Dynamic Link Library (DLL)

- Static linking – done by a linker
  Dynamic linking - load-time or run-time
- Two types of dynamic linking:
  - load-time – a module makes explicit calls to exported DLL functions as if they were local functions (Import Library)
  - run-time – a module gets an address of a function at run-time
- Advantages of using DLLs:
  - a single copy in memory for multiple processes
  - no need to recompile whole application after modifying one function in DLL
  - possibility to replace the DLL with a newer one
  - indepedent on the programming language

DLL Creation

- Content of source files:
  - exported functions and data
  - internal functions and data
  - the entry-point function (optional)
- Exporting functions:
  - VC++: the `_declspec(dllexport)` modifier or the .def file

```
LIBRARY     "SSCD"
EXPORTS
            Run @1
            ApplyDefaultSettings @2
```

- Import Library
  - contains information the linker needs to resolve external references to functions exported from the DLL
  - the system can locate the specified DLL at run time

The Entry-Point Function

- An optional function called automatically by the system
- The definition:

```
BOOL WINAPI DllMain( HINSTANCE hinstDLL,
                     DWORD fdwReason,
                     LPVOID lpReserved )
```

  - fdwReason:
    DLL_PROCESS_ATTACH, DLL_THREAD_ATTACH,
    DLL_THREAD_DETACH, DLL_PROCESS_DETACH
- The return value
  - when FALSE for DLL_PROCESS_ATTACH, **`LoadLibrary()`** returns NULL
- The entry-point function is called during:
  - loading the DLL - **`LoadLibrary()`**, **`LoadLibraryEx()`**
  - unloading the DLL - **`FreeLibrary()`**

Dynamic Linking

- Load-Time
  - the system searches for the DLL when starts a program
  - if the system cannot locate a required DLL, it terminates the process and displays a dialog box
  - otherwise entry-point function is called
- Run-Time
  - to load a DLL: **`LoadLibrary()`**, **`LoadLibraryEx()`**
    - they return NULL when cannot find the DLL
    - otherwise entry-point function is called
  - to get an address of exported function: **`GetProcAddress()`**
  - to get information about the DLL's module: **`GetModuleHandle()`**,**`GetModuleFileName()`**
  - to free the DLL: **`FreeLibrary()`**, **`FreeLibraryAndExitThread()`**

The DLL Data

- Global variables defined in DLL source code are global to each process using the DLL
  - each process has its own instances of all global and static variables defined in the DLL
- Dynamic Memory Allocation
  - **`GlobalAlloc()`, `LocalAlloc()`, `HeapAlloc()`, `VirtualAlloc()`**
  - the memory is allocated in the virtual address space of the calling process and is accessible only for threads of this process
  - the file mapping allows to share memory among processes
- Thread Local Storage (TLS)
  - enables a DLL to allocate an index for storing and retrieving a different value for each thread of a multithreaded process

DLLs in .NET

- No more so-called 'DLL Hell'

- The assembly - reusable, versionable, and self-describing building block of a common language runtime application
  - .exe, .dll, multifile
- Two ways of linking to an assembly:
  - add a reference
  - use the **`Assembly`** class at runtime
- Strong names – unique identifiers of assemblies
- Versioning
  - two or more assemblies with the same name but different version numbers can coexist
- The Global Assembly Cache (GAC)

Consuming Unmanaged DLL Functions

- Platform invoke actions:
  1. locate the DLL containing the function
  2. load the DLL into memory
  3. locate the address of the function in memory, push its arguments onto the stack, marshal data (if required)
  4. transfer control to the unmanaged function
- To consume exported DLL functions:
  1. identify functions in DLLs
  2. create a class to hold DLL functions
  3. create prototypes in managed code
  4. call a DLL function

Identifying Functions in DLLs

- Identification of a DLL function:
  - name of the DLL
  - function name or ordinal
- Win32 API DLLs:
  - GDI32.dll – Graphics Device Interface (GDI)
  - Kernel32.dll – memory management and resource handling
  - User32.dll – windows manangement (messages, menus, etc.)
- There are ANSI and Unicode versions of functions, e.g.:
  - MessageBoxA - ANSI
  - MessageBoxW - Unicode

Example of using DLLImportAttribute

the target platform will determine the character width and string marshaling

```
using System.Runtime.InteropServices;

public class Win32 {
  [DllImport("user32.dll", CharSet=CharSet.Auto)]
  public static extern int MessageBox(int hWnd,
        String text, String caption, uint type);
}

public class HelloWorld {
  public static void Main() {
    Win32.MessageBox(0, "Hello World",
                     "Platform Invoke Sample", 0);
  }
}
```

The Clipboard

- Win32 API contains a set of functions and messages that enable applications to transfer data
  - all applications have access to the clipboard
- Clipboard formats
  - a memory object on the clipboard can be in any data format
  - each format is identified by an unsigned integer value
- The clipboard owner – the window associated with the information on the clipboard (until the clipboard is closed or another window empties it)

Copying to the Clipboard

1. Open the clipboard - `OpenClipboard()`
2. Remove clipboard's content - `EmptyClipboard()`
3. Set new data in the clipboard - `SetClipboardData()`
   - the content - `HANDLE`
   - memory must be allocated using the `GlobalAlloc()` function with the `GMEM_MOVEABLE` flag
   - never free the memory copied to the clipboard
4. Close the clipboard - `CloseClipboard()`

Pasting from the Clipboard

1. Open the clipboard - `OpenClipboard()`
2. Check if there is data in an interesting format - `IsClipboardFormatAvailable()`, `EnumClipboardFormat()`, `GetPriorityClipboardFormat()`, `CountClipboardFormats()`
3. Get the data from the clipboard - `GetClipboardData()`
4. Close the clipboard - `CloseClipboard()`

Data Formats

- Standard formats:
  - CF_BITMAP, CF_DIB, CF_TIFF - images
  - CF_ENHMETAFILE, CF_METAFILEPICT - metafiles
  - CF_WAVE, CF_RIFF - sounds
  - CF_TEXT, CF_UNICODETEXT - texts
- Registering a new format in the system:
  - `RegisterClipboardFormat()`
  - `GetClipboardFormatName()`
- Private formats (used by one application)
  - use any constant value between CF_PRIVATEFIRST and CF_PRIVATELAST

Using Clipboard in Windows Forms

- The `System.Windows.Forms.Clipboard` class contains the following static methods:
  - `ContainsData()`, `GetData()`, `SetData()`
  - `ContainsAudio()`, `GetAudioStream()`, `SetAudio()`
  - `ContainsFileDropList()`, `GetFileDropList()`, `SetFileDropList()`
  - `ContainsImage()`, `GetImage()`, `SetImage()`
  - `ContainsText()`, `GetText()`, `SetText()`

Using Clipboard in WPF

- On copy or cut, WPF puts plaintext and XAML on to the clipboard
- On paste, WPF will paste in XAML if the source of that content is from a equally trusted or more trusted site
- The **`System.Windows.Clipboard`** class contains static methods for setting, getting, and checking the clipboard's content
- Members of the **`System.Windows.DataFormats`** enumeration can be used to set the data in an appropriate format:
  - **`Text, UnicodeText, Rtf, Html, CommaSeparatedValue, Xaml`**

The Windows Registry

- A structure of the registry
  - a tree of keys and values
- Predefined keys:
  - HKEY_CLASSES_ROOT – COM and shell's classes
  - HKEY_CURRENT_CONFIG – the hardware profile
  - HKEY_CURRENT_USER – settings for the current (logged) user
  - HKEY_LOCAL_MACHINE – system and hardware information of the computer
  - HKEY_USERS – configuration for new users
- Application settings
  - HKEY_CURRENT_USER\Software\<company>\<app>\<ver>
  - HKEY_LOCAL_MACHINE\Software\<company>\<app>\<ver>

Using Registry

- Keys:
  - creating (with opening) - `RegCreateKeyEx()`
  - opening - `RegOpenKeyEx()`
  - closing - `RegCloseKey()`
  - deleting - `RegDeleteKey()`
  - enumerating subkeys - `RegEnumKeyEx()`
  - getting information about a key - `RegQueryInfoKey()`
- Values:
  - setting the value - `RegSetValueEx()`
  - getting the value - `RegQueryValueEx()`, `RegQueryMultipleValues()`
  - deleting - `RegDeleteValue()`
  - enumerating values from a key - `RegEnumValue()`

Types of Data in Registry

- The binary data
  - REG_BINARY
- Numbers
  - REG_DWORD - 32 bits
  - REG_QWORD - 64 bits
- Strings
  - REG_SZ – with ending '\0' character
  - REG_MULTI_SZ – with two ending '\0' characters
  - REG_EXPAND – with environment variables (e.g. %PATH%) **`ExpandEnvironmentStrings()`**

`Registry` Class in .NET

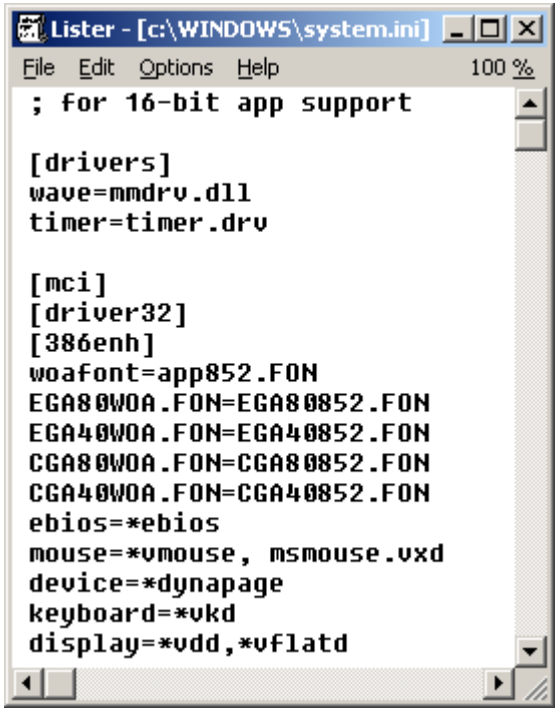
- The `Microsoft.Win32.Registry` class
  - static fields defining root keys (they return the `RegistryKey` objects):
    - `ClassesRoot` - HKEY_CLASSES_ROOT
    - `CurrentConfig` - HKEY_CURRENT_CONFIG
    - `CurrentUser` - HKEY_CURRENT_USER
    - `DynData` - HKEY_DYN_DATA
    - `LocalMachine` - HKEY_LOCAL_MACHINE
    - `PerformanceData` - HKEY_PERFORMANCE_DATA
    - `Users` - HKEY_USERS

`RegistryKey` Class in .NET

- The `RegistryKey` class methods and properties:
  - `OpenSubKey()`
  - `CreateSubKey()`
  - `SubKeyCount, GetSubKeyNames()`
  - `ValueCount, GetValueNames()`
  - `GetValue(), SetValue()`
  - `DeleteSubKey(), DeleteSubKeyTree(), DeleteValue()`

.ini Files

- Very useful text format
- Win.ini
  - GetProfileSection()
  - GetProfileInt(), GetProfileString()
  - WriteProfileSection()
  - WriteProfileString()
- <app>.ini
  - GetPrivateProfileSection()
  - GetPrivateProfileInt(), GetPrivateProfileString(), GetPrivateProfileStruct()
  - WritePrivateProfileSection()
  - WritePrivateProfileString(), WritePrivateProfileStruct()



Printing

- Printing in Windows is device independent
- The scenario of printing:
  1. Get the device context of the printer - **`PrintDlgEx()`**, **`PrintDlg()`**
  2. Check properties - **`GetDeviceCaps()`**
  3. Start printing - **`StartDoc()`**
  4. Start printing a page - **`StartPage()`**
  5. Print the page – use GDI functions for drawing
  6. At the end of printing a page - **`EndPage()`**
  7. At the end of printing - **`EndDoc()`**

Printing Interface

- GDI
- The driver of the printer
- The print spooler
  - an executable file which manages printing jobs
  - it creates printing jobs as documents with metafiles
- The print processor
  - a DLL which converts metafiles into callings of the driver's functions
  - it uses the graphics engine
- The port monitor
  - a DLL which sends data to the printer

Printing in Windows Forms

- The scenario of printing:
  1. Create a `System.Drawing.Printing.PrintDocument` object and specify the printer

```
PrintDocument pd = new PrintDocument();
pd.PrinterSettings.PrinterName = printerName;
```

  2. Set the printer (`PrinterSettings`) and page (`PageSettings`) properties
  3. Set the print-page event handler

```
pd.PrintPage += new PrintPageEventHandler(
                                    myPrintPage);
```

  4. Print the document

```
pd.Print();
```

Print Events (WinForms)

- **`BeginPrint`**
  - occurs when the **`Print()`** method is called and before the first page prints
  - it is the best place to initialize resources
  - **`PrintEventArgs: Cancel`**
- **`PrintPage`**
  - occurs before a page prints
  - **`PrintPageEventArgs: Cancel, Graphics, HasMorePages, MarginBounds, PageBounds, PageSettings`**
- **`EndPrint`**
  - occurs when the last page of the document has been printed
  - it is the best place to free resources

Using the PrintPage Event (WinForms)

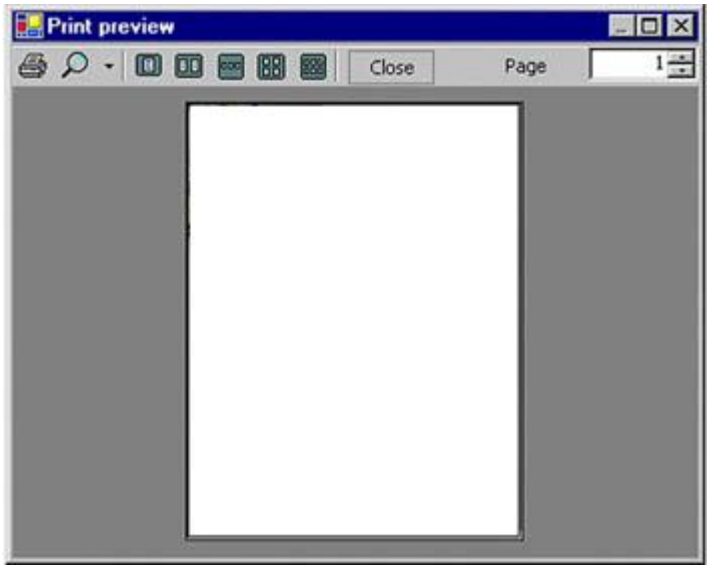
```
private void myPrintPage(object sender,
    PrintPageEventArgs ev)
{
    float linesPerPage = 0;
    float yPos = 0;
    int count = 0;
    float leftMargin = ev.MarginBounds.Left;
    float topMargin = ev.MarginBounds.Top;
    string line = null;
    linesPerPage = ev.MarginBounds.Height /
        printFont.GetHeight(ev.Graphics);

    while (count < linesPerPage &&
        ((line = streamToPrint.ReadLine()) != null))
    {
        yPos = topMargin + (count *
            printFont.GetHeight(ev.Graphics));
        ev.Graphics.DrawString(line, printFont,
            Brushes.Black, leftMargin, yPos,
            new StringFormat());
        count++;
    }
    ev.HasMorePages = (line != null);
}
```

The `PrinterSettings` Class (WinForms)

- Access to the printer's settings:
  - `PrintDocument.PrinterSettings`,
  - `PageSettings.PrinterSettings`
- Properties:
  - `InstalledPrinters` (all printers installed in the system)
  - `PrinterName`, `IsValid`
  - `PrinterResolutions`, `PaperSources`, `PaperSizes`
  - `CanDuplex`, `Duplex`, `SupportsColor`,
  - `MaximumCopies`
  - `PrintToFile`, `PrintFileName`
  - `Copies`, `FromPage`, `ToPage`, `PrintRange`
  - `DefaultPageSettings`

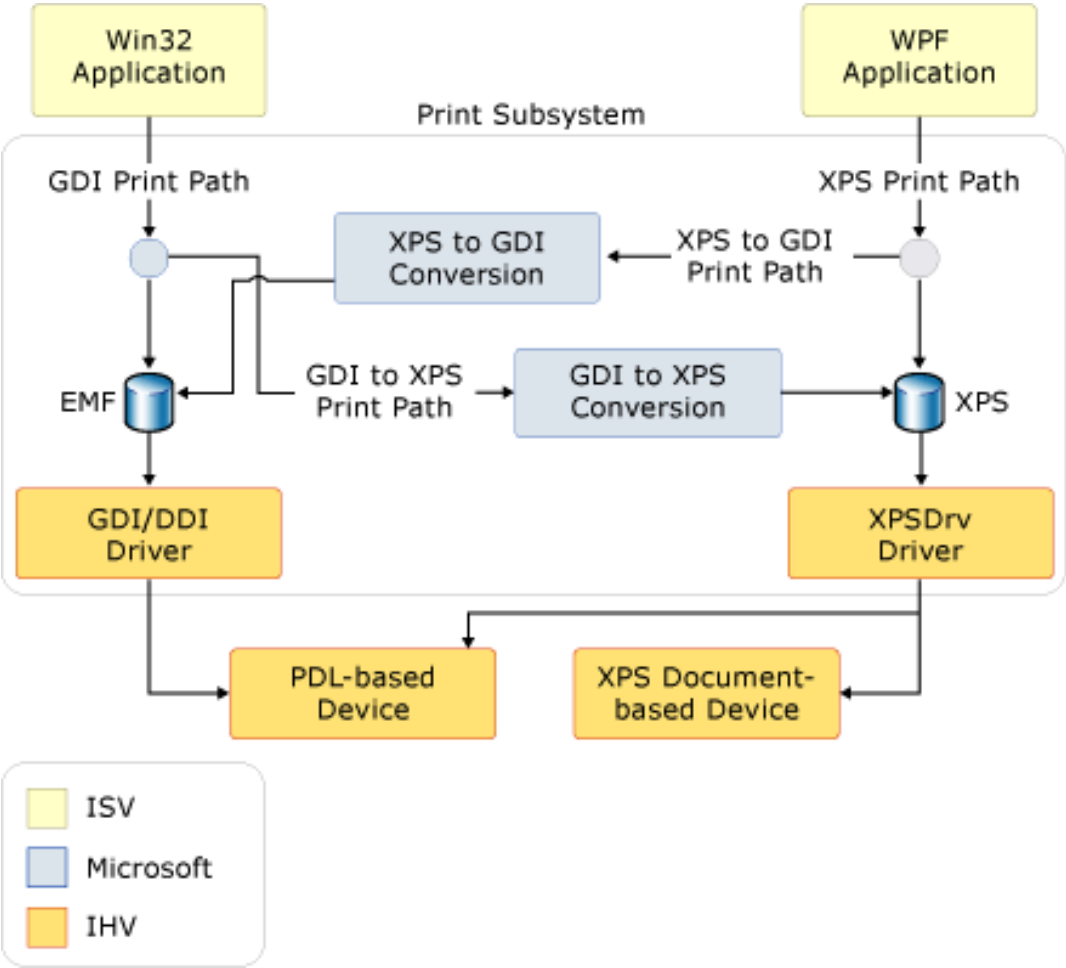
Print Dialogs (WinForms)

- `PrintDialog`
- `PageSetupDialog`
- `PrintPreviewDialog`



Windows Vista Print Management

- Windows Vista and Microsoft .NET Framework introduce a new print path — an alternative to Microsoft Windows Graphics Device Interface (GDI) printing — and a vastly expanded set of print system management APIs



XPS Print Path

- The XPS print path maintains the XPS format from application publication to the final processing in the print driver or device
  - it replaces all standard elements of printing: document presentation language (e.g. RTF), print spooler format (e.g. WMF), and page description language (e.g. Postscript or PCL)
- XPS print path allows to:
  - support of advanced colour profiles
  - improve print performance for both .NET Framework and Win32 applications
  - support XPS format
- **`System.Windows.Xps`** and **`System.Printing`** namespaces contain XPS print path API

System.Windows.Controls.PrintDialog

```
public void InvokePrint(string xpsFilePath)
{
    // Create the print dialog object and set options
    PrintDialog pDialog = new PrintDialog();
    pDialog.PageRangeSelection =
        PageRangeSelection.AllPages;
    pDialog.UserPageRangeEnabled = true;

    // Display the dialog
    if (pDialog.ShowDialog() == true) {
        XpsDocument xpsDocument = new XpsDocument(
            xpsFilePath, FileAccess.ReadWrite);
        FixedDocumentSequence fds =
            xpsDocument.GetFixedDocumentSequence();
        pDialog.PrintDocument(fds.DocumentPaginator,
            "Test print job");
    }
}
```

Programmatic Printing (System.Printing)

```
private void PrintAllXpsFiles(string path)
{
  // Create print server and print queue.
  LocalPrintServer localPrintServer = new LocalPrintServer();
  PrintQueue defaultPrintQueue =
      LocalPrintServer.GetDefaultPrintQueue();

  DirectoryInfo dir = new DirectoryInfo(path);
  foreach (FileInfo f in dir.GetFiles("*.xps")) {
    string filePath = path + "\\" + f.Name;
    try {
      // Print the Xps file while providing
      // XPS validation and progress notifications.
      PrintSystemJobInfo xpsPrintJob =
        defaultPrintQueue.AddJob(f.Name, filePath, false);
    }
    catch (PrintJobException e) {
    }
  }
}
```

Multiple Document Interface (MDI)

- There are 3 types of windows in MDI applications:
  - frame – the main window of the application
  - MDI client – background for working windows, it uses the `'MDICLIENT'` window class
  - child windows – working windows
- Features of MDI applications:
  - child windows are confined to the MDI client window and cannot appear outside it
  - the menu of the window:
    - standard menu items: Tile, Cascade, Arrange Icons, Close All, list of working windows
    - `WM_MDISETMENU`
  - accelerators
    - `TranslateMDISysAccel()`
    - ALT+-, CTRL+F4, CTRL+F6

MDI Child Windows

- There are 3 methods of creating MDI child windows:
  - calling the `CreateMDIWindow()` function
  - sending the `WM_MDICREATE` message to the MDI client
  - calling `CreateWindowEx()` with `WS_EX_MDICHILD` style
- Activation
  - `WM_MDIACTIVATE`
  - `WM_MDIGETACTIVE`
- Size
  - `WM_MDIMAXIMIZE, WM_MDIRESTORE`
- Arrangement
  - `WM_MDICASCADE, WM_MDITILE, WM_MDIICONARRANGE`
- Destroying
  - `WM_MDIDESTROY`

Creating MDI Application Windows

1. Register a class for the frame window
2. Register a class for child windows
3. Create the frame window
4. Create the MDI client window
   - **`CreateWindow()`** with **`'MDICLIENT'`** window class name
5. Call **`TranslateMDISysAccel()`** in a message loop
6. The frame window's procedure:
   - use **`DefFrameProc()`** instead of **`DefWindowProc()`**
7. Child (working) windows' procedure:
   - use **`DefMDIChildProc()`** instead of **`DefWindowProc()`**
8. Create child window(s)

MDI Application in Windows Forms

- The MDI parent form
  - a normal form with the **`IsMDIContainer`** flag set to **`true`**
- MDI child forms
  - normal forms with the **`MdiParent`** property set to the MDI parent form

```
protected void MDIChildNew_Click(
    object sender, System.EventArgs e)
{
    Form2 newMDIChild = new Form2();
    newMDIChild.MdiParent = this;
    newMDIChild.Show();
}
```

MDI Application in Windows Forms cont.

- Operations on the MDI parent form:
  - to determine the active MDI child: **`ActiveMdiChild`**
  - to arrange MDI child forms: **`LayoutMdi()`**
- Limitations for MDI child forms:
  - **`Opacity`** doesn't work
  - **`CenterToParent()`** doesn't work
- Standard **`Window`** menu for MDI applications:
  - using new menu strips [2.0]: **`MenuStrip.MdiWindowListItem`**
  - using old main menu [1.x]: **`MenuItem.MdiList`**

MDI Application in WPF

- WPF does not support MDI
- Using MDI is not recommended for modern applications, using the `TabControl` should be considered
- 3rd parties libraries provide MDI functionality for WPF
  - commercial, e.g. Actipro Docking & MDI or SandDock
  - open source, e.g. AvalonDock