Object Oriented Software Development

10. Persistent Storage

Persistent storage

- Objects in memory are destroyed when a program is shut down or if it terminates unexpectedly
- Information is lost and is not 'remembered' the next time the program runs
- It is often necessary for information to be **persistent**
  - Stored permanently even after program terminates
  - Can be retrieved by the program when needed

Types of persistent storage

- Files
  - Text files
  - XML files
  - Binary files
- Databases
  - Usually relational database although there are others, e.g. object database, "NoSQL" database
  - Database can be local (on same machine as program) or on a remote server accessed across a network

Working with files and folders in C#

- **System.IO** namespace provides classes which can be used for working with files and folders (directories)
- **FileInfo**
  - Provides properties and instance methods for the creation, copying, deletion, moving, and opening of files
- **DirectoryInfo**
  - Provides instance methods for creating, moving, and listing directories and subdirectories

Example code

- **DirectoriesAndFilesDemo**
- **Program.cs**

Listing directory contents

@ symbol before string allows \ in string – otherwise \ is interpreted as part of an escape sequence (e.g. \t means a TAB character)

```
// list directories in C drive
DirectoryInfo[] di = new DirectoryInfo(@"c:\").GetDirectories();
foreach (DirectoryInfo dir in di)
{
    Console.WriteLine(dir.Name);
}
```

Copying files

```
// make backup copy of current directory
string currentFolderName = Directory.GetCurrentDirectory();
DirectoryInfo currentFolder = new DirectoryInfo(currentFolderName);

string backupFolderName = currentFolderName + @"\backup";
DirectoryInfo backupFolder = new DirectoryInfo(backupFolderName);
if (!backupFolder.Exists)
{
    backupFolder.Create();
}

foreach (FileInfo entry in currentFolder.GetFiles())
{
    entry.CopyTo(backupFolder.FullName + @"\" + entry.Name);
}
```


Reading from and writing to files

- **FileStream**
- Allows programs to access files and read and write binary data
- **StreamReader**
  - Allow programs to read text from a FileStream
- **StreamWriter**
  - Allows programs to write text to a FileStream
- Reader/Writer “wrap around” a FileStream (or other type of stream, e.g. NetworkStream)

Opening a FileStream

```
FileStream fs = new FileStream(filename, FileMode.Append, FileAccess.Write);
```

- **FileMode**
  - How the file is opened
  - Create, CreateNew, Open, OpenOrCreate, Append
- **FileAccess**
  - How the file is accessed
  - Read, Write, ReadWrite
- These are enums (remember them?)

Example code

- **StorageDemo**
- **FileTimeSheet.cs**

FileTimeSheet

- Complete code for **FileTimeSheet** class we saw in Employees example project

```
public void AddEntry(int employeeId, int hours)
{
    // store information in file
    FileStream fs = new FileStream(filename,
        FileMode.Append, FileAccess.Write);

    using (StreamWriter sw = new StreamWriter(fs))
    {
        sw.WriteLine("{0}:{1},{2}", DateTime.Now.ToShortDateString(),
            employeeId, hours);
    }

    Console.WriteLine("Entry stored to file");
}
```

using block means we don't have to call **Close** method when finished with StreamWriter

storing **employeeId** rather than name as it uniquely identifies an employee

Working with databases in C#

- ADO.NET
  - Set of .NET classes for accessing relational databases
  - Program sends SQL queries to database and receives results
- Entity Framework
  - Object-oriented approach to working with data
  - Queries are written in C# using LINQ
  - SQL generated "behind the scenes"
  - ADO.NET is underlying technology

ADO.NET

- Classes for working directly with a database:
  - Connecting to a database
  - Sending a query to a database
  - Reading the results of a query
- Classes for working with data in memory while disconnected from the database
  - DataSet, DataTable, etc
  - Can re-connect and update database when ready

ADO.NET connected classes

- **Connection**
  - Represents a connection to a database
  - Location of the database (local file, local server, network server) defined in **connection string**
- **Command**
  - Allows a command (query) to be sent to database using a Connection
  - Command can be defined as SQL string
- **DataReader**
  - Allows program to step through query results

Connected and disconnected database access

- We will focus on connected classes here
- Similar approach to other languages, e.g. JDBC for database access in Java
- DataSets, etc. are very powerful and useful, but are specific to .NET
- Suit a data-centric rather than object-oriented approach to designing applications

Working with different databases

- There are many different relational database systems
- Use same programming model (Connection, Command, DataReader) for any database
- Actual classes used depend on the specific database

Working with different databases

- SqlConnection (etc)
  - Microsoft SQL Server databases
- OleDbConnection
  - A range of database including Access, Oracle, MySql
- OracleConnection
  - Oracle databases
- SqlCeConnection
  - Microsoft SQL Server Compact Edition databases

SQL Server

- Microsoft's enterprise database
- More scalable and robust than Access
- Designed to be run on a server, usually accessed over a network
- Compact Edition
  - A lightweight version of SQL Server intended to be used on mobile devices, etc
  - Database is a local file (.SDF file)
  - Also convenient for prototyping and example code

Example code

- **QueryDemo**
- **Program.cs**

The database – payroll.SDF

Departments

departmentname
Research
Finance
HumanResources
Production
Marketing
NULL

Employees

employeeid	employeename	username	phonenumber	location	department
1	Michael	michael	1234	loc1	Research
2	Susan	susan	2345	loc1	Marketing
3	Kyle	kyle	3456	loc1	Marketing
4	Calvin	calvin	4567	loc1	Finance
5	Gary	gary	5678	loc1	Research
6	Samantha	samantha	6789	loc1	Production
7	Sara	sara	9876	loc1	Production
8	Stacy	stacy	8765	loc1	Research
9	Leon	leon	7654	loc1	Production
10	Brandon	brandon	6543	loc1	Finance
11	Ahmad	ahmad	5432	loc1	Research
12	Bob	bob	4321	loc1	Finance
NULL	NULL	NULL	NULL	NULL	NULL

Querying a database

- Open connection

```
using (SqlCeConnection conn = new SqlCeConnection(@"Data Source=payroll.sdf"))
{
```

- Define query

```
string selectQuery;

selectQuery =
    @"SELECT employeename, username, department FROM Employees";
```

- Query can contain parameters – need to build up query string

```
selectQuery =
    @"SELECT employeename, username, department FROM Employees";
selectQuery += " WHERE department ='";
selectQuery += args[0];
selectQuery += "'";
```

Querying a database

- Create command and execute query

```
// create command object
SqlCeCommand command = new SqlCeCommand(selectQuery, conn);
command.Connection.Open();

// execute select query
SqlCeDataReader dr = command.ExecuteReader();

// Call Read before accessing data.
while (dr.Read())
{
    Console.WriteLine(String.Format("{0}, {1} , {2}",
        dr[0], dr[1], dr[2]));
}

dr.Close();
```


Example code

- **StorageDemo**
- **DatabaseTimeSheet.cs**

DatabaseTimeSheet

```
public void AddEntry(int employeeId, int hours)
{
    // create connection to database
    using (SqlCeConnection conn = new SqlCeConnection(@"Data Source=payroll.sdf"))
    {
        // build query string
        string insertQuery =
            @"INSERT INTO TimesheetEntries(entrydate,employeeid, hours)";
        insertQuery += "VALUES ('";
        insertQuery += DateTime.Now.ToShortDateString() + "', ";
        insertQuery += employeeId + ", ";
        insertQuery += hours + ")";

        // create command object
        SqlCeCommand command = new SqlCeCommand(insertQuery, conn);
        command.Connection.Open();

        // execute insert query
        int rowsAffected = command.ExecuteNonQuery();

        Console.WriteLine("rows affected: {0}", rowsAffected);
    }
}
```

Databinding

- You saw previously that you can bind a WPF window to an object and bind its controls to the objects properties
- Can also bind to data retrieved from a database
- One approach is to retrieve data and use it to construct an object, then bind window to that object

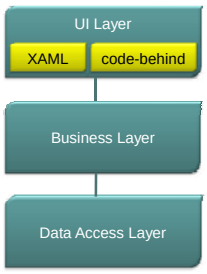
Example code

- **DatabindingDemo**
- **SingleEmployee.xaml** and **.xaml.cs**
- **Employee.cs**
- **PayrollDb.cs**

Data access classes

- It is good practice to keep code which accesses database separate from UI code in XAML code-behind class
- Separate class **PayrollDb** which has methods which store and retrieve objects
- PayrollDB in sample code could be placed in a separate **layer** of classes for data access

Layered application



User interface classes use data access classes to retrieve or store business objects

Business (model) classes: **Employee.cs**

Data access classes: **PayrollDb.cs**

GetEmployee method

- Define query on Employees table

```
public static Employee GetEmployee(int ID)
{
    Employee emp = null;

    // create connection to database
    using (SqlCeConnection conn = new SqlCeConnection(@"Data Source=payroll.sdf"))
    {
        string selectQuery;

        selectQuery =
        @"SELECT employeeid, employeename, username, phonenumber FROM Employees";
        selectQuery += " WHERE employeeid =";
        selectQuery += ID;
```


GetEmployee method in PayrollDb

- After executing command, use result to construct Employee object and return that object

```
    // execute select query
    SqlCeDataReader dr = command.ExecuteReader();

    // Call Read before accessing data.
    if (dr.Read())
    {
        emp = new Employee(dr.GetInt32(0), dr.GetString(1),
            dr.GetString(2), null, dr.GetString(3));
    }

    dr.Close();
}
return emp;
```

don't set Location property

Data bound window

- Controls are bound to Employee properties


```
<TextBlock Margin="7">Name:</TextBlock>
<TextBox Margin="5" Grid.Column="1"
         Text="{Binding Path=Name}"></TextBox>
<TextBlock Margin="7" Grid.Row="1">Username:</TextBlock>
<TextBox Margin="5" Grid.Row="1" Grid.Column="1"
         Text="{Binding Path=Username}"></TextBox>
<TextBlock Margin="7" Grid.Row="2">Phone number:</TextBlock>
<TextBox Margin="5" Grid.Row="2" Grid.Column="1"
         Text="{Binding Path=PhoneNumber}"></TextBox>
```

default binding mode here is TwoWay

Data bound window

- Set data context in button click event handler

```
private void cmdGetEmployee_Click(object sender, RoutedEventArgs e)
{
    int ID;
    if (Int32.TryParse(txtID.Text, out ID))
    {
        try
        {
            employee = PayrollDb.GetEmployee(ID);
            gridEmployeeDetails.DataContext = employee;
        }
        catch(Exception ex)
        {
            MessageBox.Show("Error contacting database." + ex.Message);
        }
    }
    else
    {
        MessageBox.Show("Invalid ID.");
    }
}
```

DataContext for part of window (Grid), not entire window

handle exceptions in UI so that they can be reported to user

Writing to the database

- Text boxes are bound to Employee (instance variable of window)
- Binding is two-way, so editing text box changes object properties
- Can save updated object to database using **UpdateEmployee** method of PayrollDb

UpdateEmployee method in PayrollDb

- Update query is executed as “non-query”

```
string updateQuery;

updateQuery =
    @"UPDATE Employees SET ";
updateQuery += "employeename ='";
updateQuery += emp.Name + "',";
updateQuery += "username ='";
updateQuery += emp.Username + "',";
updateQuery += "phonenumber ='";
updateQuery += emp.PhoneNumber + "'";
updateQuery += " WHERE employeeid =";
updateQuery += ID;

// create command object
SqlCeCommand command = new SqlCeCommand(updateQuery, conn);
command.Connection.Open();

// execute select query
rowsAffected = command.ExecuteNonQuery();
```

returns integer instead of query result

Updating

- Call update in button click event handler

```
private void cmdUpdateEmployee_Click(object sender, RoutedEventArgs e)
{
    int ID;
    if (Int32.TryParse(txtID.Text, out ID))
    {
        try
        {
            int rows = PayrollDb.UpdateEmployee(ID, employee);
            MessageBox.Show(String.Format("{0} row(s) updated",rows.ToString()));
        }
        catch (Exception ex)
        {
            MessageBox.Show("Error contacting database." + ex.Message);
        }
    }
    else
    {
        MessageBox.Show("Invalid ID.");
    }
}
```


Master-detail window

- Get collection of Employee objects from database using **GetEmployees** method of PayrollDb
- Set itemsSource property of list box to collection
- Bind grid to selected item of list box

Example code

- **DatabindingDemo**
- **EmployeesList.xaml** and **.xaml.cs**
- **Employee.cs**
- **PayrollDb.cs**

GetEmployees method in PayrollDb

```
public static ICollection<Employee> GetEmployees()
{
    ICollection<Employee> emps = new List<Employee>();
```

- Use DataReader to loop through results and construct collection

```
    // execute select query
    SqlCeDataReader dr = command.ExecuteReader();
    // Call Read before accessing data.
    while (dr.Read())
    {
        Employee emp = new Employee(dr.GetInt32(0), dr.GetString(1),
            dr.GetString(2), null, dr.GetString(3));
        emps.Add(emp);
    }

    dr.Close();
}
return emps;
```


Data bound list box

- Set itemsSource in button click event handler

```
private void cmdGetEmployees_Click(object sender, RoutedEventArgs e)
{
    employees = PayrollDb.GetEmployees();
    lstEmployees.ItemsSource = employees;
}
```

- Bind form (grid nested inside main grid) to selected item of list box

```
<Grid  Name="gridEmployeeDetails"
       DataContext="{Binding ElementName=lstEmployees, Path=SelectedItem}">
```