23 Graphical User Interfaces

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

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

An example of a graphical user interface is a web-browser, as shown in Figure 23.1. The program presents 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 typing a URL, to follow hyperlinks, and to generate new pages by entering search queries.

F# includes a number of implementations of graphical user interfaces, and at time of writing both *GTK+* and *WinForms 2.0* are supported on both the Microsoft .Net and the Mono platform. WinForms can be used without extra libraries during compilation, and therefore will be the subject of the following chapter.

· GTK+
· WinForms 2.0

WinForms is a set of libraries that simplify many common tasks for applications, and in this chapter, we will focus on the graphical user interface part of WinForms. A *form* is a visual interface used to communicate information with the user, typically a window. Communication is done through *controls*, which are elements that displays information or accepts input. Examples of controls are a box with text, a button, and a menu. When the user gives input to a control element, this generates an *event*, which you can write code to react to. Winforms is designed for *event-driven programming*, meaning that at runtime, most time is spent on waiting for the user to give input. See Chapter 24 for more on event-driven programming.

· form
· control
· event
· event-driven programming

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.

23.1 Opening a window

Programming graphical user interface with WinForms uses the namespaces: *`System.Windows.Forms`* and *`System.Drawing`*. `System.Windows.Forms` includes code for generating forms, controls, and handling events. `System.Drawing` is used for low-level drawing, and it gives 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. All controls in `System.Windows.Forms` in Mono are drawn using `System.Drawing`.

· `System.Windows.Form`
· `System.Drawing`
· Windows Graphics Device Interface
· GDI+

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 23.1, and the result is shown in Figure 23.2. Note, the present version of WinForms on MacOs only works with the 32-bit implementation of mono, *`mono32`*, as demonstrated in the example.

· `mono32`

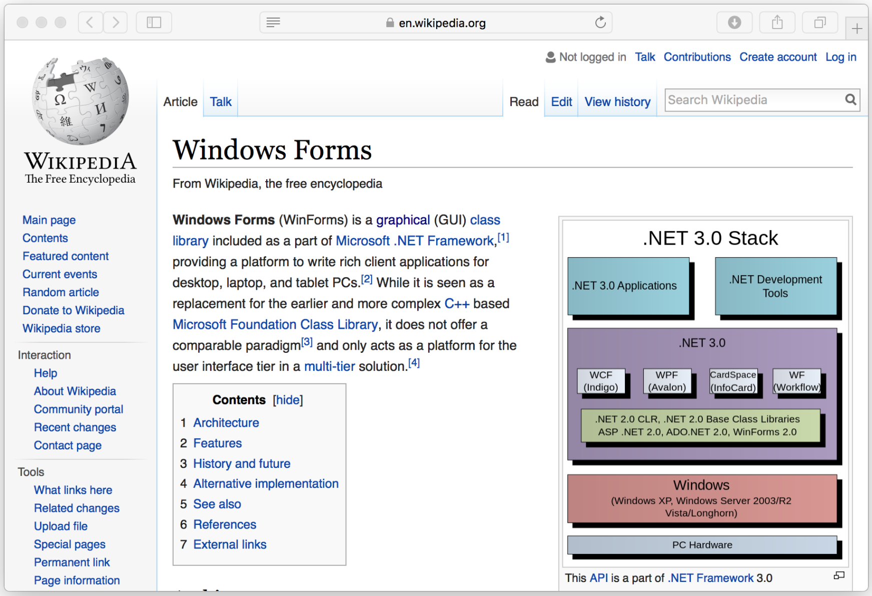


Figure 23.1: A web-browser is a graphical user interface for accessing a web-server and interacting with its services. Here the browser is showing the page https://en.wikipedia.org/wiki/Windows_Forms at time of writing.

Listing 23.1 winforms/openWindow.fsx:
Create the window and turn over control to the operating system. See Figure 23.2.

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

```
$ fsharpc --nologo openWindow.fsx && mono32 openWindow.exe
```

The `new System.Windows.Forms.Form ()` creates an object (See Chapter 20), but does not display the window on the screen. We use the optional `new` keyword since the form is an `IDisposable` object and may be implicitly disposed of. I.e., it is recommended to **instantiate `IDisposable` objects using `new` to contrast them with other object types.** Executing `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.

Advice

· event-loop

The window form 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 23.2.

· methods
· properties

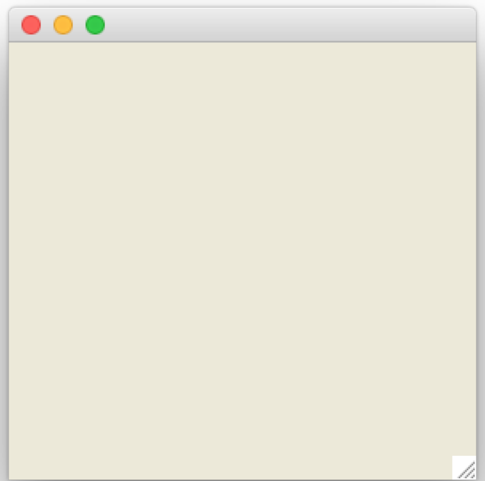


Figure 23.2: A window opened by Listing 23.1.

Listing 23.2 winforms/windowProperty.fsx:
Create the window and changing its properties. See Figure 23.3

```
// Prepare window form
let win = new System.Windows.Forms.Form ()

// Set some properties
win.BackColor <- System.Drawing.Color.White
win.Size <- System.Drawing.Size (600, 200)
win.Text <- sprintf "Color '%A' and Size '%A'" win.BackColor win.Size

// Start the event-loop.
System.Windows.Forms.Application.Run win
```

These properties are *accessors* implying that they act as mutable variables. · accessors

23.2 Drawing geometric primitives

The ***System.Drawing.Color*** is a structure for specifying colors as 4 channels: alpha, red, green, and blue. Some methods and properties for the Color structure is shown in Table 23.1. 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`, and `B` members, see Listing 23.3 · `Color`

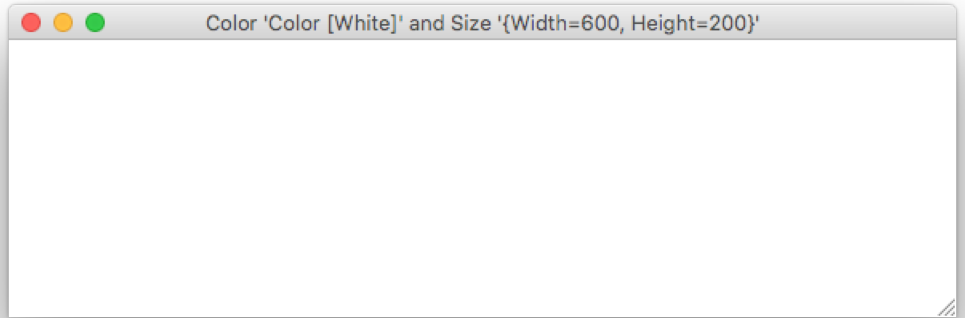


Figure 23.3: A window with user-specified size and background color, see Listing 23.2.

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 `0xFF000000`.
`Blue`	Get a predefined color with ARGB value of `0xFF0000FF`.
`Brown`	Get a predefined color with ARGB value of `0xFFA52A2A`.

`FromArgb : int -> Color`
`FromArgb : int*int*int -> Color`
`FromArgb : int*int*int*int -> Color`
	Create a color structure.

`G`	Get the value of the green channel of a color.
`Gray`	Get a predefined color with ARGB value of `0xFF808080`.
`Green`	Get a predefined color with ARGB value of `0xFF00FF00`.
`Orange`	Get a predefined color with ARGB value of `0xFFFFA500`.
`Purple`	Get a predefined color with ARGB value of `0xFF800080`.
`R`	Get the value of the red channel of a color.
`Red`	Get a predefined color with ARGB value of `0xFFFF0000`.
`ToArgb : Color -> int`	Get the 32-bit integer representation of a color.
`White`	Get a predefined color with ARGB value of `0xFFFFFFFF`.
`Yellow`	Get a predefined color with ARGB value of `0xFFFFFF00`.

Table 23.1: Some methods and properties of the `System.Drawing.Color` structure.

Constructor	Description
`Bitmap(int, int)`	Create a new empty `Image` of specified size.
`Bitmap(Stream)` `Bitmap(string)`	Create a `Image` from a `System.IO.Stream` or from a file specified by a filename.
`Font(string, single)`	Create a new font from the font's name and em-size.

`Pen(Brush)`
`Pen(Brush), single)`
`Pen(Color)`
`Pen(Color, single)`
	Create a pen to paint either with a brush or solid color and possibly with specified width.

`Point(int, int)`
`Point(Size)`
`PointF(single, single)`
	Create an ordered pair of integers or singles specifying x- and y-coordinates in the plane.

`Size(int, int)`
`Size(Point)`
`SizeF(single, single)`
`SizeF(PointF)`
	Create an ordered pair of integers or singles specifying height and width in the plane.

`SolidBrush(Color)` `TextureBrush(Image)`	Create a `Brush` as a solid color or from an image to fill the interior of a geometric shapes.

Table 23.2: Basic geometrical structures in WinForms. `Brush` and `Image` are abstract classes.

Listing 23.3 drawingColors.fsx:
Defining colors and accessing their values.

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

```
$ fsharpc --nologo drawingColors.fsx && mono drawingColors.exe
The color Color [A=255, R=127, G=255, B=212] is (ff, 7f, ff, d4)
The color Color [Red] is (ff, ff, 0, 0)
The color Color [Green] is (ff, 0, 80, 0)
The color Color [Blue] is (ff, 0, 0, ff)
The color Color [Black] is (ff, 0, 0, 0)
The color Color [Gray] is (ff, 80, 80, 80)
The color Color [White] is (ff, ff, ff, ff)
```

`System.Drawing` contains many often used structures. Listing 23.2 used `System.Drawing.Size` to specify a size by a pair of integers. Other important structures in WinForms are *`System.Drawing.Point`*, which specifies a coordinate as a pair of points; *`System.Drawing.Pen`*, which specifies how to draw lines and curves; *`System.Drawing.Font`*, which specifies the font of a string; *`System.Drawing.SolidBrush`* and *`System.Drawing.TextureBrush`*, used for filling geometric primitives, and *`System.Drawing.Bitmap`*, which is a type of *`System.Drawing.Image`*. These are summarized in Table 23.2. [1]

· `Point`
· `Pen`
· `Font`
· `SolidBrush`
· `TextureBrush`
· `Bitmap`
· `Image`

Constructor	Description
`DrawImage : Image * (Point []) -> unit` `DrawImage : Image * (PointF []) -> unit`	Draw an image at a specific point and size.
`DrawImage : Image * Point -> unit` `DrawImage : Image * PointF -> unit`	Draw an image at a specific point.
`DrawLines : Pen * (Point []) -> unit` `DrawLines : Pen * (PointF []) -> unit`	Draw a series of lines between the n'th and the $n + 1$'th points.
`DrawString : string * Font * Brush * PointF -> unit`	Draw a string at the specified point.

Table 23.3: Basic geometrical structures in WinForms.

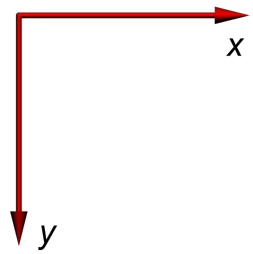


Figure 23.4: Coordinate systems in Winforms have the y axis pointing down.

The ***System.Drawing.Graphics*** is a class for drawing geometric primitives to a display device, and some of its methods are summarized in Table 23.3. · `Graphics`

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*. Both coordinate systems have their origin in the top-left corner, with the first coordinate, x, increases to the right, and the second, y, increases down as illustrated in Figure 23.4. The Screen coordinate system has its origin in the top-left corner of the screen, while the client coordinate system has its origin in the top-left corner of 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 a line like primitives and the brush for filled regions. · screen coordinates · client coordinates · `PointToClient` · `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 form's ***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.DrawLines*** 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` · `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 23.4.

[1] Jon: **Do something about the vertical alignment of minpage.**

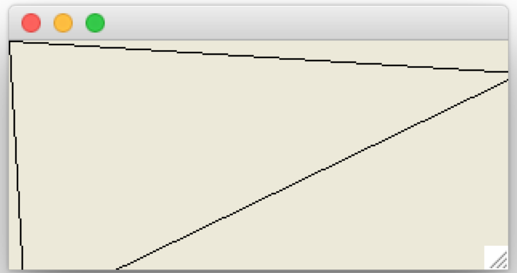


Figure 23.5: Drawing a triangle using Listing 23.4.

Listing 23.4 winforms/triangle.fsx:
Adding line graphics to a window. See Figure 23.5

```
// Open often used libraries, be ware of namespace polution!
open System.Windows.Forms
open System.Drawing

// Prepare window form
let win = new Form ()
win.Size <- Size (320, 170)

// Set paint call-back function
let paint (e : PaintEventArgs) : unit =
  let pen = new Pen (Color.Black)
  let points =
    [|Point (0,0); Point (10,170); Point (320,20); Point (0,0)|]
  e.Graphics.DrawLines (pen, points)
win.Paint.Add paint

// Start the event-loop.
Application.Run win // Start the event-loop.
```

A walk-through of the code is as follows: First, we open the two libraries that we will use heavily. This will save us some typing but also pollute our namespace. E.g., now `Point` and `Color` are existing types, and we cannot define our own identifier with these names. Then we create the form with size 320×170, we add a paint call-back function and we start the event-loop. The event-loop will call the paint function, whenever the system determines that the window's content needs to be refreshed. 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. The function `paint` chooses a pen and a set of points and draws a set of lines connecting the points.

The code in Listing 23.4 is not optimal. In spite that the triangle spans the rectangle $(0,0)$ to $(320,170)$ and the window's size is set to $(320,170)$, then our window is too small and the triangle is clipped at the window border. The error is that we set the window's `Size` property which determines the size of the window including top bar and borders. Alternatively we may set the `ClientSize`, which determines the size of the drawable area, and this is demonstrated in Listing 23.5 and Figure 23.6.

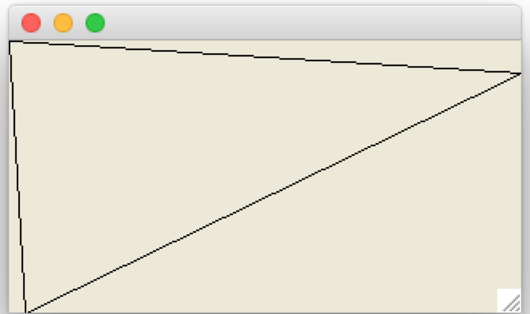


Figure 23.6: Setting the ClientSize property gives a predictable drawing area, see Listing 23.5 for code.

Listing 23.5 winforms/triangleClientSize.fsx:
Adding line graphics to a window. See Figure 23.6.

```
// Open often used libraries, be ware of namespace polution!
open System.Windows.Forms
open System.Drawing

// Prepare window form
let win = new Form ()
win.ClientSize <- Size (320, 170)

// Set paint call-back function
let paint (e : PaintEventArgs) : unit =
  let pen = new Pen (Color.Black)
  let points =
    [|Point (0,0); Point (10,170); Point (320,20); Point (0,0)|]
  e.Graphics.DrawLines (pen, points)
win.Paint.Add paint

// Start the event-loop.
Application.Run win // Start the event-loop.
```

Thus, **prefer the `ClientSize` over the `Size` property for internal consistency.** Advice

Considering the program in Listing 23.4, we may identify a part that concerns the specification of the triangle, or more generally the graphical model, and some which concerns system specific details. For future maintenance, it is often a good idea to **separate the model from how it is viewed on a specific system**. 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. We think of the model and the viewing part of the code as top and bottom layers, and these are often connected with a connection layer. This *Model-View paradigm* is shown in Figure 23.7. 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.

Advice · Model-View paradigm

In Listing 23.6, the program has been redesigned to follow the Model-View paradigm, to make use of a Winforms and a model specific function called `view` and `model`.

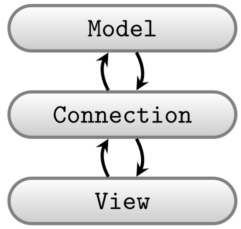


Figure 23.7: Separating model from view gives flexibility later.

Listing 23.6 winforms/triangleOrganized.fsx:
Improved organization of code for drawing a triangle. See Figure 23.8.

```
// Open often used libraries, be ware of namespace polution!
open System.Windows.Forms
open System.Drawing

///////////// WinForm specifics /////////////
/// Setup a window form and return function to activate
let view (sz : Size) (pen : Pen) (pts : Point []) : (unit -> unit) =
  let win = new System.Windows.Forms.Form ()
  win.ClientSize <- sz
  win.Paint.Add (fun e -> e.Graphics.DrawLines (pen, pts))
  fun () -> Application.Run win // function as return value

///////////// Model /////////////
// A black triangle, using winform primitives for brevity
let model () : Size * Pen * (Point []) =
  let size = Size (320, 170)
  let pen = new Pen (Color.FromArgb (0, 0, 0))
  let lines =
    [|Point (0,0); Point (10,170); Point (320,20); Point (0,0)|]
  (size, pen, lines)

///////////// Connection //////////////
// Tie view and model together and enter main event loop
let (size, pen, lines) = model ()
let run = view size pen lines
run ()
```

This program is longer, but there is a much better separation of *what* is to be displayed (model) from the *how* it is to be done (view).

To further our development of a general program for displaying graphics, consider the case, where we are to draw another two triangles, that are a translation and a rotations of the original, and where we would like to specify the color of each triangle individually. A simple extension of `model` in Listing 23.6 for generating many shapes of different colors is `model : unit -> Size * ((Point []) * Pen) list`, i.e., semantically augment each point array with a pen and return a list of such pairs. For this example we also program translation and rotation transformations. See Listing 23.7 for the result.

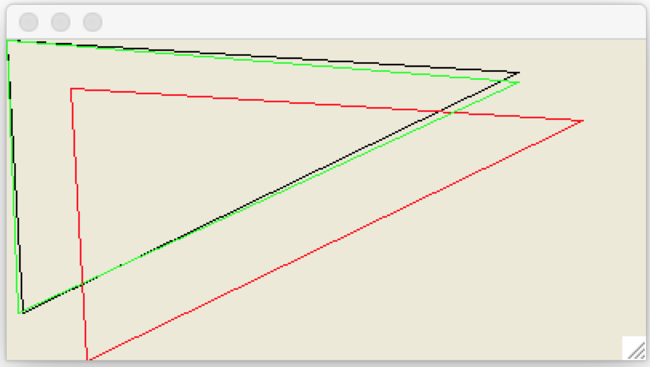


Figure 23.8: Better organization of the code for drawing a triangle, see Listing 23.6.

Listing 23.7 winforms/transformWindows.fsx:
Model of a triangle and simple transformations of it. See also Listing 23.8 and 23.9, and Figure 23.9.

```
///////////// Model /////////////
// A black triangle, using winform primitives for brevity
let model () : Size * ((Pen * (Point [])) list) =
  /// Translate a primitive
  let translate (d : Point) (arr : Point []) : Point [] =
    let add (d : Point) (p : Point) : Point =
      Point (d.X + p.X, d.Y + p.Y)
    Array.map (add d) arr

  /// Rotate a primitive
  let rotate (theta : float) (arr : Point []) : Point [] =
    let toInt = int << round
    let rot (t : float) (p : Point) : Point =
      let (x, y) = (float p.X, float p.Y)
      let (a, b) = (x * cos t - y * sin t, x * sin t + y * cos t)
      Point (toInt a, toInt b)
    Array.map (rot theta) arr

  let size = Size (400, 200)
  let lines =
    [|Point (0,0); Point (10,170); Point (320,20); Point (0,0)|]
  let black = new Pen (Color.FromArgb (0, 0, 0))
  let red = new Pen (Color.FromArgb (255, 0, 0))
  let green = new Pen (Color.FromArgb (0, 255, 0))
  let shapes =
    [(black, lines);
     (red, translate (Point (40, 30)) lines);
     (green, rotate (1.0 *System.Math.PI / 180.0) lines)]
  (size, shapes)
```

We update `view` accordingly to iterate through this list as shown in Listing 23.8.

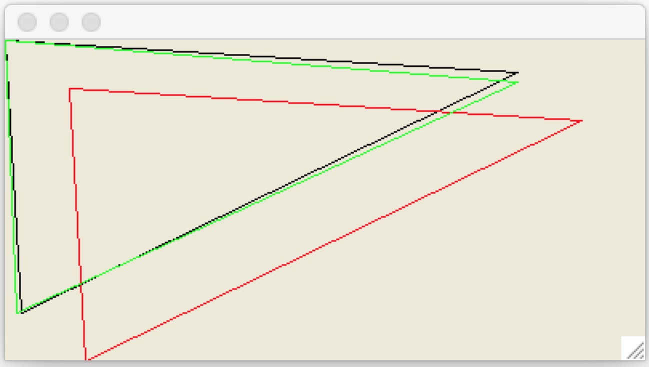


Figure 23.9: Transformed versions of the same triangle resulting from running the code in Listing 23.7–23.9.

Listing 23.8 winforms/transformWindows.fsx:
View of lists of pairs of pen and point arrays. See also Listing 23.7 and 23.9, and Figure 23.9.

```
// Open often used libraries, be ware of namespace polution!
open System.Windows.Forms
open System.Drawing

///////////// WinForm specifics /////////////
/// Setup a window form and return function to activate
let view (sz : Size) (shapes : (Pen * (Point [])) list) : (unit -> unit)
  =
  let win = new System.Windows.Forms.Form ()
  win.ClientSize <- sz
  let paint (e : PaintEventArgs) ((p, pts) : (Pen * (Point []))) : unit =
    e.Graphics.DrawLines (p, pts)
  win.Paint.Add (fun e -> List.iter (paint e) shapes)
  fun () -> Application.Run win // function as return value
```

Since we are using Winforms primitives in the model, then the connection layer is trivial as shown in Listing 23.9.

Listing 23.9 winforms/transformWindows.fsx:
Model of a triangle and simple transformations of it. See also Listing 23.7 and 23.8, and Figure 23.9.

```
///////////// Connection //////////////
// Tie view and model together and enter main event loop
let (size, shapes) = model ()
let run = view size shapes
run ()
```

23.3 Programming intermezzo: Hilbert Curve

A curve in 2 dimensions has a length but no width, and we can only visualize it by giving it a width. Thus, it came as a surprise to many when Giuseppe Peano in 1890 demonstrated that there exist curves, which fill every point in a square. The method he used to achieve this was recursion:

Problem 23.1

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 (R), and turn left (L). The turning is w.r.t. the present direction. A Hilbert Curve is a space-filling curve, which can be expressed recursively as:

$$A \to LBFRAFARFBL \tag{23.1}$$
$$B \to RAFLBFBLFAR \tag{23.2}$$

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 (23.1), and get

$$A \to LFRFRFL.$$

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

$$\begin{aligned} A \to & LBFRAFARFBL \\ \to & L(RAFLBFBLFAR)F \\ & \quad R(LBFRAFARFBL)F(LBFRAFARFBL) \\ & \quad\quad RF(RAFLBFBLFAR)L \\ \to & LRFLFLFRFRLFRFRFLFLFRFRFLRFRFLFLFRL. \end{aligned}$$

Since $LR = RL = \emptyset$ then the above simplifies to $FLFLFRFFRFRFLFLFRFRFFRFLFLF$. Make a program, that given an order produces an image of the Hilbert curve.

Our strategy to solve this problem will be first to define the curves in terms of movement commands $LRFL\ldots$. For this, we will define a discriminated union `type Command = F | L | R`. The movement commands can then be defined as a `Command list` type. The list for a specific order is a simple set of recursive functions in F#, which we will call `A` and `B`.

To produce a graphical drawing of a command list, we must transform it into coordinates, and during the conversion, we need to keep track of both the present position and the present heading, since not all commands draw. This is a concept similar to Turtle Graphics, which is often associated with the Logo programming language from the 1960's. In Turtle graphics, we command a little robot - a turtle - which moves in 2 dimensions and can turn on the spot or move forward, and its track is the line being drawn. Thus we introduce a `type Turtle = {x : float; y : float; d : float}` record. Conversion of command lists to turtle lists is a fold programming structure, where the command list is read from left-to-right, building up an accumulator by adding each new element. For efficiency, we choose to prepend the new element to the accumulator. This we have implemented as the `addRev` function. Once the full list of turtles has been produced, then it is reversed.

Finally, the turtle list is converted to Winforms `Point` array, and a window of appropriate size is chosen. The resulting model part is shown in Listing 23.10. The view and connection parts are identical to Listing 23.8 and 23.9, and Figure 23.10 shows the result of using the program to draw Hilbert curves of orders 1, 2, 3, and 5.

Listing 23.10 winforms/hilbert.fsx:
Using simple turtle graphics to produce a list of points on a polygon. The view and connection parts are identical to Listing 23.8 and 23.9.

```
// Turtle commands, type definitions must be in outer most scope
type Command = F | L | R
type Turtle = {x : float; y : float; d : float}

// A black Hilbert curve using winform primitives for brevity
let model () : Size * ((Pen * (Point [])) list) =
  /// Hilbert recursion production rules
  let rec A n : Command list =
    if n > 0 then
      [L]@B (n-1)@[F; R]@A (n-1)@[F]@A (n-1)@[R; F]@B (n-1)@[L]
    else
      []
  and B n : Command list =
    if n > 0 then
      [R]@A (n-1)@[F; L]@B (n-1)@[F]@B (n-1)@[L; F]@A (n-1)@[R]
    else
      []

  /// Convert a command to a turtle record and prepend to a list
  let addRev (lst : Turtle list) (cmd : Command) (len : float) : Turtle list =
    let toInt = int << round
    match lst with
      | t::rest ->
        match cmd with
          | L -> {t with d = t.d + 3.141592/2.0}::rest // turn left
          | R -> {t with d = t.d - 3.141592/2.0}::rest // turn right
          | F -> {t with x = t.x + len * cos t.d;
                           y = t.y + len * sin t.d}::lst // forward
      | _ -> failwith "Turtle list must be non-empty."

  let maxPoint (p1 : Point) (p2 : Point) : Point =
    Point (max p1.X p2.X, max p1.Y p2.Y)

  // Calculate commands for a specific order
  let curve = A 5
  // Convert commands to point array
  let initTrtl = {x = 0.0; y = 0.0; d = 0.0}
  let len = 20.0
  let line =
    // Convert command list to reverse turtle list
    List.fold (fun acc elm -> addRev acc elm len) [initTrtl] curve
    // Reverse list
    |> List.rev
    // Convert turtle list to point list
    |> List.map (fun t -> Point (int (round t.x), int (round t.y)))
    // Convert point list to point array
    |> List.toArray
  let black = new Pen (Color.FromArgb (0, 0, 0))
  // Set size to as large as shape
  let minVal = System.Int32.MinValue
  let maxPoint = Array.fold maxPoint (Point (minVal, minVal)) line
  let size = Size (maxPoint.X + 1, maxPoint.Y + 1)
  // return shapes as singleton list
  (size, [(black, line)])
```

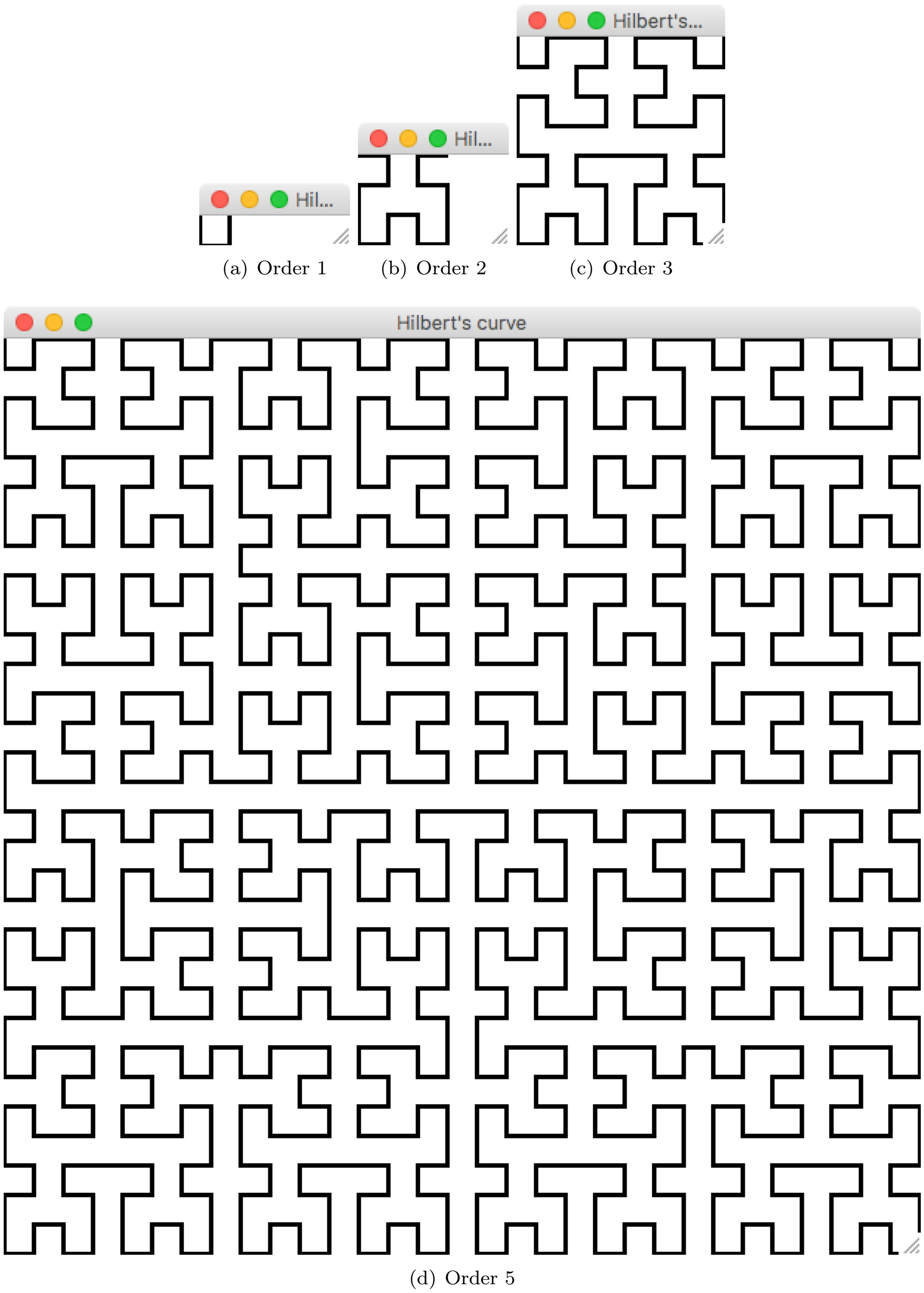


Figure 23.10: Hilbert curves of order 1, 2, 3, and 5 by code in Listing 23.10.

23.4 Handling events

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 handlers, that we may use to interact with the user. Listing 23.11 demonstrates event handlers for moving and resizing a window, for clicking in a window, and for typing on the keyboard.

**Listing 23.11 winforms/windowEvents.fsx:
Catching window, mouse, and keyboard events.**

```
open System.Windows.Forms
open System.Drawing
open System

let win = new Form () // create a form

// Window event
let windowMove (e : EventArgs) =
  printfn "Move: %A" win.Location
win.Move.Add windowMove

let windowResize (e : EventArgs) =
  printfn "Resize: %A" win.DisplayRectangle
win.Resize.Add windowResize

// Mouse event
let mutable record = false; // records when button down
let mouseMove (e : MouseEventArgs) =
  if record then printfn "MouseMove: %A" e.Location
win.MouseMove.Add mouseMove

let mouseDown (e : MouseEventArgs) =
  printfn "MouseDown: %A" e.Location; (record <- true)
win.MouseDown.Add mouseDown

let mouseUp (e : MouseEventArgs) =
  printfn "MouseUp: %A" e.Location; (record <- false)
win.MouseUp.Add mouseUp

let mouseClick (e : MouseEventArgs) =
  printfn "MouseClick: %A" e.Location
win.MouseClick.Add mouseClick

// Keyboard event
win.KeyPreview <- true
let keyPress (e : KeyPressEventArgs) =
  printfn "KeyPress: %A" (e.KeyChar.ToString ())
win.KeyPress.Add keyPress

Application.Run win // Start the event-loop.
```

Listing 23.12: Output from an interaction with the program in Listing 23.11.

```
Move: {X=22,Y=22}
Move: {X=22,Y=22}
Move: {X=50,Y=71}
Resize: {X=0,Y=0,Width=307,Height=290}
MouseDown: {X=144,Y=118}
MouseClick: {X=144,Y=118}
MouseUp: {X=144,Y=118}
MouseDown: {X=144,Y=118}
MouseUp: {X=144,Y=118}
MouseDown: {X=96,Y=66}
MouseMove: {X=96,Y=67}
MouseMove: {X=97,Y=69}
MouseMove: {X=99,Y=71}
MouseMove: {X=103,Y=74}
MouseMove: {X=107,Y=77}
MouseMove: {X=109,Y=79}
MouseMove: {X=112,Y=81}
MouseMove: {X=114,Y=82}
MouseMove: {X=116,Y=84}
MouseMove: {X=117,Y=85}
MouseMove: {X=118,Y=85}
MouseClick: {X=118,Y=85}
MouseUp: {X=118,Y=85}
KeyPress: "a"
KeyPress: "b"
KeyPress: "c"
```

In Listing 23.11 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 "abc". 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.

Common for all event-handlers is that they listen for an event, and when the event occurs, then the functions that have been added using the `Add` method are called. This is also known as sending a message. Thus, a single event can give rise to calling zero or more functions.

Graphical user interfaces and other systems often need to perform actions that depends on specific lengths of time or at a certain point in time. To time events, F# has the `System.Timer` class, which has been optimized for graphical user interfaces as ***`System.Windows.Forms.Timer`***, and which can be set to create an event after a specified duration of time. F# also has the ***`System.DateTime`*** class to specify points in time. An often used property is `System.DateTime.Now`, which returns a `DateTime` object for the date and time, when the property is accessed. The use of these two classes is demonstrated in Listing 23.13 and Figure 23.11.

· `Timer`
· `DateTime`

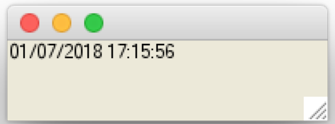


Figure 23.11: See Listing 23.13.

Listing 23.13 winforms/clock.fsx:
Using `System.Windows.Forms.Timer` and `System.DateTime.Now` to update the display of the present date and time. See Figure 23.11 for the result.

```
open System.Windows.Forms
open System.Drawing
open System

let win = new Form () // make a window form
win.ClientSize <- Size (200, 50)

// make a label to show time
let label = new Label()
win.Controls.Add label
label.Width <- 200
label.Text <- string System.DateTime.Now // get present time and date

// make a timer and link to label
let timer = new Timer()
timer.Interval <- 1000 // create an event every 1000 millisecond
timer.Enabled <- true // activiate the timer
timer.Tick.Add (fun e -> label.Text <- string System.DateTime.Now)

Application.Run win // start event-loop
```

In the code, a label has been created to show the present date and time. The label is a type of control, and it is displayed using the default font, which is rather small. How to change this and other details on controls will be discussed in the next section.

23.5 Labels, buttons, and pop-up windows

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 23.14 is shown a small program that displays a label and a button in a window, and when the button is pressed, then the label is updated. · control

Figure 23.12: After pressing the button 3 times. See Listing 23.14.

Listing 23.14 winforms/buttonControl.fsx:
Create the button and an event, see also Figure 23.12.

```
open System.Windows.Forms
open System.Drawing
open System

let win = new Form () // make a window form
win.ClientSize <- Size (140, 120)

// Create a label
let label = new Label()
win.Controls.Add label
label.Location <- new Point (20, 20)
label.Width <- 120
let mutable clicked = 0
let setLabel clicked =
  label.Text <- sprintf "Clicked %d times" clicked
setLabel clicked

// Create a button
let button = new Button ()
win.Controls.Add button
button.Size <- new Size (100, 40)
button.Location <- new Point (20, 60)
button.Text <- "Click me"
button.Click.Add (fun e -> clicked <- clicked + 1; setLabel clicked)

Application.Run win // Start the event-loop.
```

As the listing demonstrates, the button is created using the *`System.Windows.Forms.Button`* constructor, and it is added to the window's form's control list. The `Location` property controls its position w.r.t. the enclosing form. Other accessors are `Width`, `Text`, and `Size`.

· `Button`

`System.Windows.Forms` includes a long list of controls, some of which are summarized in Table 23.4. Examples are given in `controls` is shown in Listing 23.15 and Figure 23.13.

· `CheckBox`
· `DateTimePicker`
· `Label`
· `ProgressBar`
· `RadioButton`
· `TextBox`

Method/Property	Description
`Button`	A clickable button.
`CheckBox`	A clickable check box.
`DateTimePicker`	A box showing a date with a drop-down menu for choosing another.
`Label`	A displayable text.
`ProgressBar`	A box showing a progress bar.
`RadioButton`	A single clickable radio button. Can be paired with other radio buttons.
`TextBox`	A text area, which can accept input from the user.

Table 23.4: Some types of `System.Windows.Forms.Control`.

Listing 23.15 winforms/controls.fsx:
Examples of control elements added to a window form, see also Figure 23.13.

```
open System.Windows.Forms
open System.Drawing

let win = new Form ()
win.ClientSize <- Size (300, 300)

let button = new Button ()
win.Controls.Add button
button.Location <- new Point (20, 20)
button.Text <- "Click me"

let lbl = new Label ()
win.Controls.Add lbl
lbl.Location <- new Point (20, 60)
lbl.Text <- "A text label"

let chkbox = new CheckBox ()
win.Controls.Add chkbox
chkbox.Location <- new Point (20, 100)

let pick = new DateTimePicker ()
win.Controls.Add pick
pick.Location <- new Point (20, 140)

let prgrss = new ProgressBar ()
win.Controls.Add prgrss
prgrss.Location <- new Point (20, 180)
prgrss.Minimum <- 0
prgrss.Maximum <- 9
prgrss.Value <- 3

let rdbttn = new RadioButton ()
win.Controls.Add rdbttn
rdbttn.Location <- new Point (20, 220)

let txtbox = new TextBox ()
win.Controls.Add txtbox
txtbox.Location <- new Point (20, 260)
txtbox.Text <- "Type something"

Application.Run win
```

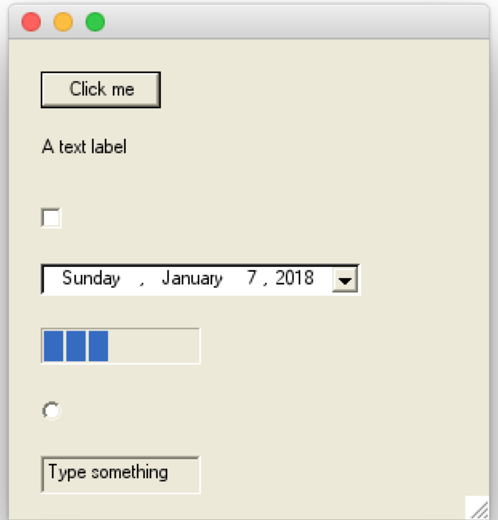


Figure 23.13: Examples of control elements. See Listing 23.15.

Some controls opens separate windows for more involved dialogue with the user. Some examples are `MessageBox`, `OpenFileDialog`, and `SaveFileDialog`.

`System.Windows.Forms.MessageBox` is used to have a simple but restrictive dialogue with the user is · `MessageBox` needed, which is demonstrated in Listing 23.16 and Figure 23.14.

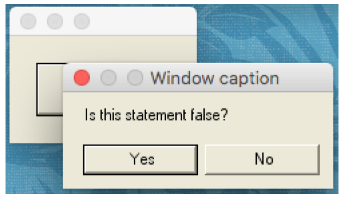


Figure 23.14: After pressing the button 3 times. See Listing 23.16.

**Listing 23.16 winforms/messageBox.fsx:
Create the `MessageBox`, see also Figure 23.14.**

```
open System.Windows.Forms
open System.Drawing
open System

let win = new Form ()
win.ClientSize <- Size (140, 80)

let button = new Button ()
win.Controls.Add button
button.Size <- new Size (100, 40)
button.Location <- new Point (20, 20)
button.Text <- "Click me"
// Open a message box, when button clicked
let buttonClicked (e : EventArgs) =
  let question = "Is this statement false?"
  let caption = "Window caption"
  let boxType = MessageBoxButtons.YesNo
  let response = MessageBox.Show (question, caption, boxType)
  printfn "The user pressed %A" response
button.Click.Add buttonClicked

Application.Run win
```

As alternative to `YesNo` response button, message box also offers `AbortRetryIgnore`, `OK`, `OKCancel`, `RetryCancel`, and `YesNoCancel`. Opens a new window for a simple dialogue with the user. All other windows of the process are blocked until the user closes the dialogue window

With *`System.Windows.Forms.OpenFileDialog`* you can ask the user to select and existing filename as demonstrated in Listing 23.17 and Figure 23.15. · `OpenFileDialog`

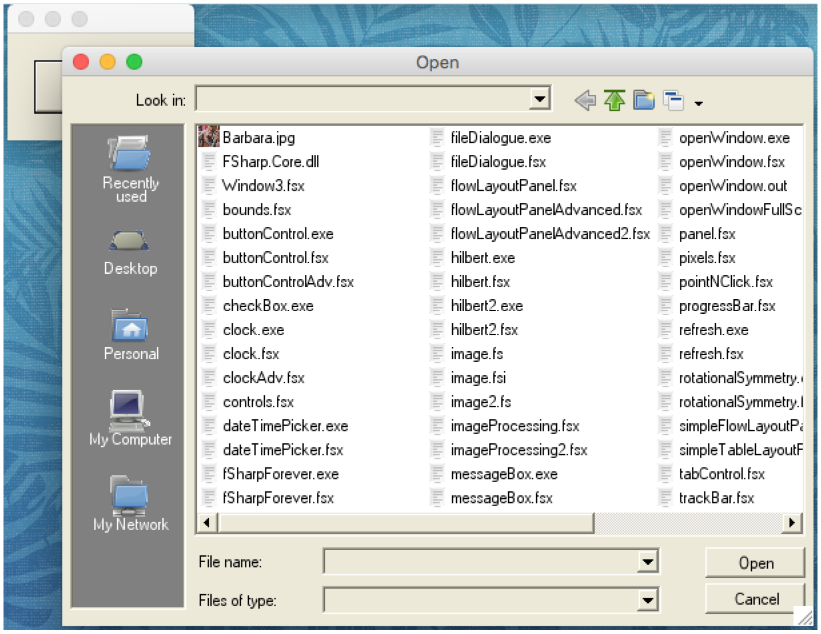


Figure 23.15: Ask the user for a filename to read from. See Listing 23.17.

Listing 23.17 winforms/openFileDialog.fsx:
Create the `OpenFileDialog`, see also Figure 23.15.

```
open System.Windows.Forms
open System.Drawing
open System

let win = new Form ()
win.ClientSize <- Size (140, 80)

let button = new Button ()
win.Controls.Add button
button.Size <- new Size (100, 40)
button.Location <- new Point (20, 20)
button.Text <- "Click me"
// Open a message box, when button clicked
let buttonClicked (e : EventArgs) =
  let opendlg = new OpenFileDialog()
  let okOrCancel = opendlg.ShowDialog()
  printfn "The user pressed %A and selected %A" okOrCancel
   opendlg.FileName
button.Click.Add buttonClicked

Application.Run win
```

Similarly to `OpenFileDialog`, *`System.Windows.Forms.SaveFileDialog`* asks for a file name, but if an existing file is selected, then the user will be asked to confirm the choice. · `SaveFileDialog`

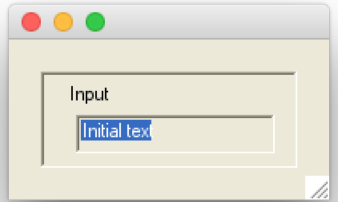


Figure 23.16: A panel including a label and a text input field, see Listing 23.18.

23.6 Organising controls

It is often useful to organise the controls in groups, and such groups are called *Panels* in WinForms. 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 23.18 and Figure 23.16.

· Panels

**Listing 23.18 winforms/panel.fsx:
Create a panel, label, text input controls.**

```
open System.Drawing
open System.Windows.Forms

let win = new Form () // Create a window form
win.ClientSize <- new Size (200, 100)

// Customize the Panel control
let panel = new Panel ()
panel.ClientSize <- new Size (160, 60)
panel.Location <- new Point (20,20)
panel.BorderStyle <- BorderStyle.Fixed3D
win.Controls.Add panel // Add panel to window

// Customize the Label and TextBox controls
let label = new Label ()
label.ClientSize <- new Size (120, 20)
label.Location <- new Point (15,5)
label.Text <- "Input"
panel.Controls.Add label // add label to panel

let textBox = new TextBox ()
textBox.ClientSize <- new Size (120, 20)
textBox.Location <- new Point (20,25)
textBox.Text <- "Initial text"
panel.Controls.Add textBox // add textbox to panel

Application.Run win // Start the event-loop
```

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.

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 23.20 and in Figure 23.17. 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.

· `FlowLayoutPanel`

Listing 23.19 winforms/flowLayoutPanel.fsx:
Create a FlowLayoutPanel, with checkbox and radiobuttons.

```
open System.Windows.Forms
open System.Drawing

let flowLayoutPanel = new FlowLayoutPanel ()
let buttonLst =
  [(new Button (), "Button0");
   (new Button (), "Button1");
   (new Button (), "Button2");
   (new Button (), "Button3")]
let panel = new Panel ()
let wrapContentsCheckBox = new CheckBox ()
let initiallyWrapped = true
let radioButtonLst =
  [(new RadioButton (), (3, 34), "TopDown", FlowDirection.TopDown);
   (new RadioButton (), (3, 62), "BottomUp", FlowDirection.BottomUp);
   (new RadioButton (), (111, 34), "LeftToRight",
   FlowDirection.LeftToRight);
   (new RadioButton (), (111, 62), "RightToLeft",
   FlowDirection.RightToLeft)]

// customize buttons
for (btn, txt) in buttonLst do
  btn.Text <- txt

// customize wrapContentsCheckBox
wrapContentsCheckBox.Location <- new Point (3, 3)
wrapContentsCheckBox.Text <- "Wrap Contents"
wrapContentsCheckBox.Checked <- initiallyWrapped
wrapContentsCheckBox.CheckedChanged.Add (fun _ ->
   flowLayoutPanel.WrapContents <- wrapContentsCheckBox.Checked)

// customize radio buttons
for (btn, loc, txt, dir) in radioButtonLst do
  btn.Location <- new Point (fst loc, snd loc)
  btn.Text <- txt
  btn.Checked <- flowLayoutPanel.FlowDirection = dir
  btn.CheckedChanged.Add (fun _ -> flowLayoutPanel.FlowDirection <- dir)
```

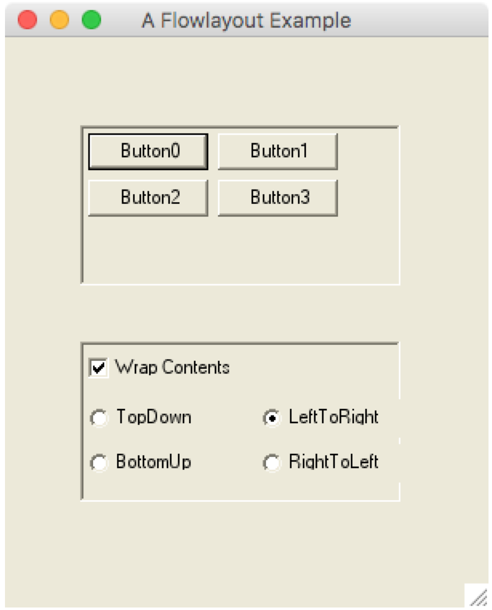


Figure 23.17: Demonstration of the `FlowLayoutPanel` panel, `CheckBox`, and `RadioButton` controls, see Listing 23.20.

Listing 23.20 winforms/flowLayoutPanel.fsx:
Create a FlowLayoutPanel, with checkbox and radiobuttons. Continued from Listing 23.19.

```
// customize flowLayoutPanel
for (btn, txt) in buttonLst do
  flowLayoutPanel.Controls.Add btn
flowLayoutPanel.Location <- new Point (47, 55)
flowLayoutPanel.BorderStyle <- BorderStyle.Fixed3D
flowLayoutPanel.WrapContents <- initiallyWrapped

// customize panel
panel.Controls.Add (wrapContentsCheckBox)
for (btn, loc, txt, dir) in radioButtonLst do
  panel.Controls.Add (btn)
panel.Location <- new Point (47, 190)
panel.BorderStyle <- BorderStyle.Fixed3D

// Create a window, add controls, and start event-loop
let win = new Form ()
win.ClientSize <- new Size (302, 356)
win.Controls.Add flowLayoutPanel
win.Controls.Add panel
win.Text <- "A Flowlayout Example"
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 `FloatLayoutPanel` 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 do it differently, and the reader is encouraged to experiment with the program.

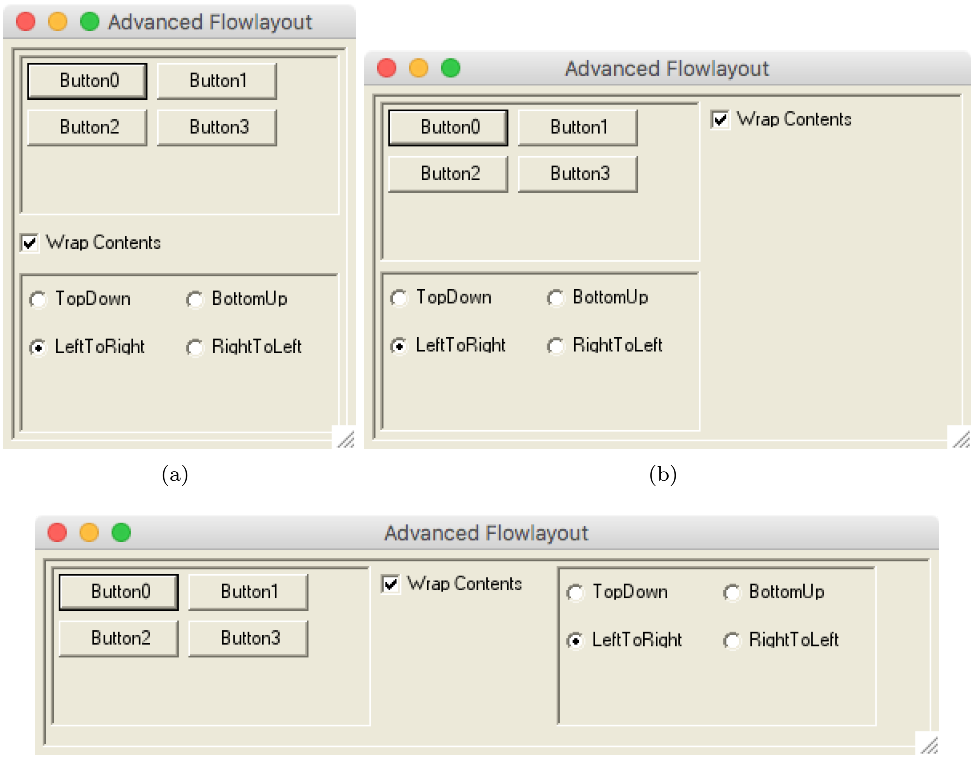
The program in Listing 23.20 has not completely separated the semantic blocks of the interface and relies on explicitly setting of coordinates of controls. This can be avoided by using nested panels. E.g., in Listing 23.22, the program has been rewritten as a nested set of `FloatLayoutPanel` in three 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 23.18.

Listing 23.21 winforms/flowLayoutPanelAdvanced.fsx: Create nested FlowLayoutPanel.

```
open System.Windows.Forms
open System.Drawing
open System

let win = new Form ()
let mainPanel = new FlowLayoutPanel ()
let mainPanelBorder = 5
let flowLayoutPanel = new FlowLayoutPanel ()
let buttonLst =
  [(new Button (), "Button0");
   (new Button (), "Button1");
   (new Button (), "Button2");
   (new Button (), "Button3")]
let wrapContentsCheckBox = new CheckBox ()
let panel = new FlowLayoutPanel ()
let initiallyWrapped = true
let radioButtonLst =
  [(new RadioButton (), "TopDown", FlowDirection.TopDown);
   (new RadioButton (), "BottomUp", FlowDirection.BottomUp);
   (new RadioButton (), "LeftToRight", FlowDirection.LeftToRight);
   (new RadioButton (), "RightToLeft", FlowDirection.RightToLeft)]

// customize buttons
for (btn, txt) in buttonLst do
  btn.Text <- txt

// customize radio buttons
for (btn, txt, dir) in radioButtonLst do
  btn.Text <- txt
  btn.Checked <- flowLayoutPanel.FlowDirection = dir
  btn.CheckedChanged.Add (fun _ -> flowLayoutPanel.FlowDirection <- dir)

// customize flowLayoutPanel
for (btn, txt) in buttonLst do
  flowLayoutPanel.Controls.Add btn
flowLayoutPanel.BorderStyle <- BorderStyle.Fixed3D
flowLayoutPanel.WrapContents <- initiallyWrapped

// customize wrapContentsCheckBox
wrapContentsCheckBox.Text <- "Wrap Contents"
wrapContentsCheckBox.Checked <- initiallyWrapped
wrapContentsCheckBox.CheckedChanged.Add (fun _ ->
   flowLayoutPanel.WrapContents <- wrapContentsCheckBox.Checked)
```

Listing 23.22 winforms/flowLayoutPanelAdvanced.fsx:
Create nested FlowLayoutPanel. Continued from Listing 23.21.

```
// customize panel
// changing border style changes ClientSize
panel.BorderStyle <- BorderStyle.Fixed3D
let width = panel.ClientSize.Width / 2 - panel.Margin.Left -
    panel.Margin.Right
for (btn, txt, dir) in radioButtonLst do
  btn.Width <- width
  panel.Controls.Add (btn)

mainPanel.Location <- new Point (mainPanelBorder, mainPanelBorder)
mainPanel.BorderStyle <- BorderStyle.Fixed3D
mainPanel.Controls.Add flowLayoutPanel
mainPanel.Controls.Add wrapContentsCheckBox
mainPanel.Controls.Add panel

// customize window, add controls, and start event-loop
win.ClientSize <- new Size (220, 256)
let windowResize _ =
  let size = win.DisplayRectangle.Size
  mainPanel.Size <- new Size (size.Width - 2 * mainPanelBorder,
    size.Height - 2 * mainPanelBorder)
windowResize ()
win.Resize.Add windowResize
win.Controls.Add mainPanel
win.Text <- "Advanced Flowlayout"
Application.Run win
```



(a) (b) (c)

Figure 23.18: Nested `FlowLayoutPanel`, see Listing 23.22, allows for dynamic arrangement of content. Content flows, when the window is resized.

24 The Event-driven programming paradigm

In *event-driven programming*, the flow of the program is determined by *events* such as the user moving the mouse, an alarm going off, a message arrives from another program, or an exception is thrown, and is very common for programs with extensive interaction with a user such as a graphical user-interface. The events are monitored by *listeners*, and the programmer can *handlers*, which are *call-back* functions to be executed when an event occurs. In event-driven programs, there is almost always a main loop, to which the program relinquishes control to when all handlers have been set up. Event-driven programs can be difficult to test since they often rely on difficult to automate mechanisms for triggering events, e.g., testing a graphical user-interface often requires users to point-and-click, which is very slow compared to automatic unit-testing.

· event-driven programming
· events
· listeners
· handlers
· call-back

25 Where to go from here

You have now learned to program in a number of important paradigms and mastered the basics of F#, so where are good places to go now? I will highlight a number of options:

Program, program, program
: You are at this stage no longer a novice programmer, so it is time for you to use your skills and create programs that solve problems. I have always found great inspiration in interacting with other domains and seek solutions by programming. And experience is a must if you want to become a good programmer, since your newly acquired skills need to settle in your mind, and you need to be exposed to new problems that require you to adapt and develop your skills.

Learn to use an Integrated Development Environment effectively
: An Integrated Development Environment (IDE) is a tool that may increase your coding efficiency. IDEs can help you get started in different environments, such as on a laptop or a phone, it can quickly give you an overview of available options when you are programming. E.g., all IDEs will show you available members for identifiers as you type reducing time to search members and reducing the risk of spelling errors. Many IDEs will also help you to quickly refactor your code, e.g., by highlighting all occurrences of a name in a scope and letting you change it once in all places.

  In this book, we have emphasized the console, compiling and running from the console is the basis of which all IDEs build, and many of the problems with using IDEs efficiently are related to understanding how it can best help you compiling and running programs.

Learn other cool features of F#
: F# is a large language with many features. Some have been presented in this book, but more advanced topics have been neglected. Examples are:

  - regular expressions: Much computations concerns processing of text. Regular expressions is a simple but powerful language for search and replace in strings. F# has built-in support for regular expressions as `System.Text.RegularExpressions`.
  - sequence `seq`: All list type data structures in F# are built on sequences. Sequences are, however, more than lists and arrays. A key feature is that sequences can effectively contain large even infinite ordered lists, which you do not necessarily need or use, i.e., they are lazy and only compute its elements as needed. Sequences are programmed using computation expressions.
  - computation expressions: Sequential expressions is an example of a computation expressions, e.g., the sequence of squares $i^2, i = 0..9$ can be written as `seq {for i in 0 .. 9 -> i * i}`
  - asynchronous computations `async`: F# has a native implementation of asynchronous computation, which means that you can very easily set up computations that run independently on others such that they do not block for each other. This is extremely convenient if you, e.g., need to process a list of homepages, where each homepage may be slow to read, such that reading them in sequence will be slow, but with asynchronous computations, then they

can easily be read in parallel with a huge speedup for the total task as a result. Asynchronous workflows rely on computation expressions.

Learn another programming language

F# is just one of a great number of programming languages, and you should not limit yourself. Languages are often designed to be used for particular tasks, and when looking to solve a problem, you would do well in selecting the language that best fits the task. C# is an example from the Mono family, which emphasizes object-oriented programming, and many of the built-in libraries in F# are written in C#. C++ and C are ancestors of C# and are very popular since they allow for great control over the computer at the expense of programming convenience. Python is a popular prototyping language which emphasizes interactive programming like `fsharpi`, and it is seeing a growing usage in web-backends and machine learning. And the list goes on. To get an idea of the wealth of languages, browse to `http://www.99-bottles-of-beer.net` which has examples of solutions to the simple problem: Write a program that types the lyrics of song "99 bottles of beer on the wall" and stops. At present many solutions in more than 1500 different languages have been submitted.