Lab 01: PCL, Linking and compiler directives

Prerequisites

You will need a development environment, either a Mac or Windows PC with the iOS or Android SDK (or both, depending on which platform you prefer to use) and Xamarin tools installed.

Downloads

https://university2.xamarin.com/materials/xam300-advanced-cross-platform-development

Lab Goals

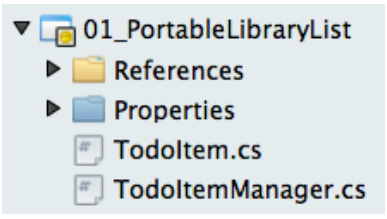
The goals of this lab will are to:

- Call from a PCL into platform-specific code using an Action.
- Call from a PCL into platform-specific code by implementing an Interface.
- Use file-linking and compiler directives instead of PCL to share code.
- Examine a complex example of file-linking and compiler directives by using the SQLite-NET library from https://github.com/praeclarum/sqlite-net/ in an app.

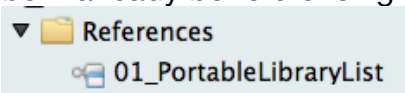
Steps

Step 1: Basic PCL with Action

1. Launch your IDE (either Xamarin Studio or Visual Studio)
2. Open the **PortableLab1.sln** solution
3. Expand the **01_PortableLibraryList** project – this is a Portable Class Library that can be shared between iOS, Android and Windows Phone.



4. Check the References of the iOS and Android projects. They should both already be referencing the PCL:



5. Run either the iOS or Android app (or both). You can add, edit and delete to-do items but they are only stored while the application is running.

> **Tip**: This sample only stores data in memory. Every time the app starts the list is empty. This demonstrates one of the limitations of PCL – without any support for filesystem access it is difficult to write any sort of persistence mechanism.

6. To allow the PCL to trigger some platform-specific functionality, locate the comment `TODO: Step1a: add Action` in **TodoItemManager.cs** and uncomment the Action declaration

```
public Action Saved {get;set;}
```

7. Then uncomment the line below `TODO: Step1b: set default` in **TodoItemManager.cs** to set a default empty Action

```
Saved = () => {};
```

8. Uncomment the line below `TODO: Step1a: add Action` in **TodoItemManager.cs** to call the Action after each call to the SaveTask method.

```
Saved ();
```

By default, the action empty action we specified above will get called, however any other code that uses this class could set the Saved Action property to perform some custom behavior in the application.

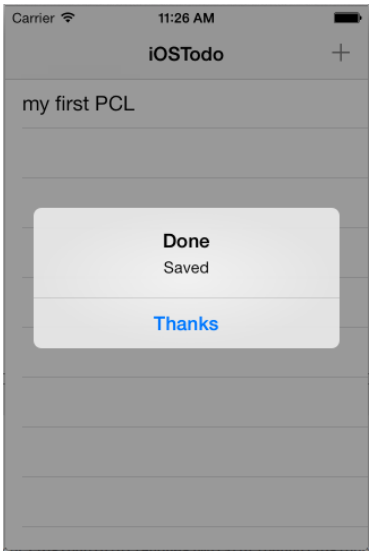
9. To actually implement some platform-specific behavior that is triggered by the PCL, uncomment a snippet of code in each platform-specific app:
For Android, `TODO: Android Step 1d: implement Action` and uncomment line right below it.

```
TaskMgr.Saved = () => {
    Android.Widget.Toast.MakeText(this, "Saved", Android.Widget.ToastLength.Short)
    .Show();
};
```

For iOS, `TODO: iOS Step 1d: implement Action` and uncomment line right below it.

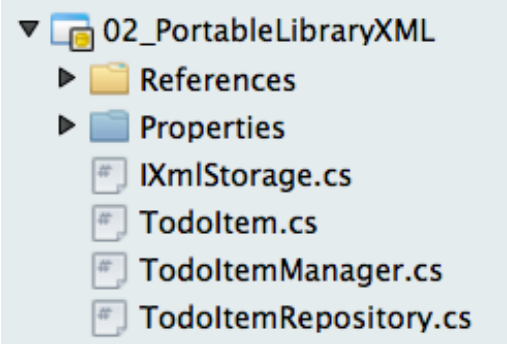
```
TaskMgr.Saved = () => {
    new UIAlertView("Done", "Saved", null, "Thanks", null).Show();
};
```

Following is an iOS screenshot where the Saved Action is called from the PCL and displays UIAlertView



Step 2: PCL with injected code

1. Expand the **02_PortableLibraryXML** project – this is a Portable Class Library that can be shared between iOS, Android and Windows Phone.



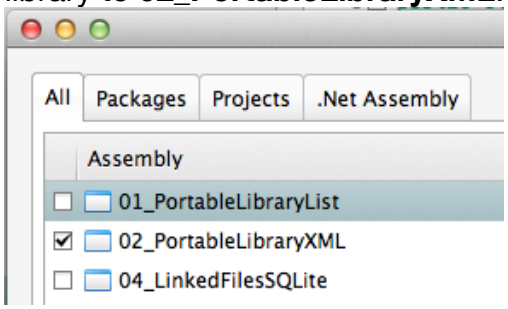
2. This code differs from Step 1 because we've created an IXmlStorage interface that contains the following code:

```
public interface IXmlStorage
{
    List<TodoItem> ReadXml (string filename);
    void WriteXml (List<TodoItem> tasks, string filename);
}
```

In the `TodoItemRepository` class we call this interface to load and save a serialized list of `TodoItem` objects. We know that PCL Profiles don't contain classes that can load and save files, so we'll implement this interface in each application project and pass it into the PCL shared code.

3. Open an application projects (iOS, Android or both). Right-click the project and choose Edit References… then change the referenced

library to **02_PortableLibraryXML**.



4. Now open the **/Implementations/XmlStorage_Implementation.cs** file. Select the entire file and uncomment the code – for iOS and Android the code looks like this:

```
public class XmlStorageImplementation : IXmlStorage
{
    public XmlStorageImplementation () {}
    public List<TodoItem> ReadXml (string filename)
    {        if (File.Exists (filename)) {
            var serializer = new XmlSerializer (typeof(List<TodoItem>));
            using (var stream = new FileStream (filename, FileMode.Open)) {
                return (List<TodoItem>)serializer.Deserialize (stream);
            }
        }
        return new List<TodoItem> ();
    }

    public void WriteXml (List<TodoItem> tasks, string filename)
    {
        var serializer = new XmlSerializer (typeof(List<TodoItem>));
        using (var writer = new StreamWriter (filename)) {
            serializer.Serialize (writer, tasks);
        }
    }
}
```

In the `TodoItemRepository` class we call this interface to load and save a serialized list of `TodoItem` objects. We know that PCL Profiles don't contain classes that can load and save files, so we'll implement this interface in each application project and pass it into the PCL shared code.

5. If you are using Visual Studio, you can also add the Windows Phone project to the solution and look at a Windows Phone-compatible implementation, shown here:

```
public class XmlStorageImplementation : IXmlStorage
{
    public XmlStorageImplementation  () {}
    public List<TodoItem> ReadXml(string filename)
    {
        IsolatedStorageFile fileStorage =
IsolatedStorageFile.GetUserStoreForApplication();
        if (fileStorage.FileExists(filename))
        {
            var serializer = new XmlSerializer(typeof(List<TodoItem>));

            using (var stream = new StreamReader(new
IsolatedStorageFileStream(filename, FileMode.Open, fileStorage)))
```

```
            {
                return (List<TodoItem>)serializer.Deserialize(stream);
            }
        }
        return new List<TodoItem> ();
    }
    public void WriteXml(List<TodoItem> tasks, string filename)
    {
        IsolatedStorageFile fileStorage =
IsolatedStorageFile.GetUserStoreForApplication();
        var serializer = new XmlSerializer(typeof(List<TodoItem>));
        using (var writer = new StreamWriter(new IsolatedStorageFileStream(filename,
FileMode.OpenOrCreate, fileStorage)))
        {
            serializer.Serialize(writer, tasks);
        }
    }
}
```

The two different implementations demonstrate how we can provide platform-specific code to enable the Portable Class Library code to load and save data even though we can write that code in the PCL.

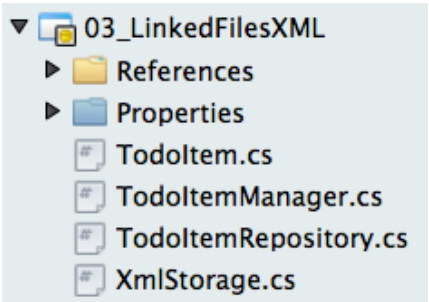
6. Find the task `//TODO: Step2: XML PCL` and uncomment the code that creates an instance of the storage implementation and passes it to the PCL class `TodoItemManager` (comment out or delete the code above from step 1).

```
var xmlFilename = "TodoList.xml";
var path = Path.Combine (libraryPath, xmlFilename);
var xmlStorage = new XmlStorageImplementation ();
TaskMgr = new TodoItemManager (path, xmlStorage);
```

Tip: This sample stores data in an XML file – on iOS and Android it uses System.IO.FileSteam and on Windows Phone it uses StreamWriter to IsolatedStorage. If you run the app now the to-do items that you create will be saved and visible when the app is restarted.

Step 3: Replace PCL with Linked Files

1. Expand the **03_LinkedFilesXML** project – this is a .NET C# Library Project – NOT a Portable Class Library. We cannot add this as a reference to any mobile application project. The project looks like this:

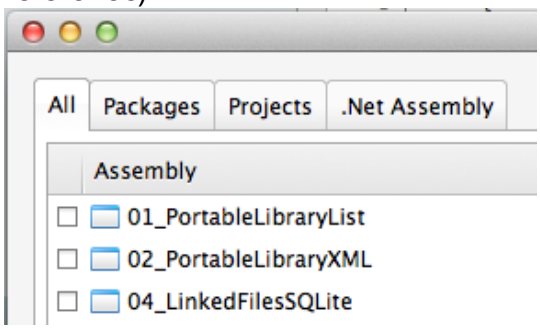


2. Instead of containing an interface like the PCL from step 2, the project contains the `XmlStorage` class which contains both different implementations, surrounded by compiler directives:

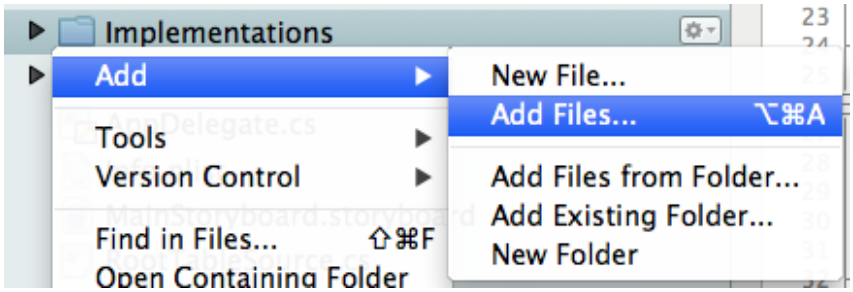
```
public XmlStorage ()
{
}
#if __IOS__ || __ANDROID__
public List<TodoItem> ReadXml (string filename)
{
    if (File.Exists (filename)) {
        var serializer = new XmlSerializer (typeof(List<TodoItem>));
        using (var stream = new FileStream (filename, FileMode.Open)) {
            return (List<TodoItem>)serializer.Deserialize (stream);
        }
    }
    return new List<TodoItem> ();
}

public void WriteXml (List<TodoItem> tasks, string filename)
{
    var serializer = new XmlSerializer (typeof(List<TodoItem>));
    using (var writer = new StreamWriter (filename)) {
        serializer.Serialize (writer, tasks);
    }
}
#elif WINDOWS_PHONE
public List<Task> ReadXml(string filename)
{
    IsolatedStorageFile fileStorage = IsolatedStorageFile.GetUserStoreForApplication();

    if (fileStorage.FileExists(filename))
    {
        var serializer = new XmlSerializer(typeof(List<Task>));

        using (var stream = new StreamReader(new IsolatedStorageFileStream(filename, Fi
leMode.Open, fileStorage)))
        {
        return (List<Task>)serializer.Deserialize(stream);
        }
    }
    return new List<Task>();
}

public void WriteXml(List<Task> tasks, string filename)
{
    IsolatedStorageFile fileStorage = IsolatedStorageFile.GetUserStoreForApplication();

    var serializer = new XmlSerializer(typeof(List<Task>));
    using (var writer = new StreamWriter(new IsolatedStorageFileStream(filename, FileMo
de.OpenOrCreate, fileStorage)))
    {
        serializer.Serialize(writer, tasks);
    }
}
#endif
}
```

The compiler directives are important – when we file-link this source code into different project types (iOS, Android or Windows Phone) then different source code will get compiled.

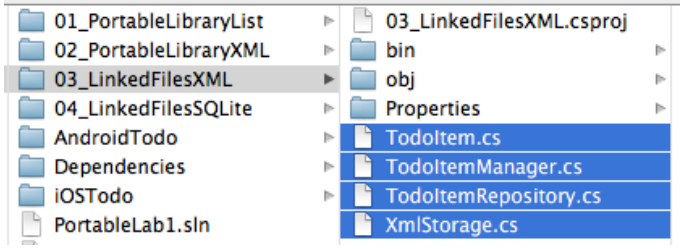
3. Open an application projects (iOS, Android or both). Right-click the project and choose **Edit References…** then remove all the project references (we are going to link the files instead of using a project reference).



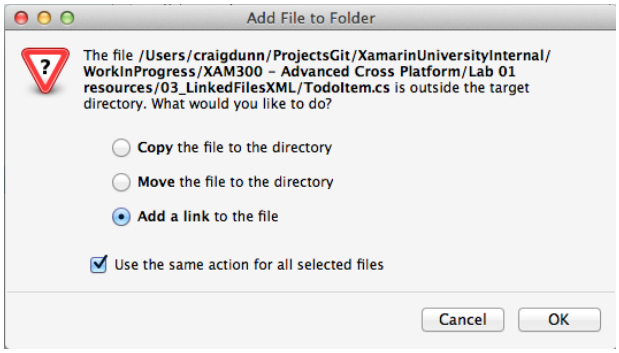
4. Right-click on the project's Implementations folder and choose **Add > Add Files…**



5. Navigate to the source code for the 03_LinkedFilesXML project in the filesystem and select them all, then press **Open**.



6. Then choose **Add a link** in the next window and press **OK**.



7. Now REPEAT this process for the other application projects. You need to keep the file links synchronized for each project – so when you are writing new code remember that any new code files (or files

that are moved, renamed or deleted) should be re-linked/deleted in each application project.

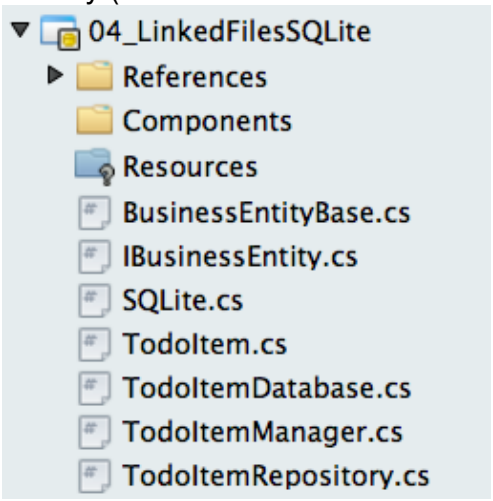
8. Find the task `//TODO: Step3: XML LINKED FILES` and uncomment the code that creates the `TodoItemManager` that is built using compiler directives for each platform (comment out or delete the code above from step 2).

```
var xmlFilename = "TodoList.xml";
var path = Path.Combine(libraryPath, xmlFilename);
TaskMgr = new TodoItemManager(path);
```

> **Tip**: This sample stores data in an XML file – on iOS and Android it uses System.IO.FileSteam and on Windows Phone it uses StreamWriter to IsolatedStorage – the same way as the demo in Step 2.
>
> The difference is that all the code to save the file is in the one file – XmlStorage.cs – and selectively compiled into each application project by linking the C# files and using compiler directives.

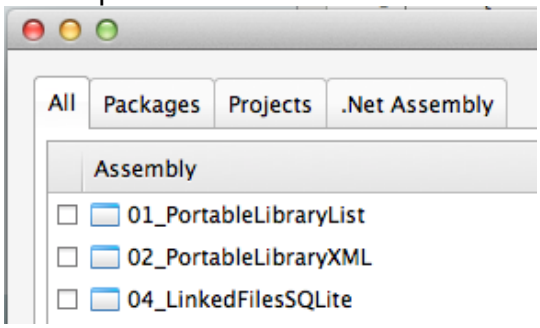
Step 4: Using Complex Linked Files

1. Expand the **04_LinkedFilesSQL** project – this is a regular .NET Library (NOT a Portable Class Library).

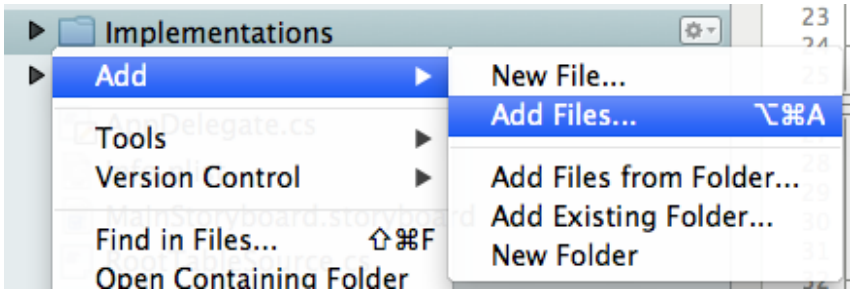


2. Open an application projects (iOS, Android or both). Right-click the project and choose **Edit References…** there should not be any references (see screenshot below) because we are going to link to

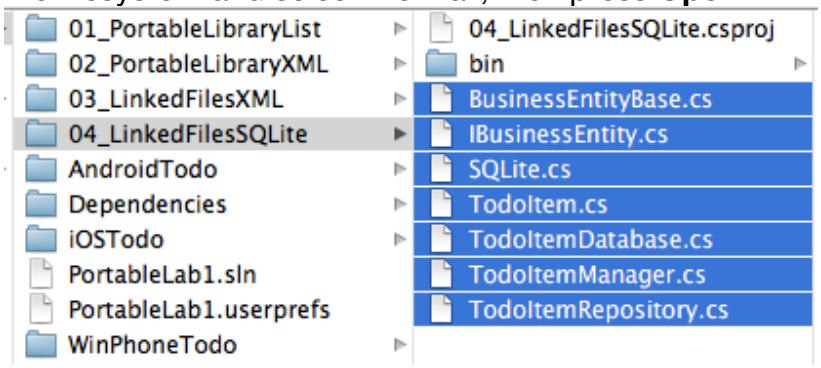
the required source code files.



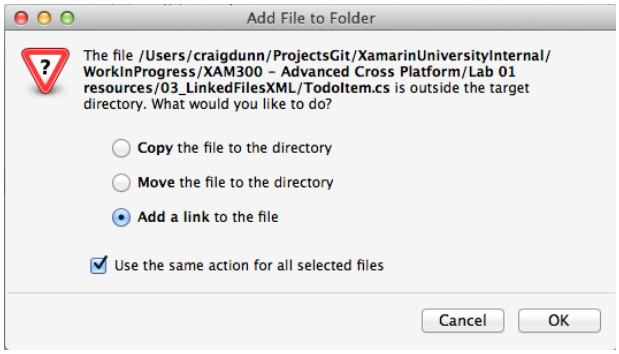
3. Right-click on the project's Implementations folder and choose **Add > Add Files…**



4. Navigate to the source code for the 04_LinkedFilesSQLite project in the filesystem and select them all, then press **Open**.



5. Then choose **Add a link** in the next window and press **OK**.



6. Now REPEAT this process for the other application projects. You need to keep the file links synchronized for each project – so when you are writing new code remember that any new code files (or files

that are moved, renamed or deleted) should be re-linked/deleted in each application project.

7. Find the task `//TODO: Step4: SQLite LINKED FILES` and uncomment the code that creates an instance of the SQLite implementation (comment out or delete the code above from step 3).

```
var path = Path.Combine(libraryPath, sqliteFilename);
var conn = new SQLite.SQLiteConnection (path);
```

You can follow this same approach to include SQLite-NET ORM functionality in your own applications.

> **Tip**: This sample stores data using the SQLite database engine.
>
> It reads and writes to the database via the SQLite-NET code that we linked into our apps. This is a lightweight Object Relational Mapping tool (ORM) that helps to store and retrieve objects without writing SQL statements ourselves.
>
> This library requires slightly different implementation on each platform, which is only enabled by the compiler directives and file linking approach.

Summary

In this lab we have examined four different sample apps. Each uses the same user-interface to present a simple to-do list, however the back-end was different each time. We used:

- A simple in-memory list via a Portable Class Library (also using an Action to trigger native code from the PCL).
- An XML file via a Portable Class Library using dependency injection.
- An XML file via code that was file-linked into each app.
- A SQLite database via code that was file-linked into each app.

These examples are designed to illustrate the pros and cons of PCLs and file-linking for structuring your cross-platform mobile application code.