

Learning to program with F#

Jon Sparring

November 20, 2016

Contents

1	Preface	5
2	Introduction	6
2.1	How to learn to program	6
2.2	How to solve problems	7
2.3	Approaches to programming	7
2.4	Why use F#	8
2.5	How to read this book	9
I	F# basics	10
3	Executing F# code	11
3.1	Source code	11
3.2	Executing programs	11
4	Quick-start guide	14
5	Using F# as a calculator	19
5.1	Literals and basic types	19
5.2	Operators on basic types	24
5.3	Boolean arithmetic	26
5.4	Integer arithmetic	27
5.5	Floating point arithmetic	29
5.6	Char and string arithmetic	31
5.7	Programming intermezzo	32

6	Constants, functions, and variables	34
6.1	Values	37
6.2	Non-recursive functions	42
6.3	User-defined operators	46
6.4	The Printf function	48
6.5	Variables	51
7	In-code documentation	57
8	Controlling program flow	62
8.1	For and while loops	62
8.2	Conditional expressions	66
8.3	Recursive functions	68
8.4	Programming intermezzo	71
9	Ordered series of data	75
9.1	Tuples	76
9.2	Lists	79
9.3	Arrays	84
10	Testing programs	89
10.1	White-box testing	92
10.2	Black-box testing	95
10.3	Debugging by tracing	98
11	Exceptions	105
12	Input and Output	113
12.1	Interacting with the console	114
12.2	Storing and retrieving data from a file	115
12.3	Working with files and directories.	120
12.4	Reading from the internet	120
12.5	Programming intermezzo	121

II	Imperative programming	124
13	Graphical User Interfaces	126
13.1	Drawing primitives in Windows	126
13.2	Programming intermezzo	136
13.3	Events, Controls, and Panels	142
14	Imperative programming	142
14.1	Introduction	142
14.2	Generating random texts	143
14.2.1	0'th order statistics	143
14.2.2	1'th order statistics	143
III	Declarative programming	144
15	Sequences and computation expressions	145
15.1	Sequences	145
16	Patterns	151
16.1	Pattern matching	151
17	Types and measures	154
17.1	Unit of Measure	154
18	Functional programming	158
IV	Structured programming	161
19	Namespaces and Modules	162
20	Object-oriented programming	164
V	Appendix	165
A	Number systems on the computer	166

A.1	Binary numbers	168
A.2	IEEE 754 floating point standard	168
B	Commonly used character sets	169
B.1	ASCII	169
B.2	ISO/IEC 8859	170
B.3	Unicode	170
C	A brief introduction to Extended Backus-Naur Form	174
D	F_b	178
E	Language Details	183
E.1	Arithmetic operators on basic types	183
E.2	Basic arithmetic functions	186
E.3	Precedence and associativity	187
E.4	Lightweight Syntax	189
F	The Some Basic Libraries	190
F.1	<code>System.String</code>	191
F.2	List, arrays, and sequences	191
F.3	Mutable Collections	194
F.3.1	Mutable lists	194
F.3.2	Stacks	194
F.3.3	Queues	194
F.3.4	Sets and dictionaries	194
	Bibliography	195
	Index	196

Chapter 13

Graphical User Interfaces

A *command-line interface (CLI)* is a method for communicating with the user through text. In contrast, a *graphical user interface (GUI)* also includes graphical elements such as windows, icons, and sound, and a typical way to activate these elements are through a pointing device such as the mouse or by touch. Some of these elements may themselves be textual, and thus most operating systems offers access to a command-line interface in a window alongside other interface types.

- command-line interface
- CLI
- graphical user interface
- GUI

Fsharp includes a number of implementations of graphical user interfaces, but at time of writing only *WinForms* is supported on both the Microsoft .Net and the Mono platform, and hence, WinForms will be the subject of the following chapter.

- WinForms

WinForms is designed for *event driven programming*, meaning that at run-time, most time is spend on waiting for the user to perform an action, called and *event*, and each possible event has a predefined response to be performed.

- event driven programming
- event

An example of a graphical user interface is a web-browser, as shown in Figure 13.1. The program present information to the user in terms of text and images and has active areas that may be activated by clicking and which allows the user to go to other web-pages by type URL, to follow hyperlinks, and to generate new pages by entering search queries. Designing easy to use graphical user interfaces is a challenging task. This chapter will focus on examples of basic graphical elements and how to program these in WinForms.

13.1 Drawing primitives in Windows

The main workhorse of WinForms are the functions and classes defined in the namespaces: *System.Windows.Forms* and *System.Drawing*. These give access to the *Windows Graphics Device Interface (GDI+)*, which allows you to create and manipulate graphics objects targeting several platforms such as screens and paper.

- System.Windows.Forms
- System.Drawing
- Windows Graphics Device Interface
- GDI+
- forms

To display a graphical user interface on the screen, the first thing to do is open a window, which acts as a reserved screen-space for our output. In WinForms windows are called *forms*. Code for opening a window is shown in Listing 13.1, and the result is shown in Figure 13.3.

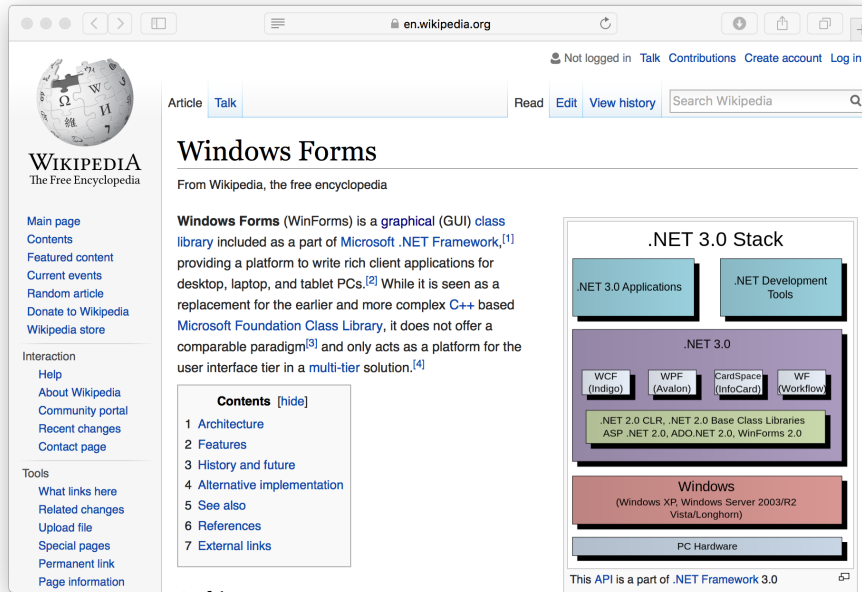


Figure 13.1: A web-browser is a graphical user interface for accessing a web-server and interacting with its services. Here the browser is showing the page https://en.wikipedia.org/wiki/Windows_Forms at time of writing.

Listing 13.1, winforms/openWindow.fsx:
Create the window and turn over control to the operating system.

```
1 // Create a window
2 let win = new System.Windows.Forms.Form ()
3 // Start the event-loop.
4 System.Windows.Forms.Application.Run win
```

The `new System.Windows.Forms.Form ()` creates an object (See Chapter 20), but does not display the window on the screen. When the function `System.Windows.Forms.Application.Run` is applied to the object, then the control is handed over to the WinForms' *event-loop*, which continues until the window is closed by, e.g., pressing the icon designated by the operating system. On the mac OSX that is the red button in the top left corner of the window frame, and on Window it is the cross on the top right corner of the window frame.

The window has a long list of *methods* and *properties*. E.g., the background color may be set by `BackColor`, the title of the window may be set by `Text`, and you may get and set the size of the window with the `Size`. This is demonstrated in Listing 13.2.

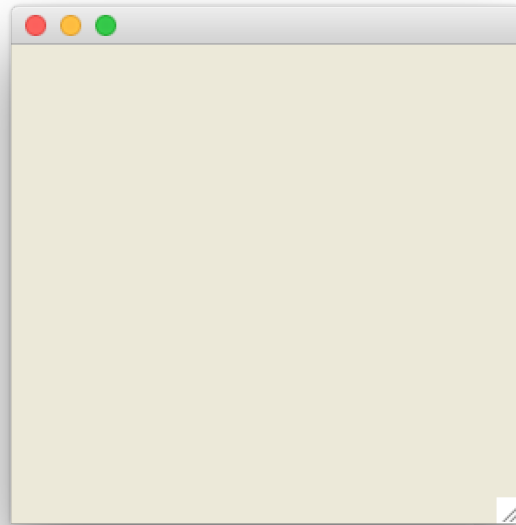


Figure 13.2: A window opened by Listing 13.1.

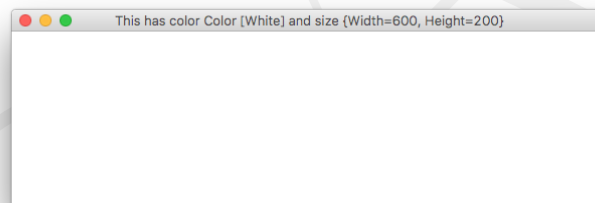


Figure 13.3: A window with user-specified size and background color, see Listing 13.2.

Listing 13.2, winforms/windowAttributes.fsx:
Create the window and changing its properties.

```
1 // Create a window
2 let win = new System.Windows.Forms.Form ()
3 // Set some properties
4 win.BackColor <- System.Drawing.Color.White
5 win.Size <- System.Drawing.Size (600, 200)
6 win.Text <- sprintf "This has color %A and size %A" win.BackColor win.Size
7 // Start the event-loop.
8 System.Windows.Forms.Application.Run win
```

These properties have been programmed as *accessors* implying that they may be used as mutable variables.

· `accessors`

The *System.Drawing.Color* is a general structure for specifying colors as 4 channels: alpha, red, green,

· `System.
Drawing.Color`

Method/Property	Description
A	Get the value of the alpha channel of a color.
B	Get the value of the blue channel of a color.
Black	Get a predefined color with ARGB value of 0 xFF000000.
Blue	Get a predefined color with ARGB value of 0 xFF0000FF.
FromArgb : int -> Color FromArgb : int*int*int*int -> Color	Create a color structure..
G	Get the value of the green channel of a color.
Green	Get a predefined color with ARGB value of 0 xFF00FF00.
R	Get the value of the red channel of a color.
Red	Get a predefined color with ARGB value of 0 xFFFF0000.
ToArgb : Color -> int	Get the 32 bit integer representation of a color.
White	Get a predefined color with ARGB value of 0 xFFFFFFFF.

Table 13.1: Some methods and properties of the `System.Drawing.Color` structure.

blue. Some methods and properties for the `Color` structure is shown in Table 13.1. Each channel is an 8 bit unsigned integer, but often referred as the 32 bit unsigned integer by concatenating the channels. The alpha channel specifies the transparency of a color, where values 0–255 denotes the range of fully transparent to fully opaque, and the remaining channels denote the amount of red, green, and blue, where 0 is none and 255 is full intensity. Any color may be created using the `FromArgb` method, e.g., an opaque red is given by `System.Drawing.Color.FromArgb (255, 255, 0, 0)`. There are also many build-in colors, e.g., the same red color is also a known color and may be obtained as `System.Drawing.Color.Red`. For a given color, then the 4 alpha, red, green, and blue channel's values may be obtained as the A, R, G, B, see Listing 13.3

Constructor	Description
Point(int, int) Point(Size)	An ordered pair of integers specifying x- and y-coordinates in the plane.
Size(int, int) Size(Point)	An ordered pair of integers specifying height and width in the plane.
Rectangle(int, int, int, int) Rectangle(Point, Size)	A structure specifying a rectangular region by its upper left corner and its size.

Table 13.2: Basic geometrical structures in WinForms.

Listing 13.3, drawingColors.fsx:
Defining colors and accessing their values.

```

1 // open namespace for brevity
2 open System.Drawing
3 // Define a color from ARGB
4 let c = Color.FromArgb (0xFF, 0x7F, 0xFF, 0xD4) //Aquamarine
5 printfn "The color %A is (%x, %x, %x, %x)" c c.A c.R c.G c.B
6 // Define a list of named colors
7 let colors =
8     [Color.Red; Color.Green; Color.Blue;
9      Color.Black; Color.Gray; Color.White]
10 for col in colors do
11     printfn "The color %A is (%x, %x, %x, %x)" col col.A col.R col.G col.B

```

```

1 The color Color [A=255, R=127, G=255, B=212] is (ff, 7f, ff, d4)
2 The color Color [Red] is (ff, ff, 0, 0)
3 The color Color [Green] is (ff, 0, 80, 0)
4 The color Color [Blue] is (ff, 0, 0, ff)
5 The color Color [Black] is (ff, 0, 0, 0)
6 The color Color [Gray] is (ff, 80, 80, 80)
7 The color Color [White] is (ff, ff, ff, ff)

```

The `System.Drawing.Size` is a general structure for specifying sizes as height and width pair of integers. WinForms uses a number of types for specifying various objects some of which are shown in Table 13.2.¹²

WinForms supports drawing of geometric primitives such as lines, rectangles, and ellipses, but not points. To draw an element similar to a point, you must draw a small rectangle or ellipse. The location and shape of geometrical primitives are specified in a coordinate system, and WinForms operates with 2 coordinate systems: *screen coordinates* and *client coordinates*. Screen coordinate (x, y) have their origin in the top-left corner of the screen, and x increases to the right, while y increases down. Client coordinates refers to the drawable area of a form or a control, i.e., for a window this will be the area without the window borders, scroll and title bars. A control is a graphical object such as a clickable button, which will be discussed later. Conversion between client and screen coordinates is done with `System.Drawing.PointToClient` and `System.Drawing.PointToScreen`. To draw geometric primitives, we must also specify the pen using for line like primitives and the brush for filled regions.

¹Todo: Note on difference between Size and ClientSize.

²Todo: Do something about the vertical alignment of minpage.

· screen
coordinates
· client
coordinates
· System.
Drawing.
PointToClient
· System.
Drawing.
PointToScreen

Displaying graphics in WinForms is performed as the reaction to an event. E.g., windows are created by the program, moved, minimized, occluded by other windows, resized, etc., by the user or the program, and each action may require that the content of the window is refreshed. Thus, we must create a function that WinForms can call any time. This is known as a *call-back function*, and it is added to an existing form using the `Paint.Add` method. Due to the event-driven nature of WinForms, functions for drawing graphics primitives are only available when responding to an event, e.g., `System.Drawing.Graphics.DrawLine` draws a line in a window, and it is only possible to call this function, as part of an event handling.

· call-back
function
· `Paint.Add`
· `System.`
`Drawing.`
`Graphics.`
`DrawLines`

As an example, consider the problem of drawing a triangle in a window. For this we need to make a function that can draw a triangle not once, but any time. An example of such a program is shown in Listing 13.4.

Listing 13.4, winforms/triangle.fsx:
Adding line graphics to a window.

```
1 // Choose some points and a color
2 let Points =
3     [|System.Drawing.Point (0,0);
4      System.Drawing.Point (10,170);
5      System.Drawing.Point (320,20);
6      System.Drawing.Point (0,0)|]
7 let penColor = System.Drawing.Color.Black
8 // Create window and setup drawing function
9 let pen = new System.Drawing.Pen (penColor)
10 let win = new System.Windows.Forms.Form ()
11 win.Text <- "A triangle"
12 win.Paint.Add (fun e -> e.Graphics.DrawLine (pen, Points))
13 // Start the event-loop.
14 System.Windows.Forms.Application.Run win
```

A walk-through of the code is as follows: First we create an array of points and a pen color, then we create a pen and a window. The method for drawing the triangle is added as an anonymous function using the created window's `Paint.Add` method. This function is to be called as a response to a paint event and takes a `System.Windows.Forms.PaintEventArgs` object, which includes the `System.Drawing.Graphics` object. Since this object will be related to a specific device, when the window's `Paint` method is called, then we may call the `System.Drawing.Graphics.DrawLine` function to sequentially draw lines between our array of points. Finally, we hand the form to the event-loop, which as one of the earliest events will open the window and call the `Paint` function we have associated with the form.

Considering the program in Listing 13.4, we may identify a part that concerns the specification of the triangle, or more generally the graphical model, some parts which handling events, and some which concerns system specific details on initialization of the interface. For future maintenance, it is often a good idea to **separate the model from the implementation**. E.g., it may be that at some point, you decide that you would rather use a different library than WinForms. In this case, the general graphical model will be the same but the specific details on initialization and event handling will be different. While it is not easy to completely separate the general from the specific, it is often a good idea to strive some degree of separation. E.g., in Listing 13.4, the program has been redesigned to make use of an initialization function and a paint function.

Advice

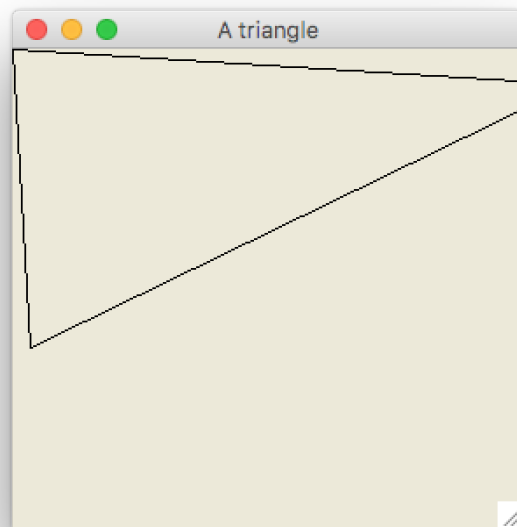


Figure 13.4: Drawing a triangle using Listing 13.4.

Listing 13.5, winforms/triangleOrganized.fsx:

Improved organization of code for drawing a triangle. Compare with Listing 13.4.

```
1 open System.Windows.Forms
2 open System.Drawing
3
4 type coordinates = (float * float) list
5 type pen = Color * float
6
7 /// Create a form and add a paint function
8 let createForm backgroundColor (width, height) title draw =
9     let win = new Form ()
10     win.Text <- title
11     win.BackColor <- backgroundColor
12     win.ClientSize <- Size (width, height)
13     win.Paint.Add draw
14     win
15
16 /// Draw a polygon with a specific color
17 let drawPoints (coords : coordinates) (pen : pen) (e : PaintEventArgs) =
18     let pairToPoint (x : float, y : float) =
19         Point (int (round x), int (round y))
20     let color, width = pen
21     let Pen = new Pen (color, single width)
22     let Points = Array.map pairToPoint (List.toArray coords)
23     e.Graphics.DrawLines (Pen, Points)
24
25 // Setup drawing details
26 let title = "A well organized triangle"
27 let backgroundColor = Color.White
28 let size = (400, 200)
29 let coords = [(0.0, 0.0); (10.0, 170.0); (320.0, 20.0); (0.0, 0.0)]
30 let pen = (Color.Black, 1.0)
31
32 // Create form and start the event-loop.
33 let win = createForm backgroundColor size title (drawPoints coords pen)
34 Application.Run win
```

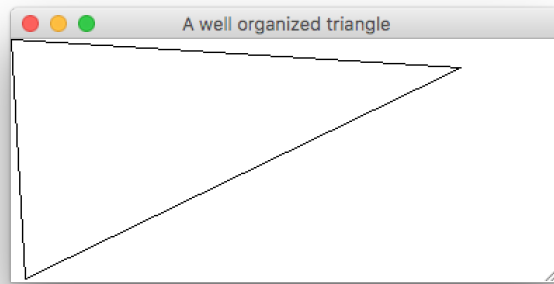


Figure 13.5: Better organization of the code for drawing a triangle, see Listing 13.5.

While this program is longer, to this author there is a much better separation of *what* is to be displayed from the *how* it is to be done, since the *how* is not contained in the functions `createForm` and `drawPoints`. The user-defined types `coordinates` and `pen` further emphasizes the semantic content of the data structures in use and makes use of Fsharp's type checker to reduce run-time errors.³⁴

Organizing code into functions that operate on data structures as Listing 13.5 is the first step in *Structured programming* to be discussed in Part IV. Consider the case, where are to draw two new triangles, that are a translation and a rotations of the original. A simple extension of Listing 13.5 is to make a list of lists of Points and to extend `drawPoints` with a loop for drawing all shapes in the list. This structure should include the ability to draw shapes in different styles, hence we arrive at a structure of type `(coordinates * pen) list`. Furthermore, since the problem is to draw the same shape at different locations and orientations, instead of calculating the new coordinates by hand, it is useful to add functions to translate and rotation a given shape. Thus we arrive at the program shown in Listing 13.7, and which results in the output shown in Figure 13.6.

Structured
programming

³Todo: **requires the introduction of type declarations.**

⁴Todo: **Remember to talk about pen width.**

Listing 13.6, winforms/transformWindows.fsx:
Reusable code for drawing in windows.

```
1 open System.Windows.Forms
2 open System.Drawing
3
4 type coordinates = (float * float) list
5 type pen = Color * float
6 type polygon = coordinates * pen
7
8 /// Create a form and add a paint function
9 let createForm backgroundColor (width, height) title draw =
10     let win = new Form ()
11     win.Text <- title
12     win.BackColor <- backgroundColor
13     win.ClientSize <- Size (width, height)
14     win.Paint.Add draw
15     win
16
17 /// Draw a polygon with a specific color
18 let drawPoints (polygLst : polygon list) (e : PaintEventArgs) =
19     let pairToPoint (x : float, y : float) =
20         Point (int (round x), int (round y))
21
22     for polyg in polygLst do
23         let coords, (color, width) = polyg
24         let pen = new Pen (color, single width)
25         let Points = Array.map pairToPoint (List.toArray coords)
26         e.Graphics.DrawLines (pen, Points)
27
28 /// Translate a point
29 let translatePoint (dx, dy) (x, y) =
30     (x + dx, y + dy)
31
32 /// Translate point array
33 let translatePoints (dx, dy) arr =
34     List.map (translatePoint (dx, dy)) arr
35
36 /// Rotate a point
37 let rotatePoint theta (x, y) =
38     (x * cos theta - y * sin theta, x * sin theta + y * cos theta)
39
40 /// Rotate point array
41 let rotatePoints theta arr =
42     List.map (rotatePoint theta) arr
```

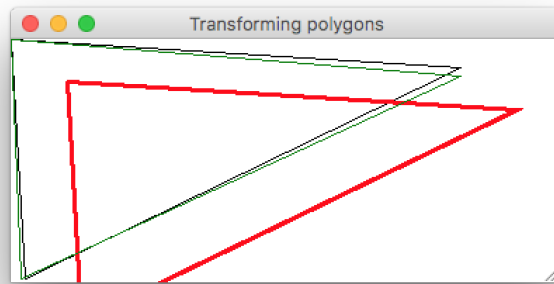


Figure 13.6: Transformed versions of the same triangle resulting from running the code in Listing 13.7.

Listing 13.7, winforms/transformWindows.fsx:

Code for drawing triangles using the reusable part shown in Listing 13.7.

```

44 // Setup drawing details
45 let title = "Transforming polygons"
46 let backgroundColor = Color.White
47 let size = (400, 200)
48 let points = [(0.0, 0.0); (10.0, 170.0); (320.0, 20.0); (0.0, 0.0)]
49 let polygLst =
50     [(points, (Color.Black, 1.0));
51      (translatePoints (40.0, 30.0) points, (Color.Red, 2.0));
52      (rotatePoints (1.0 * System.Math.PI / 180.0) points, (Color.Green, 1.0))
53     ]
54 // Create form and start the event-loop.
55 let win = createForm backgroundColor size title (drawPoints polygLst)
56 Application.Run win

```

We now have a basis for solving the following problem:

Problem 13.1:

Given a triangle produce a Mandela drawing, where n rotated versions of the triangle is drawn around its center of mass.

Reusing the top part of Listing 13.7 and replacing the bottom part with the code shown in Listing 13.8, we arrive at the solution illustrated in Figure 13.7.

Listing 13.8, winforms/rotationalSymmetry.fsx:
Create the window and changing its properties.

```

44 /// Calculate the mass center of a list of points
45 let centerOfPoints (points : (float * float) list) =
46     let addToAccumulator acc elm = (fst acc + fst elm, snd acc + snd elm)
47     let sum = List.fold addToAccumulator (0.0, 0.0) points
48     (fst sum / (float points.Length), snd sum / (float points.Length))
49
50 /// Generate repeated rotated point-color pairs
51 let rec rotatedLst points color width src dest nth n =
52     if n > 0 then
53         let newPoints =
54             points
55             |> translatePoints (- fst src, - snd src)
56             |> rotatePoints ((float n) * nth)
57             |> translatePoints dest
58         (newPoints, (color, width))
59         :: (rotatedLst points color width src dest nth (n - 1))
60     else
61         []
62
63 // Setup drawing details
64 let title = "Rotational Symmetry"
65 let backgroundColor = Color.White
66 let size = (600, 600)
67 let points = [(0.0, 0.0); (10.0, 170.0); (320.0, 20.0); (0.0, 0.0)]
68 let src = centerOfPoints points
69 let dest = ((float (fst size)) / 2.0, (float (snd size)) / 2.0)
70 let n = 36;
71 let nth = (360.0 / (float n)) * (System.Math.PI / 180.0)
72 let orgPoints =
73     points
74     |> translatePoints (fst dest - fst src, snd dest - snd src)
75 let polygLst =
76     rotatedLst points Color.Blue 1.0 src dest nth n
77     @ [(orgPoints, (Color.Red, 3.0))]
78
79 // Create form and start the event-loop.
80 let win = createForm backgroundColor size title (drawPoints polygLst)
81 Application.Run win

```

5

The `System.Drawing.Graphics` class contains many other algorithms for drawing graphics primitives, some of which are listing in Table 13.3 ⁶

13.2 Programming intermezzo

Reusing the top part of Listing 13.7 we are now able to tackle complicated problems such as space-filling curves. A curve in 2 dimensions has a length but no width, and we can only visualize it by

⁵Todo: Remember to add something on piping.

⁶Todo: Give examples of more methods

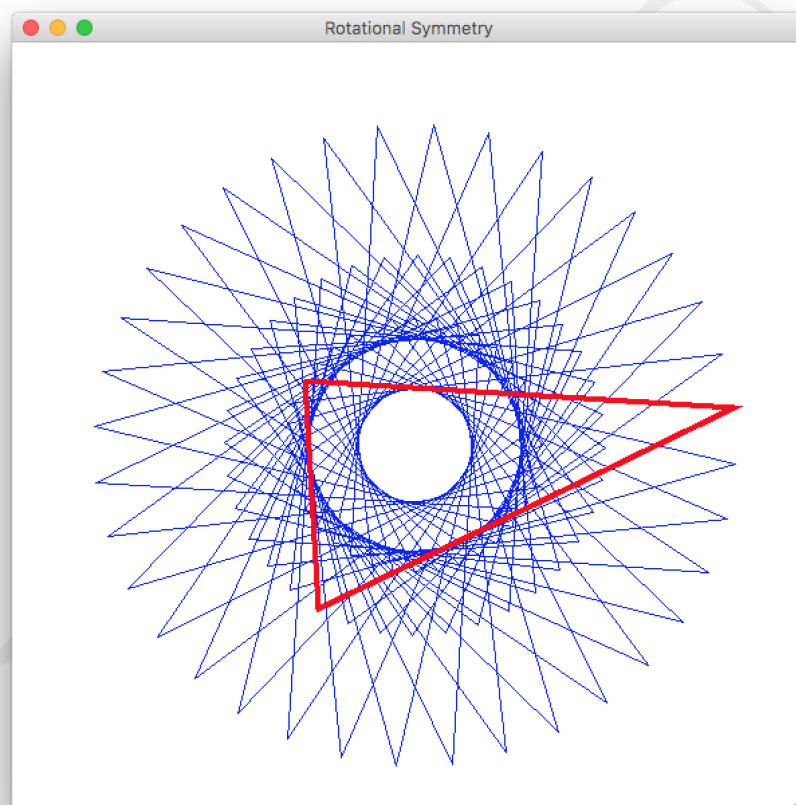


Figure 13.7: A symmetric figure resulting from Listing 13.8.

Function	Description
<code>DrawArc : Pen * Rectangle * Single * Single</code>	Draws an arc representing a portion of an ellipse specified by a <code>Rectangle</code> structure.
<code>DrawBezier : Pen * Point * Point * Point * Point</code>	Draws a Bézier spline defined by four <code>Point</code> structures.
<code>DrawCurve : Pen * Point[]</code>	Draws a cardinal spline through a specified array of <code>Point</code> structures.
<code>DrawEllipse : Pen * Rectangle</code>	Draws an ellipse specified by a bounding <code>Rectangle</code> structure.
<code>DrawImage : Image * Point[]</code>	Draws the specified <code>Image</code> at the specified location and with the specified shape and size.
<code>DrawLine : Pen * Point * Point</code>	Draws a series of line segments that connect an array of <code>Point</code> structures.
<code>DrawLines : Pen * Point[]</code>	Draws a series of line segments that connect an array of <code>Point</code> structures.
<code>DrawPie : Pen * Rectangle * Single * Single</code>	Draws a pie shape defined by an ellipse specified by a <code>Rectangle</code> structure and two radial lines.
<code>DrawPolygon : Pen * Point[]</code>	Draws a polygon defined by an array of <code>Point</code> structures.
<code>DrawRectangles : Pen * Rectangle[]</code>	Draws a series of rectangles specified by <code>Rectangle</code> structures.
<code>DrawString : String * Font * Brush * PointF</code>	Draws the specified text string at the specified location with the specified <code>Brush</code> and <code>Font</code> objects.
<code>FillClosedCurve : Brush * Point[]</code>	Fills the interior of a closed cardinal spline curve defined by an array of <code>Point</code> structures.
<code>FillEllipse : Brush * Rectangle</code>	Fills the interior of an ellipse defined by a bounding rectangle specified by a <code>Rectangle</code> structure.
<code>FillPie : Brush * Rectangle * Single * Single</code>	Fills the interior of a pie section defined by an ellipse specified by a <code>RectangleF</code> structure and two radial lines.
<code>FillPolygon : Brush * Point[]</code>	Fills the interior of a polygon defined by an array of points specified by <code>Point</code> structures.
<code>FillRectangle : Brush * Rectangle</code>	Fills the interior of a rectangle specified by a <code>Rectangle</code> structure.

Table 13.3: Some methods of the `System.Drawing.Graphics` class.

giving it a width. It is thus came as a surprise to many when Giuseppe Peano in 1890 demonstrated that there exists curves, which fill every point in a square. The method he used to achieve this was recursion.

Problem 13.2:

Consider a curve consisting of piecewise straight lines all with the same length but with varying angles 0° , 90° , 180° , or 270° w.r.t. the horizontal axis. To draw this curve we need 3 basic operations: Move forward (F), turn right ($+$), and turn left ($-$). The turning is w.r.t. the present direction. A Hilbert Curve is a space-filling curve, which be expressed recursively as:

$$A \rightarrow -BF + AFA + FB- \quad (13.1)$$

$$B \rightarrow +AF - BFB - FA+ \quad (13.2)$$

starting with A . For practical illustrations, we typically only draw space filling curves to a specified depth of recursion, which is called the order of the curve. Hence, to draw a first order curve, we don't recurse at all, i.e., ignore all occurrences of the symbols A and B on the right-hand-side of (13.1), and get

$$A \rightarrow -F + F + F - .$$

For the second order curve, we recurse once, i.e.,

$$\begin{aligned} A &\rightarrow -BF + AFA + FB- \\ &\rightarrow -(+AF - BFB - FA+)F \\ &\quad + (-BF + AFA + FB-)F(-BF + AFA + FB-) \\ &\quad + F(+AF - BFB - FA+)- \\ &\rightarrow AF - BFB - FA + FBF + AFA + FB - F - BF + AFA + FBF + AF - BFB - FA \\ &\rightarrow F - F - F + FF + F + F - F - F + F + FF + F - F - F. \end{aligned}$$

Make a program, that given an order produces an image of the Hilbert curve.

Our strategy will be to draw the curve sequentially from the origin to the end point. For this we will make a data structure `type curve = float * float * coordinates` length and the orientation of the next forward move and a partial list straight line pieces. The initial point would thus be `1.0, 0.0, [(0.0, 0.0)]`. We will also make 2 mutually recursive functions `ruleA : float curve -> curve` and `ruleB : float curve -> curve`, which takes n order and our data structure as input and for positive orders adds its rule to the end of the curve. We also need the three basic operations. To move forward, we will use the `rotatePoint` and `translatePoint` functions from Listing 13.7, that is, we may take line piece of specified length starting in the origin, rotate it according to the present orientation of the curve, and translate it to the present end-point of the curve. The two turn right and left only need to add or subtract 90° to the present orientation. Thus, reusing the top part of Listing 13.7 we arrive at the solutions shown in Listing 13.9. In Figure 13.8 are shown the result of using the program to draw Hilbert curves of orders 1, 2, 3, and 5.

Listing 13.9, winforms/hilbert.fsx:
Create the window and changing its properties.

```

44 type curve = float * float * coordinates
45
46 /// Turn 90 degrees left
47 let left (l, dir, c) : curve = (l, dir + 3.141592/2.0, c)
48
49 /// Turn 90 degrees right
50 let right (l, dir, c) : curve = (l, dir - 3.141592/2.0, c)
51
52 /// Add a line to the curve of present direction
53 let forward (l, dir, c) : curve =
54     let nextPoint = rotatePoint dir (l, 0.0)
55     (l, dir, c @ [translatePoint c.[c.Length-1] nextPoint])
56
57 /// Find the maximum value of each coordinate element in a list
58 let maximum (c : coordinates) =
59     let maxPoint p1 p2 =
60         (max (fst p1) (fst p2), max (snd p1) (snd p2))
61     List.fold maxPoint (-infinity, -infinity) c
62
63 /// Hilbert recursion production rules
64 let rec ruleA n C : curve =
65     if n > 0 then
66         (C |> left |> ruleB (n-1) |> forward |> right |> ruleA (n-1) |>
67          forward |> ruleA (n-1) |> right |> forward |> ruleB (n-1) |> left)
68     else
69         C
70 and ruleB n C : curve =
71     if n > 0 then
72         (C |> right |> ruleA (n-1) |> forward |> left |> ruleB (n-1) |>
73          forward |> ruleB (n-1) |> left |> forward |> ruleA (n-1) |> right)
74     else
75         C
76
77 // Calculate curve
78 let order = 5
79 let l = 20.0
80 let (_, dir, c) = ruleA order (l, 0.0, [(0.0, 0.0)])
81
82 // Setup drawing details
83 let title = "Hilbert's curve"
84 let backgroundColor = Color.White
85 let cMax = maximum c
86 let size = (int (fst cMax)+1, int (snd cMax)+1)
87 let polygLst = [(c, (Color.Black, 3.0))]
88
89 // Create form and start the event-loop.
90 let win = createForm backgroundColor size title (drawPoints polygLst)
91 Application.Run win

```

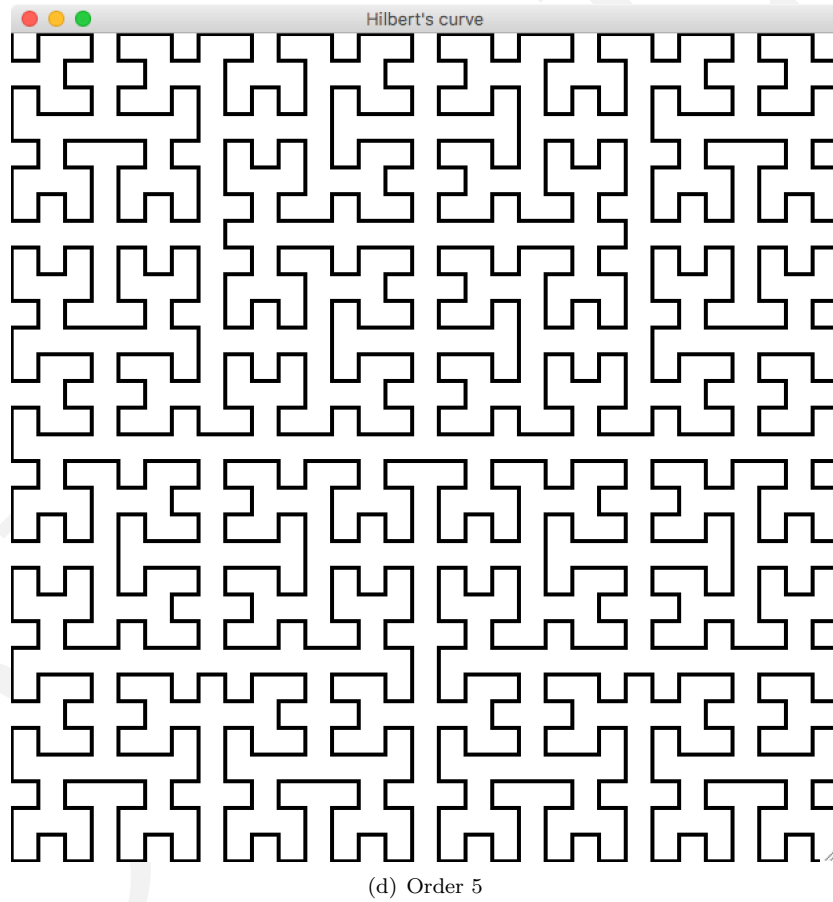
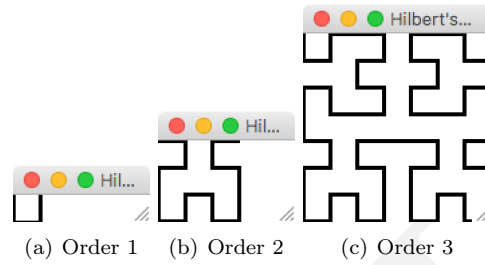


Figure 13.8: Hilbert curves of order 1, 2, 3, and 5 by code in Listing 13.9.

13.3 Events, Controls, and Panels

In the previous section, we have looked at how to draw graphics using the `Paint` method of an existing form object. Forms have many other event listeners, that we may use to interact with the user. Listing 13.11 demonstrates event handlers for moving and resizing a window, for clicking in a window, and for typing on the keyboard.

Listing 13.10, winforms/windowEvents.fsx:
Catching window, mouse, and keyboard events.

```
1 open System.Windows.Forms
2 open System.Drawing
3 open System
4
5 // create a form
6 let win = new Form ()
7
8 // Customize the window
9 win.Text <- "Window events"
10 win.BackColor <- Color.White
11 win.ClientSize <- Size (200, 200)
12 // Window event
13 let windowMove (e : EventArgs) =
14     printfn "Move: %A" win.Location
15 win.Move.Add windowMove
16 let windowResize (e : EventArgs) =
17     printfn "Resize: %A" win.DisplayRectangle
18 win.Resize.Add windowResize
19 // Mouse event
20 let mutable record = false;
21 let mouseMove (e : MouseEventArgs) =
22     if record then printfn "MouseMove: %A" e.Location
23 win.MouseMove.Add mouseMove
24 let mouseDown (e : MouseEventArgs) =
25     printfn "MouseDown: %A" e.Location; (record <- true)
26 win.MouseDown.Add mouseDown
27 let mouseUp (e : MouseEventArgs) =
28     printfn "MouseUp: %A" e.Location; (record <- false)
29 win.MouseUp.Add mouseUp
30 let mouseClicked (e : MouseEventArgs) =
31     printfn "MouseClicked: %A" e.Location
32 win.MouseClick.Add mouseClicked
33 // Keyboard event
34 win.KeyPreview <- true
35 let keyPress (e : KeyPressEventArgs) =
36     printfn "KeyPress: %A" (e.KeyChar.ToString ())
37 win.KeyPress.Add keyPress
38
39 // Start the event-loop.
40 Application.Run win
```

Listing 13.11: Output from an interaction with the program in Listing 13.11.

```
1 Move: {X=22,Y=22}
2 Move: {X=22,Y=22}
3 Move: {X=83,Y=161}
4 Resize: {X=0,Y=0,Width=275,Height=211}
5 MouseDown: {X=179,Y=123}
6 MouseClick: {X=179,Y=123}
7 MouseUp: {X=179,Y=123}
8 MouseDown: {X=179,Y=123}
9 MouseMove: {X=178,Y=123}
10 MouseMove: {X=174,Y=123}
11 MouseMove: {X=169,Y=123}
12 MouseMove: {X=164,Y=123}
13 MouseMove: {X=159,Y=123}
14 MouseMove: {X=154,Y=123}
15 MouseMove: {X=150,Y=123}
16 MouseMove: {X=146,Y=123}
17 MouseMove: {X=137,Y=123}
18 MouseMove: {X=130,Y=123}
19 MouseMove: {X=124,Y=122}
20 MouseMove: {X=119,Y=120}
21 MouseMove: {X=113,Y=118}
22 MouseMove: {X=109,Y=115}
23 MouseMove: {X=105,Y=112}
24 MouseMove: {X=101,Y=110}
25 MouseMove: {X=100,Y=109}
26 MouseMove: {X=97,Y=108}
27 MouseMove: {X=95,Y=107}
28 MouseClick: {X=95,Y=107}
29 MouseUp: {X=95,Y=107}
30 KeyPress: "f"
31 KeyPress: "s"
32 KeyPress: "h"
33 KeyPress: "a"
34 KeyPress: "r"
35 KeyPress: "p"
```

In Listing 13.11 is shown the output from an interaction with the program which is the result of the following actions: moving the window, resizing the window, clicking the left mouse key, pressing and holding the down the left mouse key while moving the mouse, and releasing the left mouse key, and type “fsharp”. As demonstrated, some actions like moving the mouse results in a lot of events, and some like the initial window moves results are surprising. Thus, event driven programming should take care to interpret the events robustly and carefully.

In WinForms buttons, menus and other interactive elements are called *Controls*. A form is a type of control, and thus, programming controls are very similar to programming windows. In Listing 13.12 is shown a small program that displays a button in a window, and when the button is pressed, then a dialogue is opened in another window. Figure 13.9 shows an example dialogue.

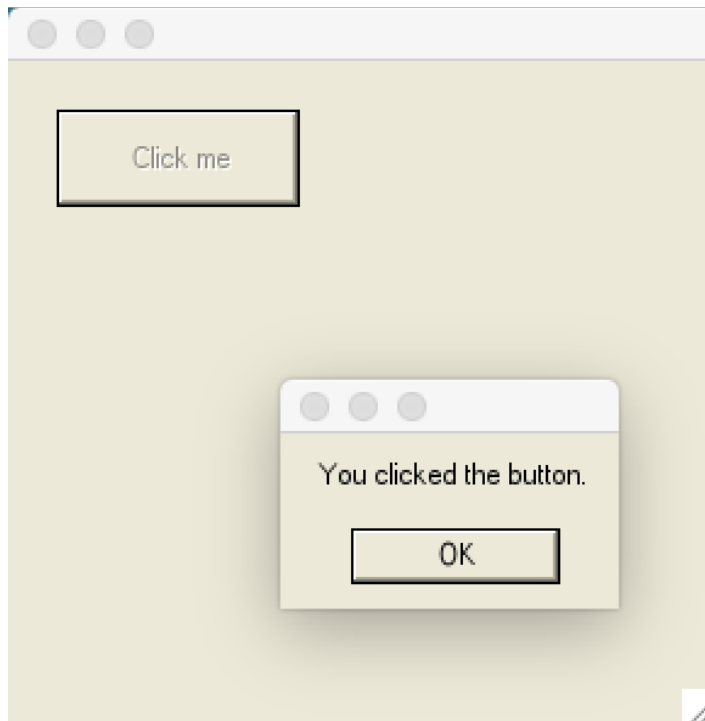


Figure 13.9: A button is pressed and the event handler calls the `MessageBox.Show` dialogue window by the code in Listing 13.12.

Listing 13.12, winforms/buttonControl.fsx:
Create the button and an event.

```

1 open System
2 open System.Windows.Forms
3 open System.Drawing
4
5 /// A button event
6 let buttonClicked (e : EventArgs) =
7     MessageBox.Show "You clicked the button." |> ignore
8
9 // Create a button
10 let button = new Button ()
11 button.Size <- new Size (100, 40)
12 button.Location <- new Point (20, 20)
13 button.Text <- "Click me"
14 button.Click.Add buttonClicked
15
16 // Create a window and add button
17 let win = new Form ()
18 win.Controls.Add button
19
20 // Start the event-loop.
21 Application.Run win

```

As the listing demonstrates, the button is created using the *System.Windows.Forms.Button* constructor, and it is added to the form's control list. It is possible to have many controls on each form, but

· `System.Windows.Forms.Button`

at times it is useful to organize the controls in groups. Such groups are called *Panels* in WinForms, and an example of creating a *System.Windows.Forms.Panel* that includes a *System.Windows.Forms.TextBox* and *System.Windows.Forms.Label* for getting user input is shown in Listing 13.13 and Figure 13.10.

Listing 13.13, winforms/panel.fsx:
Create a panel, label, text input controls.

```

1 open System.Drawing
2 open System.Windows.Forms
3
4 // Initialize a form containing a panel, textbox, and a label
5 let win = new Form ()
6 let panel = new Panel ()
7 let textBox = new TextBox ()
8 let label = new Label ()
9
10 // Customize the window
11 win.Text <- "Window with a panel"
12 win.ClientSize <- new Size (400, 300)
13
14 // Customize the Panel control
15 panel.Location <- new Point (56,72)
16 panel.Size <- new Size (264, 152)
17 panel.BorderStyle <- BorderStyle.Fixed3D
18
19 // Customize the Label and TextBox controls
20 label.Location <- new Point (16,16)
21 label.Text <- "label"
22 label.Size <- new Size (104, 16)
23 textBox.Location <- new Point (16,32)
24 textBox.Text <- "Initial text"
25 textBox.Size <- new Size (152, 20)
26
27 // Add panel to window and label and textBox to panel
28 win.Controls.Add panel
29 panel.Controls.Add label
30 panel.Controls.Add textBox
31
32 // Start the event-loop
33 Application.Run win

```

· Panels
· System.
Windows.Forms
.Panel
· System.
Windows.Forms
.TextBox
· System.
Windows.Forms
.Label

The label and textbox are children of the panel, and the main advantage of using panels is that the coordinates of the children are relative to the top left corner of the panel. I.e., moving the panel will move the label and the textbox at the same time.

Several types of panels exists in WinForms. A very flexible panel is the *System.Windows.Forms.FlowLayoutPanel*, which arranges its objects according to the space available. This is useful for graphical user interfaces targeting varying device sizes, such as a computer monitor and a tablet, and it also allows the program to gracefully adapt, when the user changes window size. A demonstration of *System.Windows.Forms.FlowLayoutPanel* together with *System.Windows.Forms.CheckBox* and *System.Windows.Forms.RadioButton* is given in Listing 13.15 and in Figure 13.11. The program illustrates how the button elements flow under four possible *System.Windows.FlowDirections*, and how *System.Windows.WrapContents* influences, what happens to content that flows outside the panels region.

· System.
Windows.Forms
.FlowLayoutPanel

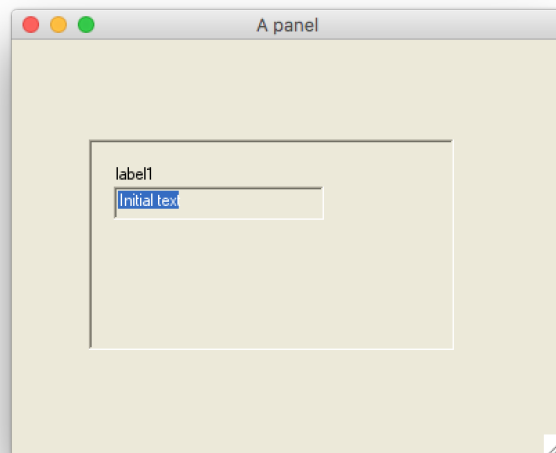


Figure 13.10: A panel including a label and a text input field, see Listing 13.13.

Listing 13.14, winforms/flowLayoutPanel.fsx:
Create a FlowLayoutPanel, with checkbox and radiobuttons.

```

1 open System.Windows.Forms
2 open System.Drawing
3
4 let flowLayoutPanel = new FlowLayoutPanel ()
5 let buttonLst =
6   [(new Button (), "Button0");
7    (new Button (), "Button1");
8    (new Button (), "Button2");
9    (new Button (), "Button3")]
10 let panel = new Panel ()
11 let wrapContentsCheckBox = new CheckBox ()
12 let initiallyWrapped = true
13 let radioButtonLst =
14   [(new RadioButton (), (3, 34), "TopDown", FlowDirection.TopDown);
15    (new RadioButton (), (3, 62), "BottomUp", FlowDirection.BottomUp);
16    (new RadioButton (), (111, 34), "LeftToRight", FlowDirection.
17     LeftToRight);
18    (new RadioButton (), (111, 62), "RightToLeft", FlowDirection.
19     RightToLeft)]
20
21 // customize buttons
22 for (btn, txt) in buttonLst do
23   btn.Text <- txt
24
25 // customize wrapContentsCheckBox
26 wrapContentsCheckBox.Location <- new Point (3, 3)
27 wrapContentsCheckBox.Text <- "Wrap Contents"
28 wrapContentsCheckBox.Checked <- initiallyWrapped
29 wrapContentsCheckBox.CheckedChanged.Add (fun _ -> flowLayoutPanel.
30   WrapContents <- wrapContentsCheckBox.Checked)
31
32 // customize radio buttons
33 for (btn, loc, txt, dir) in radioButtonLst do
34   btn.Location <- new Point (fst loc, snd loc)
35   btn.Text <- txt
36   btn.Checked <- flowLayoutPanel.FlowDirection = dir
37   btn.CheckedChanged.Add (fun _ -> flowLayoutPanel.FlowDirection <- dir)

```

Listing 13.15, winforms/flowLayoutPanel.fsx:
Create a FlowLayoutPanel, with checkbox and radiobuttons.

```
36 // customize flowLayoutPanel
37 for (btn, txt) in buttonLst do
38     flowLayoutPanel.Controls.Add btn
39 flowLayoutPanel.Location <- new Point (47, 55)
40 flowLayoutPanel.BorderStyle <- BorderStyle.Fixed3D
41 flowLayoutPanel.WrapContents <- initiallyWrapped
42
43 // customize panel
44 panel.Controls.Add (wrapContentsCheckBox)
45 for (btn, loc, txt, dir) in radioButtonLst do
46     panel.Controls.Add (btn)
47 panel.Location <- new Point (47, 190)
48 panel.BorderStyle <- BorderStyle.Fixed3D
49
50 // Create a window, add controls, and start event-loop
51 let win = new Form ()
52 win.ClientSize <- new Size (302, 356)
53 win.Controls.Add flowLayoutPanel
54 win.Controls.Add panel
55 win.Text <- "A Flowlayout Example"
56 Application.Run win
```

A walkthrough of the program is as follows. The goal is to make 2 areas, one giving the user control over display parameters, and another displaying the result of the user's choices. These are **FlowLayoutPanel** and **Panel**. In the **FlowLayoutPanel** there are four **Buttons**, to be displayed in a region, that is not tall enough for the buttons to be shown in vertical sequence and not wide enough to be shown in horizontal sequence. Thus the **FlowDirection** rules come into play, i.e., the buttons are added in sequence as they are named, and the default **FlowDirection.LeftToRight** arranges places the **buttonLst.[0]** in the top left corner, and **buttonLst.[1]** to its right. Other flow directions does it differently, and the reader is encourage to experiment with the program.

The program in Listing 13.15 has not completely separated the semantic blocks of the interface and relies on explicitly setting of coordinates of controls. This can be avoided by using nested panels. E.g., in Listing 13.17, the program has been rewritten as a nested set of **FlowLayoutPanel** in three groups: The button panel, the checkbox, and the radio button panel. Adding a **Resize** event handler for the window to resize the outermost panel according to the outer window, allows for the three groups to change position relative to each other, which results in three different views all shown in Figure 13.12.

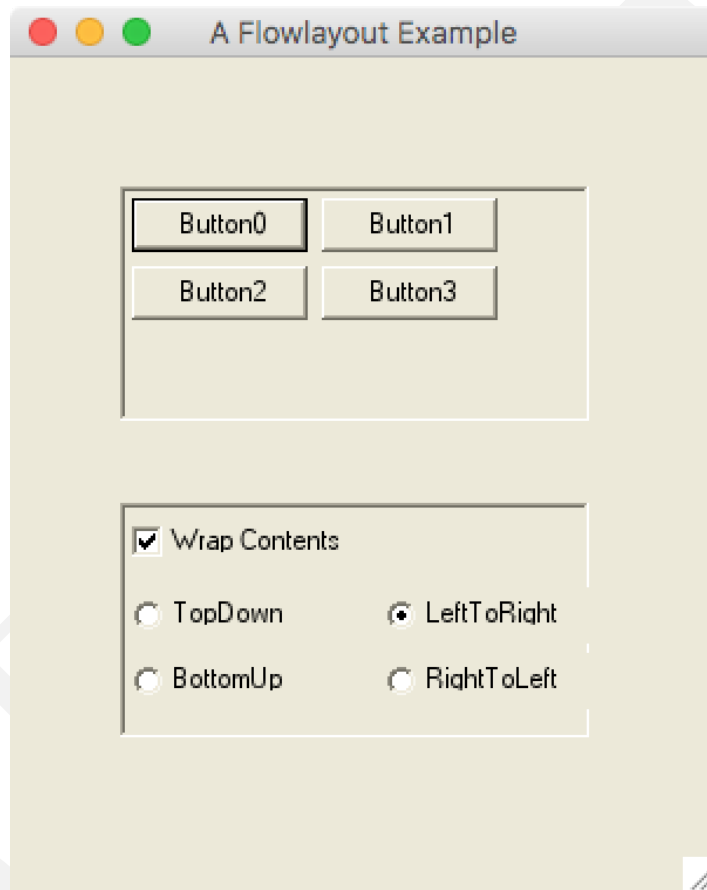


Figure 13.11: Demonstration of the `FlowLayoutPanel` panel, `CheckBox`, and `RadioButton` controls, see Listing 13.15.

**Listing 13.16, winforms/flowLayoutPanelAdvanced.fsx:
Create nested FlowLayoutPanel.**

```
1 open System.Windows.Forms
2 open System.Drawing
3 open System
4
5 let win = new Form ()
6 let mainPanel = new FlowLayoutPanel ()
7 let mainPanelBorder = 5
8 let flowLayoutPanel = new FlowLayoutPanel ()
9 let buttonLst =
10     [(new Button (), "Button0");
11      (new Button (), "Button1");
12      (new Button (), "Button2");
13      (new Button (), "Button3")]
14 let wrapContentsCheckBox = new CheckBox ()
15 let panel = new FlowLayoutPanel ()
16 let initiallyWrapped = true
17 let radioButtonLst =
18     [(new RadioButton (), "TopDown", FlowDirection.TopDown);
19      (new RadioButton (), "BottomUp", FlowDirection.BottomUp);
20      (new RadioButton (), "LeftToRight", FlowDirection.LeftToRight);
21      (new RadioButton (), "RightToLeft", FlowDirection.RightToLeft)]
22
23 // customize buttons
24 for (btn, txt) in buttonLst do
25     btn.Text <- txt
26
27 // customize radio buttons
28 for (btn, txt, dir) in radioButtonLst do
29     btn.Text <- txt
30     btn.Checked <- flowLayoutPanel.FlowDirection = dir
31     btn.CheckedChanged.Add (fun _ -> flowLayoutPanel.FlowDirection <- dir)
32
33 // customize flowLayoutPanel
34 for (btn, txt) in buttonLst do
35     flowLayoutPanel.Controls.Add btn
36 flowLayoutPanel.BorderStyle <- BorderStyle.Fixed3D
37 flowLayoutPanel.WrapContents <- initiallyWrapped
38
39 // customize wrapContentsCheckBox
40 wrapContentsCheckBox.Text <- "Wrap Contents"
41 wrapContentsCheckBox.Checked <- initiallyWrapped
42 wrapContentsCheckBox.CheckedChanged.Add (fun _ -> flowLayoutPanel.
    WrapContents <- wrapContentsCheckBox.Checked)
```

Listing 13.17, winforms/flowLayoutPanelAdvanced.fsx:
Create nested FlowLayoutPanel, see Listing 13.17.

```
44 // customize panel
45 // changing border style changes ClientSize
46 panel.BorderStyle <- BorderStyle.Fixed3D
47 let width = panel.ClientSize.Width / 2 - panel.Margin.Left - panel.Margin.
    Right
48 for (btn, txt, dir) in radioButtonLst do
49     btn.Width <- width
50     panel.Controls.Add (btn)
51
52 mainPanel.Location <- new Point (mainPanelBorder, mainPanelBorder)
53 mainPanel.BorderStyle <- BorderStyle.Fixed3D
54 mainPanel.Controls.Add flowLayoutPanel
55 mainPanel.Controls.Add wrapContentsCheckBox
56 mainPanel.Controls.Add panel
57
58 // Create a window, add controlse, and start event-loop
59 win.ClientSize <- new Size (220, 256)
60 let windowResize _ =
61     let size = win.DisplayRectangle.Size
62     mainPanel.Size <- new Size (size.Width - 2 * mainPanelBorder, size.
        Height - 2 * mainPanelBorder)
63 windowResize ()
64 win.Resize.Add windowResize
65 win.Controls.Add mainPanel
66 win.Text <- "Advanced Flowlayout"
67 Application.Run win
```

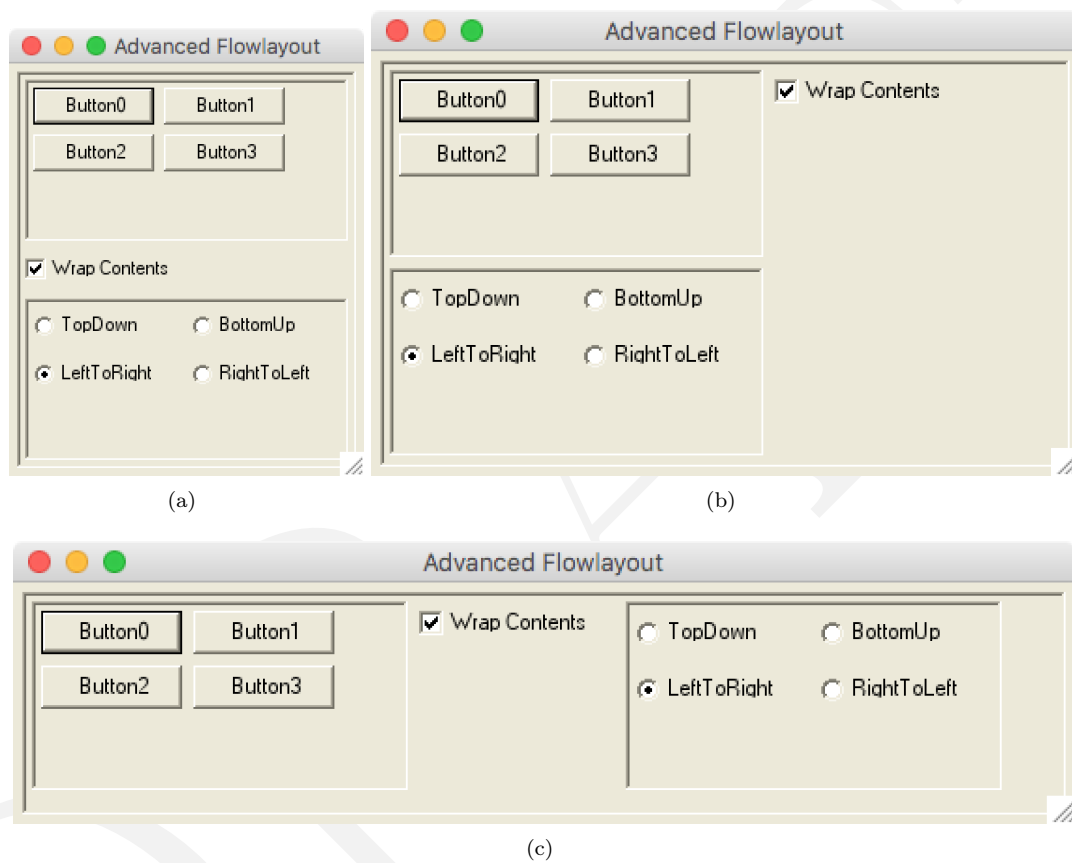


Figure 13.12: Nested `FlowLayoutPanel`, see Listing 13.17, allows for dynamic arrangement of content. Content flows, when the window is resized.

Bibliography

- [1] Alonzo Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 33(2):346–366, 1932.
- [2] Programming Research Group. Specifications for the ibm mathematical formula translating system, fortran. Technical report, Applied Science Division, International Business Machines Corporation, 1954.
- [3] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, 1960.
- [4] X3: ASA Sectional Committee on Computers and Information Processing. American standard code for information interchange. Technical Report ASA X3.4-1963, American Standards Association (ASA), 1963. <http://worldpowersystems.com/projects/codes/X3.4-1963/>.
- [5] George Pólya. *How to solve it*. Princeton University Press, 1945.

Index

`Paint.Add`, 131
`System.Drawing.Color`, 128
`System.Drawing.Graphics.DrawLine`, 131
`System.Drawing.PointToClient`, 130
`System.Drawing.PointToScreen`, 130
`System.Drawing`, 126
`System.Windows.Forms`, 126

accessors, 128

call-back function, 131

CLI, 126

client coordinates, 130

command-line interface, 126

event, 126

event driven programming, 126

event-loop, 127

forms, 126

GDI+, 126

graphical user interface, 126

GUI, 126

methods, 127

properties, 127

screen coordinates, 130

Structured programming, 133

Windows Graphics Device Interface, 126

WinForms, 126