search is not case sensitive;
- *LB_GETANCHORINDEX* - Gets the index of the anchor item
- *LB_GETCARETINDEX* - Retrieves the index of the item that has the focus rectangle in a multiple-selection list box. The item may or may not be selected;

- *LB_GETCOUNT* - Gets the number of items in a list box;
- *LB_GETCURSEL* - Gets the index of the currently selected item, if any, in a single-selection list box;
- *LB_GETHORIZONTALEXTENT* - Gets the width, in pixels, that a list box can be scrolled horizontally (the scrollable width) if the list box has a horizontal scroll bar;
- *LB_GETITEMDATA* - Gets the application-defined value associated with the specified list box item;
- *LB_GETITEMHEIGHT* - Gets the height of items in a list box;
- *LB_GETITEMRECT* - Gets the dimensions of the rectangle that bounds a list box item as it is currently displayed in the list box;
- *LB_GETLISTBOXINFO* - Gets the number of items per column in a specified list box;
- *LB_GETLOCALE* - Gets the current locale of the list box. You can use the locale to determine the correct sorting order of displayed text (for list boxes with the LBS_SORT style) and of text added by the LB_ADDSTRING message;
- *LB_GETSEL* - Gets the selection state of an item;
- *LB_GETSELCOUNT* - Gets the total number of selected items in a multiple-selection list box;
- *LB_GETSELITEMS* - Fills a buffer with an array of integers that specify the item numbers of selected items in a multiple-selection list box;
- *LB_GETTEXT* - Gets a string from a list box;
- *LB_GETTEXTLEN* - Gets the length of a string in a list box;
- *LB_GETTOPINDEX* - Gets the index of the first visible item in a list box. Initially the item with index 0 is at the top of the list box, but if

the list box contents have been scrolled another item may be at the top;

- *LB_INITSTORAGE* - Allocates memory for storing list box items. This message is used before an application adds a large number of items to a list box;
- *LB_INSERTSTRING* - Inserts a string or item data into a list box. Unlike the LB_ADDSTRING message, the LB_INSERTSTRING message does not cause a list with the LBS_SORT style to be sorted;
- *LB_ITEMFROMPOINT* - Gets the zero-based index of the item nearest the specified point in a list box;
- *LB_RESETCONTENT* - Removes all items from a list box;
- *LB_SELECTSTRING* - Searches a list box for an item that begins with the characters in a specified string. If a matching item is found, the item is selected;
- *LB_SELITEMRANGE* - Selects or deselects one or more consecutive items in a multiple-selection list box;
- *LB_SELITEMRANGEEX* - Selects one or more consecutive items in a multiple-selection list box;
- *LB_SETANCHORINDEX* - Sets the anchor item;
- *LB_SETCARETINDEX* - Sets the focus rectangle to the item at the specified index in a multiple-selection list box. If the item is not visible, it is scrolled into view;
- *LB_SETCOLUMNWIDTH* - Sets the width, in pixels, of all columns in a multiple-column list box;
- *LB_SETCOUNT* - Sets the count of items in a list box created with the LBS_NODATA style and not created with the LBS_HASSTRINGS style;

- *LB_SETCURSEL* - Selects a string and scrolls it into view, if necessary. When the new string is selected, the list box removes the highlight from the previously selected string;
- *LB_SETHORIZONTALEXTENT* - Sets the width, in pixels, by which a list box can be scrolled horizontally (the scrollable width). If the width of the list box is smaller than this value, the horizontal scroll bar horizontally scrolls items in the list box. If the width of the list box is equal to or greater than this value, the horizontal scroll bar is hidden;
- *LB_SETITEMDATA* - Sets a value associated with the specified item in a list box;
- *LB_SETITEMHEIGHT* - Sets the height, in pixels, of items in a list box. If the list box has the LBS_OWNERDRAWVARIABLE style, this message sets the height of the item specified by the *wParam* parameter. Otherwise, this message sets the height of all items in the list box;
- *LB_SETLOCALE* - Sets the current locale of the list box. You can use the locale to determine the correct sorting order of displayed text (for list boxes with the LBS_SORT style) and of text added by the LB_ADDSTRING message;
- *LB_SETSEL* - Selects a string in a multiple-selection list box;
- *LB_SETTABSTOPS* - Sets the tab-stop positions in a list box;
- *LB_SETTOPINDEX* - Ensures that the specified item in a list box is visible;

- http://www.functionx.com/vcsharp/index.htm
- http://msdn.microsoft.com/en-us/library/bb773173%28VS.85%29.aspx

3. Class work

Ex1: Create a window with a title and some text in it.

Ex2: Create a window with a button in it, and a counter which counts the number of times the user has clicked the button.

Ex3: Create a simple calculator (Figure 1).

Figure 19: Simple Calculator

Ex4: Create a **X-0** game (Figure 2).

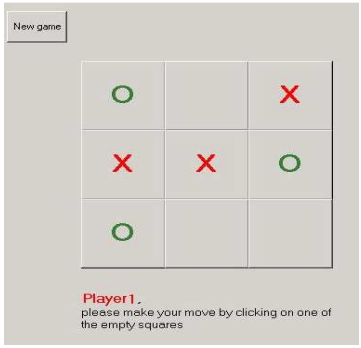

Figure 20: X-0 game

4. Home Work

Ex1: Create a game called Snake through C#. Concept is to create a snake which move around a given screen according to user input. It eats objects randomly emerging on screen and if succesfull in eating them it becomes larger in size and gains score. When the snake dies, the game is over. The snake dies when hits an obstacle or hit its own body.