CSC-200

Module 5

Implementing Console & Window Applications

1 Console Application

1.1 Interactive Application

So far we have yet to build a real-life console-based application. You can choose to build an interactive or non-interactive application. An interactive application requires that the user to be present to provide input data and view the results. You can capture input data from the user using **Console.ReadLine** method use **Console.Write** or **WriteLine** methods to present results. Add in a new project using the following information. Add SymBank.Data project to the same solution and reference it in the console application project.

Application project information

```
Project Name     :     AddAccount1
Project Type     :     Visual C# - Windows – Console Application
Location         :     C:\CSDEV\SRC
Solution         :     Module5
```

Since you are building an application, an application should always handle any runtime errors that occur. Use **try catch** to catch exceptions and process them as shown below. The custom exception text can be retrieved using a **Message** property. Call **ToString** method instead to provide all the information about the exception. You can use a **IF DEBUG** preprocessor to compile different code for Debug and Release versions.

Creating an interactive application: AddAccount1\Program.cs

```
static void Main() {
      try {
            Account item = new Account();
            Console.Write("ID:"); item.ID = int.Parse(Console.ReadLine());
            Console.Write("Name:"); item.Name = Console.ReadLine();
            Console.Write("Balance:"); item.Balance = decimal.Parse(
                  Console.ReadLine()); AccountService.Add(item);
            Console.WriteLine("Account {0} added successfully!", id);
      }     catch (Exception ex) {
#if DEBUG
            Console.WriteLine(ex.ToString());
#else
            Console.WriteLine("Could not add new account." + ex.Message);
#endif
      }
}
```

1.2 Command-Line Application

You can also build a non-interactive application which is called a command-line application. This type of application does not require the user to be present as it can be scheduled to run and results can be redirected into a file. Input data can be retrieve from a command-line arguments that is passed to **Main** method and the method can also return an **int** value called the exit code to determine if the program has succeeded or failed. If successful, the exit code is normally zero otherwise it represents an error level. You can also use a Exit method from the Environment class to terminate an application from anywhere and to return the exit code.

Application project information

```
Project Name     :     AddAccount2
Project Type     :     Visual C# - Windows – Console Application
Location         :     C:\CSDEV\SRC
Solution         :     Module5
```

A command-line application: AddAccount2\Program.cs

```
static int Main(string[] arguments) {
      try {
            if (arguments.Length != 3)
                  throw new Exception("Invalid number of arguments.");
            Account item = new Account {
                  ID = int.Parse(arguments[0]),
                  Name = arguments[1],
                  Balance = decimal.Parse(arguments[2]) };
            AccountService.Add(item);
            Console.WriteLine("Account {0} added successfully!", code);
            return 0;  // or Environment.Exit(0);
      }
      catch (Exception ex) {
            Console.WriteLine("Could not add new account." + ex.Message);
            return 1;  // or Environment.Exit(1);
      }
}
```

Adding accounts using command line or batch program: NewAccounts.cmd

```
AddAccount2 500 "ALTA ELECTRONICS" 60000
AddAccount2 600 "TORO INDUSTRIES" 55000
```

Running batch program and redirect results to another file

```
NewAccounts.cmd > NewAccounts.log
```

2 Windows Application

2.1 Application & Form

We will begin by creating a Windows application project in Visual Studio. To do this select the **File** | **New Project...** option from **File** menu and then use the following information to create the project. Then add the project that you have created in the last module **SymBank.Data** into the same solution.

New project information

```
Project Name     : SymBank
Project Type     : Visual C# | Windows | Windows Application
Location         : C:\CSDEV\SRC
Solution         : SymBank
```

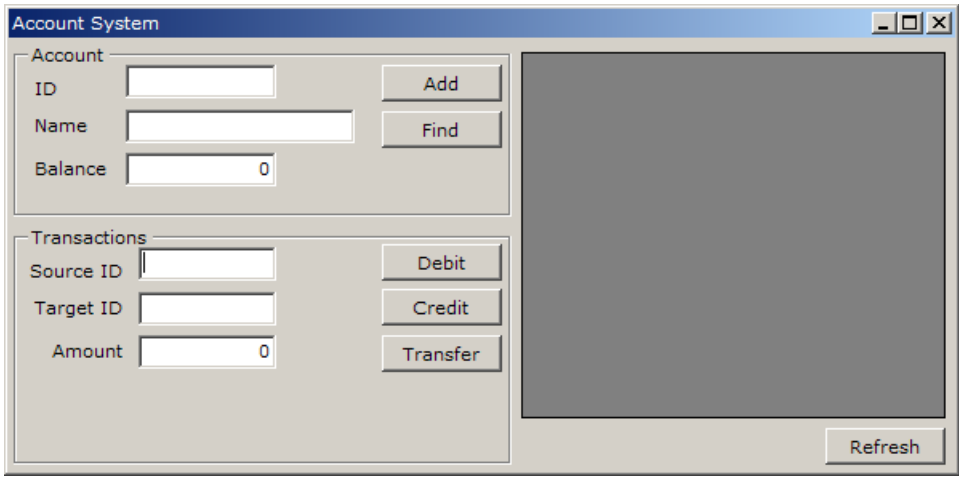
Since the class library and application is in the same solution, we need to setup which project in the solution to execute. You can do this by right-clicking on the project in the **Solution Explorer** window and select **Set as Startup Project** in popup menu. We can now add references to any external assemblies that we intend to use in our application. We will add a reference to the **SymBank.Data** class library. If the class library is in the same solution, it will be better to reference the project rather than the assembly directly. To add a reference to an external assembly, right-click over the project and then select the **Add Reference...** option from the popup menu. Use the **Projects** page for adding references to other projects in the same solution or use the **Browse** page to reference the assembly directly.

Reference to add to current project

```
SymBank.Data
```

We will now implement a form that users can utilize to add a new account to the database. There should already be a source file called **Form1** in the project when it is created. Rename the source file to **MainForm**. Open it and use the **Form Designer** window to build the user-interface in the form by adding controls from the **Toolbox** window and configuring the properties of the form or controls using the **Properties** window. The following shows the completed form with all the controls. Each control is becomes an object assigned to a field in the form class. Import the namespace for the **AccountService** class and we can then start writing all the data access code for the buttons in the form.

Visual representation of the completed form: MainForm.cs



Before we start adding controls to the form, you can setup the properties of the form first. Select the form and then use the **Properties** window to setup the properties for the form. If the window is not visible, you can press **F4** to open it. The following is the basic list of properties to configure for the form.

Form properties

```
Text            :    Account System
Font            :    Verdana, 12 pt
StartPosition   :    CenterScreen
```

2.2 Using Controls

Once the form has been setup, add the controls from the **Toolbox** window and setup their properties using the following information. Note that you should use the **Tab Order** option from **View** menu to set the **TabIndex** property rather than using the **Properties** window directly. Add two **GroupBox** controls to separate the form into two sections; one section to add and view accounts while the other to perform account transactions.

Frame controls

```
GroupBox1  Text  : Account Information          TabIndex   : 1
GroupBox2  Text  : Transaction Information      TabIndex   : 2
```

In the first group, place 3 **Label**, 3 **TextBox** controls and 2 **Button** controls as shown in the visual representation above. Then configure the controls as shown in the next page.

Label controls

```
Label1     Text  : &ID                          TabIndex   : 0
Label2     Text  : &Name                        TabIndex   : 2
Label3     Text  : &Balance                     TabIndex   : 4
```

TextBox controls

```
txtID      MaxLength  :  4                          TabIndex   : 1
txtName    MaxLength  : 30                          TabIndex   : 3
txtBalance MaxLength  : 10  TextAlign  : Right     TabIndex   : 5
```

Button controls

```
btnAdd     Text  : &Add                             TabIndex   : 6
btnFind    Text  : &Find                            TabIndex   : 7
```

In the second group, add and configure the following controls. Ensure that the shortcut key assigned using the **&** in the Text property for Labels and Buttons do not clash with the ones already assigned in the above controls.

Label controls

```
Label4     Text  :     &Source ID                   TabIndex   : 0
Label5     Text  :     &Target ID                   TabIndex   : 2
Label6     Text  :     A&mount                      TabIndex   : 4
```

TextBox controls

```
txtSourceID MaxLength :  4                          TabIndex   : 1
txtTargetID MaxLength :  4                          TabIndex   : 3
txtAmount   MaxLength : 10  TextAlign : Right       TabIndex   : 5
```

Button controls

```
btnDebit         Text  :     &Debit                 TabIndex   : 6
btnCredit        Text  :     &Credit                TabIndex   : 7
btnTransfer      Text  :     T&ransfer              TabIndex   : 8
```

Finally, you can add a **DataGridView** and an additional Button control on the right side of the form. The grid is to display the list of accounts when the button is clicked.

DataGridView control

```
grdAccounts      ReadOnly               : True      TabIndex   : 2
                 AllowUsersToAddRows    : False
                 AllowUsersToDeleteRows: False
```

Button control

```
btnRefresh       Text  : Refres&h                   TabIndex   : 3
```

If you allow the form to be resized, the controls have to be anchored or docked in order to be automatically resized to fit the form. You should also make sure to provide a minimum size so that the form cannot be too small that will cause some of the controls not to be visible. Alternatively, you can enable **AutoScroll** property so that scrollbars will automatically appear if all the controls cannot fit into the visible area of the form.

To make sure **GroupBox1** is not moved and not resized set **Anchor** to *Top* + *Left*. To ensure **GroupBox2** is not moved but is resized vertically set **Anchor** to *Top* + *Left* + *Bottom*. To ensure that the grid would not be moved but resized both horizontally and vertically, set **Anchor** to all sides. Finally to ensure that the **btnRefresh** is moved but not resized set **Anchor** to *Right* + *Bottom*. Your UI is now completed.

Anchoring for controls

```
GroupBox1        Anchor : Top + Left
GroupBox2        Anchor : Top + Left + Bottom
grdAccounts      Anchor : Top + Left + Right + Bottom
btnRefresh       Anchor : Right + Bottom
```

2.3 Providing Helper Methods

When operations succeed or fail, an application would usually inform the user of the result. You can use the **MessageBox** class for this purpose and since this is a common usage, you can implement helper methods to show messages to the user in the form.

Helper methods to show messages

```
public void Success(string message) {
      MessageBox.Show(this, message, "Information",
            MessageBoxButtons.OK, MessageBoxIcon.Information);
}

public void Failure(string message) {
      MessageBox.Show(this, message, "Error",
            MessageBoxButtons.OK, MessageBoxIcon.Error);
}
```

It is also common that after performing certain operations you might want that the form or controls to be cleared and the focus to be moved to a control in the form. Since we have two different control groups in the form, you can add the following two methods.

Methods to reset form controls

```
public void ResetAccountForm() {
      txtID.Text = string.Empty;
      txtName.Text = string.Empty;
      txtBalance.Text = "0";
      txtID.Focus();
}

public void ResetTransactionForm() {
      txtSourceID.Text = string.Empty;
      txtTargetID.Text = string.Empty;
      txtBalance.Text = "0";
      txtSourceID.Focus();
}
```

2.4 Adding Event Handlers

Each control has a set of events where you can assign delegates to call methods that will handle the events when they occur. For example, a button control has an event called **Click**. By assigning a delegate to a method in the form to this event, when user clicks on the button, the method will be called automatically by the event. Note that **EventHandler** is the standard delegate type for form and control events. Below is the format of the method that this delegate type will call.

Method signature for EventHandler delegates

```
private void ControlName_EventName(object sender, EventArgs e) {
            :
}
```

In Visual Studio, you can easily create event handler by either using the event section in the **Properties** window. You can see what events are available for the control and double-clicking on the event will automatically add the event handler method. There is also a default event for each control. For example, **Click** event is the default event for a button control. Double-clicking on the control will automatically create a handler for the default event.

We can now double-click on each button in turn and program the event handler to complete the required operation. A standard event handler will perform some standard operations such as handling errors and updating controls. Following is the list of event handlers that will perform the data access operations we need in the form. Since runtime errors may occur, we need to implement a **try-catch** block to capture any exceptions that occur and display the error message to the user by using the **Message** property of the exception. We also put finalization to re-enable controls and change control focus regardless of operation status. Note how simple it is to call the data service to complete the operation.

Example event handler method to add an account

```
private void btnAdd_Click(object sender, EventArgs e) {
      try {
            btnAdd.Enabled = false;
            Account item = new Account {
                  ID = int.Parse(txtID.Text),
                  Name = txtName.Text,
                  Balance = decimal.Parse(txtBalance.Text)
            };
            AccountService.Add(item);
            Success("Account has been added.");
            ResetAccountForm();
      }
      catch (Exception ex) {
            Failure("Cannot add account. " + ex.Message);
      }
      finally {
            btnAdd.Enabled = true;
      }
}
```

Event handler for Find button

```
private void btnFind_Click(object sender, EventArgs e) {
      try   {
            btnFind.Enabled = false;
            Account item = AccountService.GetItem(int.Parse(txtID.Text));
            txtName.Text = item.Name;
            txtBalance.Text = item.Balance.ToString();
            txtID.Focus();
      }
      catch (Exception ex) {
            Failure("Cannot find account. " + ex.Message);
            ResetAccountForm();
      }
      finally {
            btnFind.Enabled = true;
      }
}
```

Method to debit, credit and transfer

```
private void btnDebit_Click(object sender, EventArgs e) {
      try   {
            btnDebit.Enabled = false;
            AccountService.Debit(int.Parse(txtSourceID.Text),
                  decimal.Parse(txtAmount.Text));
            Success("Account has been debited.");
            ResetTransactionForm();
      }
      catch (Exception ex) {
            Failure("Cannot debit account. " + ex.Message);
      }
      finally { btnDebit.Enabled = true; }
}

private void btnCredit_Click(object sender, EventArgs e) {
      try {
            btnCredit.Enabled = false;
            AccountService.Credit(int.Parse(txtSourceID.Text),
                  decimal.Parse(txtAmount.Text));
            Success("Account has been credited.");
            ResetTransactionForm();
      }
      catch (Exception ex) {
            Failure("Cannot credit account. " + ex.Message);
      }
      finally { btnCredit.Enabled = true; }
}

private void btnTransfer_Click(object sender, EventArgs e) {
      try {
            btnTransfer.Enabled = false;
            AccountService.Transfer(
                  int.Parse(txtSourceID.Text),
                  int.Parse(txtTargetID.Text),
                  decimal.Parse(txtAmount.Text));
            Success("Amount has been transferred.");
            ResetTransactionForm();
      }
      catch (Exception ex) {
            Failure("Cannot transfer amount. " + ex.Message);
      }
      finally { btnTransfer.Enabled = true; }
}
```

Method to display a list of accounts

```
private void btnRefresh_Click(object sender, EventArgs e) {
      try   {
            btnRefresh.Enabled = false;
            var list = AccountService.GetList();
            grdAccounts.DataSource = list;
      }
      catch (Exception ex) {
            Failure("Cannot retrieve accounts. " + ex.Message);

      }
      finally { btnRefresh.Enabled = true; }
}
```

To test the application, ensure that you are compiling to a **Debug** version in Visual Studio. You can right-click on a line in the editor to insert breakpoints. Then start the application with debugging. The program will enter break mode when any of the breakpoints are reached. You can then use the **Debug** toolbar and the windows to step through code and check the result of each operation.

2.5 Extension Methods

Currently we are using **Parse** methods to convert the input text in controls into value types. Problem is that if the text cannot be converted, it will automatically generate a runtime error. We will not be able to provide a custom message nor be able to keep the focus on the control. You can use **TryParse** method instead as it only returns false and does not generate a runtime error. Since converting text in controls is a common practice, you can create helper methods to do this and you can even compile them in a library so that you can use them again for future applications.

Helper methods to convert text in controls: UIExtensions.cs

```
public static class UIExtensions {
      public static int GetInt32(TextBox control, string error) {
            int value = 0;
            if (int.TryParse(control.Text, out value)) return value;
            control.Focus(); throw new Exception(error);
      }
      public static decimal GetDecimal(TextBox control, string error) {
            decimal value = 0m;
            if (decimal.TryParse(control.Text, out value)) return value;
            control.Focus(); throw new Exception(error);
      }
}
```

Example of using the above methods

```
Account item = new Account() {
      ID = UIExtensions.GetInt32(txtID, "ID must be an integer."),
      Balance = UIExtensions.GetDecimal(txtBalance,
            "Opening balance must be a value."),
      Name = txtName.Text
};
```

Even though it works perfectly, calling static methods is a bit cumbersome than calling a method already provided by an object. However there is a way where we can make static methods work as though they are methods of the object we are accessing. This feature is called extension methods. All you need to do is to add **this** keyword to the first parameter of the static methods. The only rule is that the type containing the methods must also be **static**.

C# will now treat the methods as though they are part of the object passed in as the parameter. The difference is that the static type name will be replaced by the first parameter. This is fully supported by intellisense. However extension methods are only visible if the type namespace is pre-imported.

Making static methods into extension methods

```
public static int GetInt32(this TextBox control, string error) {
                  :
}
public static decimal GetDecimal(this TextBox control, string error) {
                  :
}
```

Calling extension methods

```
Account item = new Account() {
      ID = txtID.GetInt32("ID must be an integer."),
      Balance = txtBalance.GetDecimal("Opening balance must be a value."),
      Name = txtName.Text
};
```

2.6 Asynchronous Methods

Certain operations may take a long time to complete. Calling these methods in GUI application will block the UI from functioning until the operation completes. You can implement asynchronous methods that can run without blocking caller's thread. An asynchronous method must always return **Task** or **Task<T>** object depending on whether the operation is void or has a return value. A delegate is passed to the constructor to run code as a separate task. The task must then be started using a **Start** method and return from the asynchronous method. In the next page we implement async versions of **Add** and **GetList** methods.

Async version of methods: SymBank.Data\AccountService.cs

```
public static Task AddAsync(Account item) {
      var task = new Task(() => {
            var dc = new SymBankDataContext();
            dc.Accounts.InsertOnSubmit(item);
            dc.SubmitChanges();
      });
      task.Start(); return task;
}

public static Task<List<Account>> GetListAsync() {
      var task = new Task<List<Account>>(() => {
            var dc = new SymBankDataContext();
            return dc.Accounts.ToList();
      });
      task.Start(); return task;
}
```

Even though calling the methods will not block the UI thread, you may still need to wait for it to complete in the event handler in order to obtain the result. This is not a problem as the event handler can also run asynchronously by marking it with the **async** keyword and then use **await** to wait for method to complete and also receive the results.

Calling async methods from the application

```
private async void btnRefresh_Click(object sender, EventArgs e) {
      try   {
            btnRefresh.Enabled = false;
            var list = await AccountService.GetListAsync();
            grdAccounts.DataSource = list;
      }
      catch (Exception ex) {
            Failure("Cannot retrieve accounts. " + ex.Message);

      }
      finally { btnRefresh.Enabled = true; }
}
```

2.7 Signing an Assembly

You can optionally create and add a key file to your project which will then can be used to sign the compiled assembly. Once an assembly is signed the CLR will be able to detect if someone has tampered with the compiled assembly and will refuse to load the assembly. This is one option that we can use to stop hackers from hacking our compiled code. If even one byte in the code is changed the .NET Runtime will detect it and will not run the assembly.

Other advantages in signing your assemblies is that your assemblies will gain a unique public key token to be used to uniquely identity the assembly. Once you reference a signed assembly, the .NET Runtime will only load that assembly. It would not accidentally load in the wrong assembly only because they have the same name so a hacker cannot replace your assembly with their own as the application will not use a hacker's assembly because it does not have the same identity as the one that is referenced during compilation.

Assembly identity

```
AssemblyIdentity = Name + Version + Culture + PublicKeyToken
```

Example identity

```
SymBank.Data,Version=1.0.0.0,Culture=Neutral,
      PublicKeyToken=3434fdca129dce0
```

Use assembly attributes to assign application and version information. Examine the **AssemblyInfo.cs** file in **Properties** folder in the project, there is already a set of assembly attributes added that are used to attach version information to the assembly.

Example assembly attributes: Properties\AssemblyInfo.cs

```
[assembly: AssemblyTitle("SymBank Application")]
[assembly: AssemblyDescription("Sample Windows Application")]
[assembly: AssemblyCopyright("Copyright © 2018 Symbolicon Systems")]
[assembly: AssemblyProduct("Symbolicon Training Samples")]
[assembly: AssemblyCompany("Symbolicon Systems")]
[assembly: AssemblyVersion("1.0.0.0")]
[assembly: AssemblyFileVersion("1.0.0.0")]
```

To create the key file and sign your assembly, use the *Signing* property page of the application project. The only setback is that signed assemblies can only use assemblies that are also signed. It means if you sign an application assembly the application cannot reference any assembly that is unsigned. However this is required to ensure that the entire application is protected from using assemblies that have been hacked. Basically all assemblies should be signed unless there is a reason not to do so. You do not have to create a new key file per project, you can copy and use the same key file across all projects. However to protect the key file, you should always assign a password. If anyone manages to obtain the key file, they cannot use it to sign their projects without the password.

Only signed assemblies can be shared assemblies. Shared assemblies can also be placed into the GAC (Global Assembly Cache) so you don't need to deploy a copy of the assembly with every application. All applications can get it from the GAC. If you need to update, you only need to install the new version in the GAC and all applications will use the new version. Note that assembly version is part of the identify so do not change the version for backward compatibility. You can just update file version instead.

Installing/updating and removing assembly into GAC

```
gacutil /i SymBank.Data.dll
gacutil /u SymBank.Data
```