C# for Beginners

Sasha and Dina Goldshtein, she codes; Academy

Introduction

Welcome to the *she codes;* learning path for C#. In this learning path we are going to follow the Microsoft Virtual Academy course "C# Fundamentals for Absolute Beginners" by Bob Tabor. The online course will provide the learning materials and basic assessments and we will guide you through with more resources and suggested exercises. The course is intended for beginners with no programming background.

The estimated planned duration is 21 3-hour long *she codes;* hack nights. The syllabus is divided into 8 modules. For each module we specified the estimated time for completion. But this is just an estimation. Don't be discouraged if it takes longer. We encourage you to follow along the videos and repeat what you see in the lecture.

Solutions for the exercises are available on the course GitHub repository. Don't forget to bring your laptop, headphones and an appetite :-)

C# is a very popular programming language that is used on desktops, servers, mobile devices, and web applications. It was first released by Microsoft in 2001. In the last few years, Microsoft and some other companies partnered to bring the C# programming language to non-Microsoft platforms. Today, you can write C# applications for iOS, Android, OS X (Mac desktop), Linux, embedded devices, microcontrollers, and many other platforms. You can even use C# to write

cross-platform mobile applications, where the same code will work on iOS, Android, and Windows Phone.

To follow this course it would be best to have a computer with the Windows operating system. We recommend you work with Windows 7 or 8, which is what the instructor uses in the video course. However, it is definitely possible to follow this course with a non-Windows operating system as well.

We might change a bit the order of watching the videos. It's OK. Don't worry, we know what we're doing :-)

Each module starts with a "Time Breakdown" section. This is a short description of how the material of the module is going to be distributed between weekly sessions, and the main goals of each session. A detailed description then follows.

Let's start!

Module 1 - Introduction and First Program

Time Breakdown

Night #	Objectives
1	Watch all videos from the below Videos and Assessments section. Install Visual Studio and write first program

Goals

In this module you will learn a general introduction to programming and will set up your C# working environment. You won't get to leave before you write your first C# program! :-)

Videos and Assessments

All referrals to videos refer to the MVA course "C# Fundamentals for Absolute Beginners".

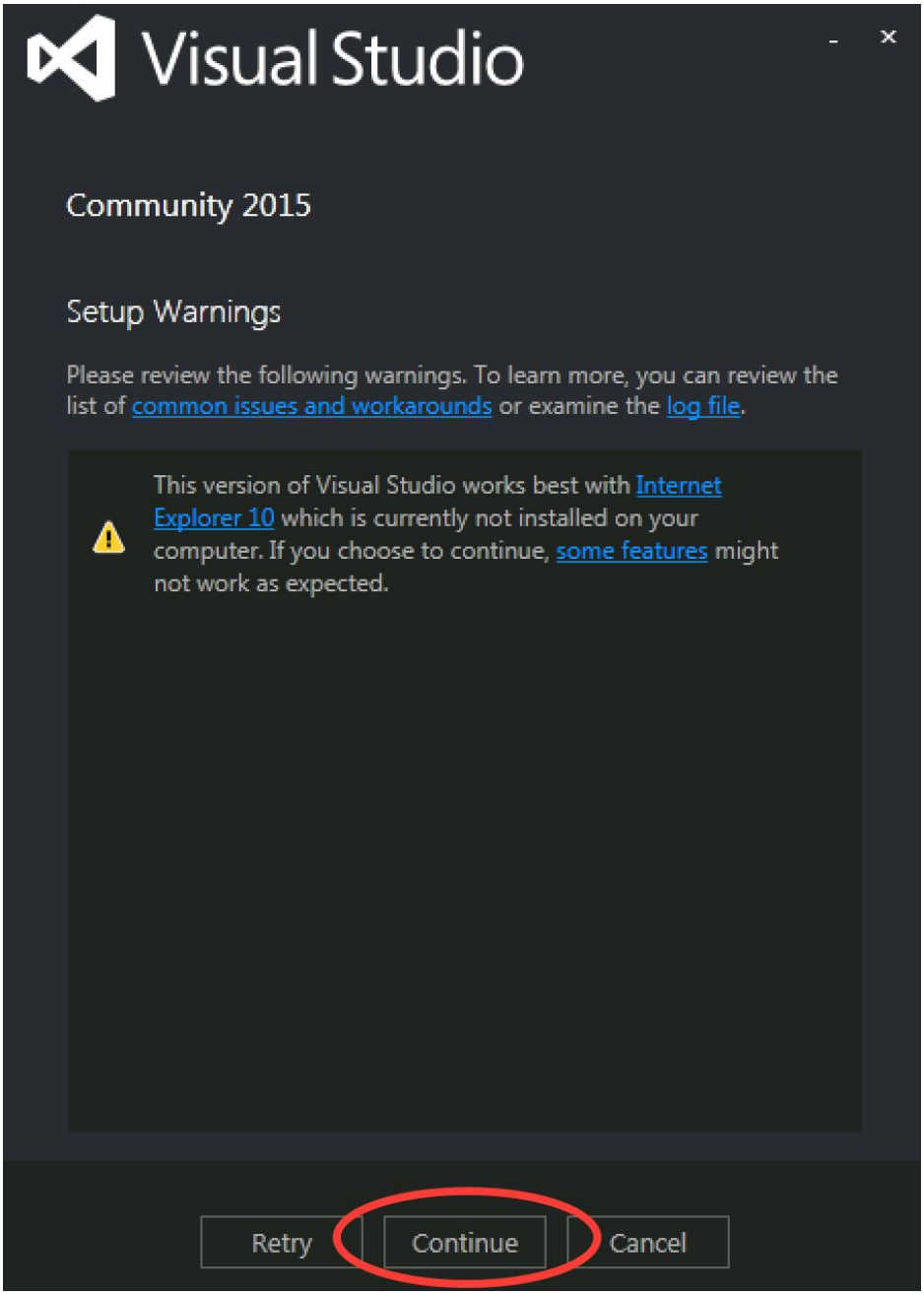
Series Introduction (5m)
Installing Visual Studio 2013 Express for Windows Desktop (8m)
Creating Your First C# Program (13m)

Lab

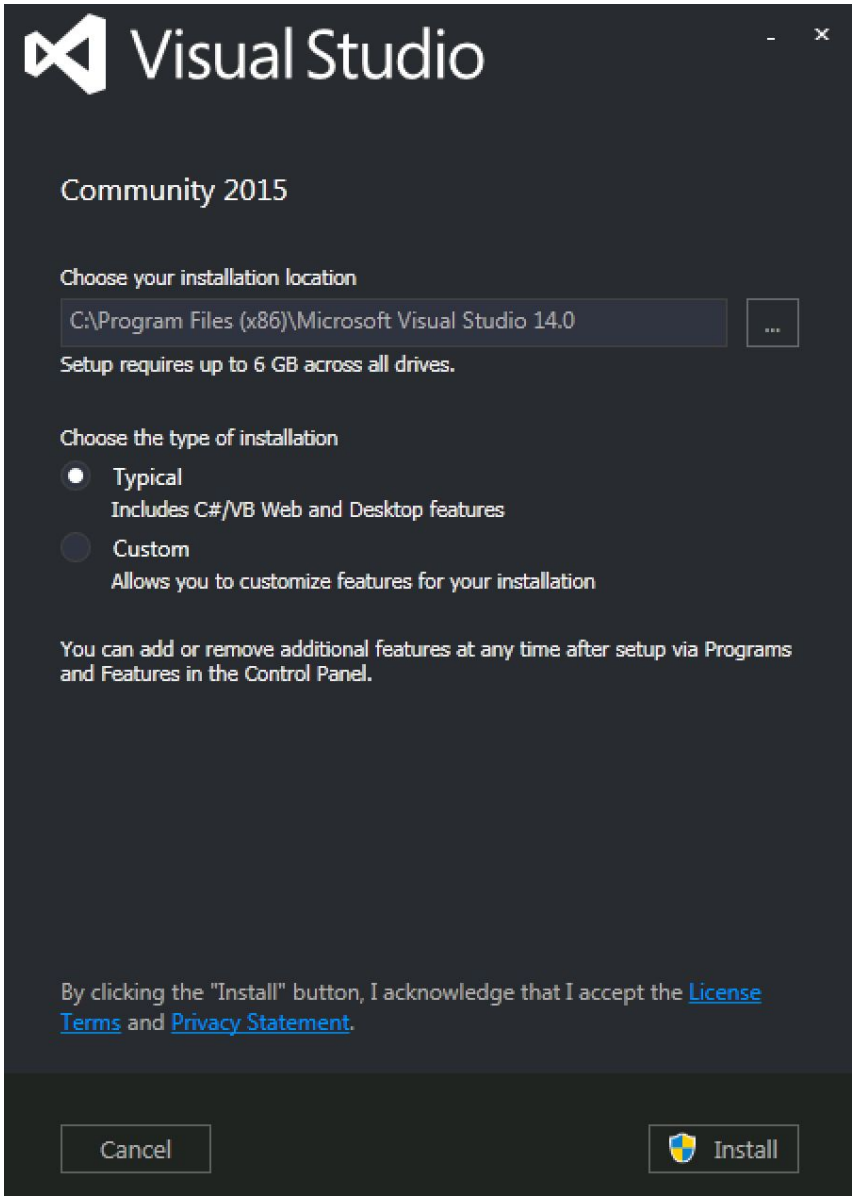
The online course we're following is a couple of years old. Although the C# fundamentals haven't changed since then, the tools have. In July 2015, Microsoft released a fully-featured free Community Edition of Visual Studio 2015 which you can download here. The main difference between the Express and Community Editions is that the Express edition is limited to a single platform (e.g. Web development, C++ development, C# development) whereas with the Community Edition you can develop in multiple languages using a single integrated development environment (IDE). The installation process is similar to that you can see in the video. Although the instructor uses the Visual C# Express Edition, the Visual Studio Community Edition is almost the same and you should have no problem following along.

Just a few notes regarding the installation:

- If the installer prompts you about installing Internet Explorer 10 (IE10), you can just ignore it by clicking “Continue”. You don’t really need IE10.

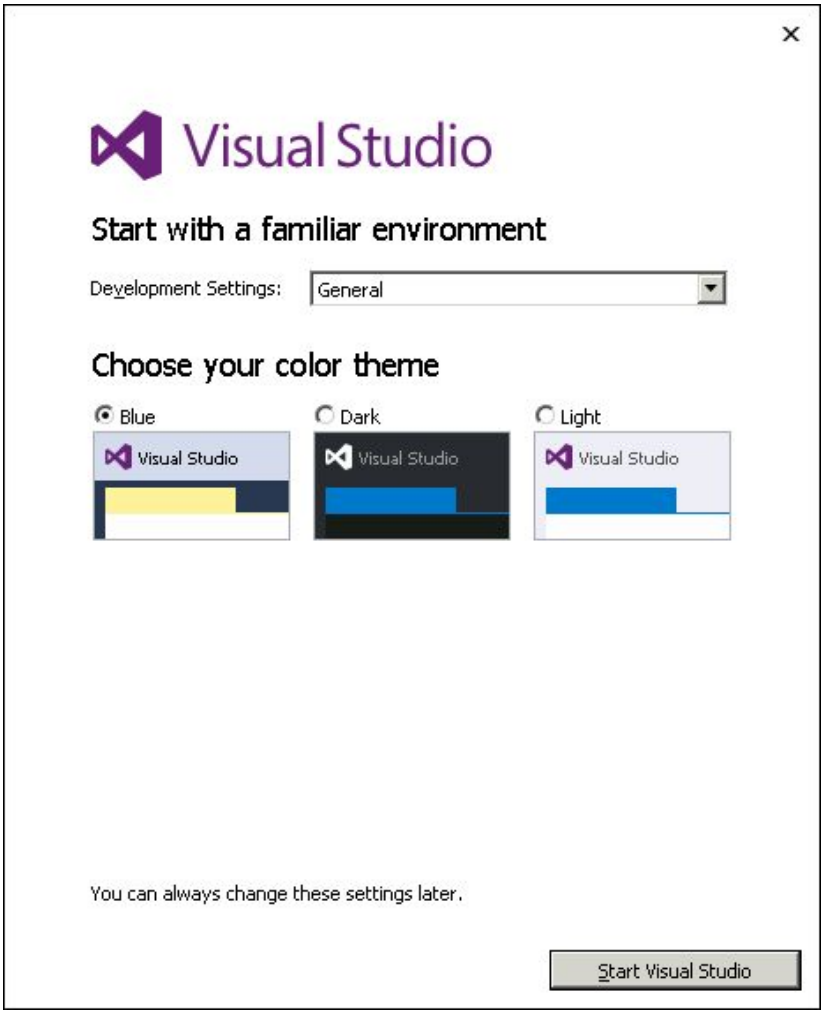


- Keep the default radio button, “Typical”, selected.



- After the installation is complete and you start Visual Studio it might prompt you for a Microsoft ID. You don't have to have one to get started, but in order to continue using Visual Studio for more than a month you will need to get one. To create an account simply follow these instructions. The account is free.

- Finally, when prompted for development settings and color theme, feel free to use the defaults. Just click **Start Visual Studio**.



If you don't have Windows 7 or newer, please refer to your branch manager for assistance with setting up the environment. On non-Windows platforms, your best bet is probably MonoDevelop, which is a free IDE for C# development.

After the IDE is installed, create a Console application in C# which displays a list of your 5 favorite TV shows -- each TV show on a new line. If you get stuck, copy the Hello World program, and copy-paste the line that uses **Console.WriteLine**. Change the printed strings to your favorite shows. For example:

```
Console.WriteLine("The Sopranos");
```

Module 2 - Variables and Values

Time Breakdown

Night #	Objectives
2	Watch all videos Complete the practice questions below
3	Work on the lab (mortgage calculator)

Goals

In this session you will begin to understand the details of the first program you wrote during the last session. You will also become more familiar with the development environment. A good developer's IDE is her best friend, and can make many things go faster and smoother. You will also start writing programs that store information in memory locations, called "variables". Along the way you will learn about debugging, the process of removing errors from your programs.

Videos and Assessments

Dissecting the First C# Program You Created (31m)
Quick Overview of the Visual C# Express Edition IDE (30m)
Declaring Variables and Assigning Values (28m)

While working on the videos, notice there are assessment questions after most clips (these assessments require a one-time registration, which is completely free). Take the time to answer the questions. If you don't know the answer, don't guess -- you can copy the code to Visual Studio and check it yourself. It's important to understand *why* the answer is what it is.

Practice Questions

Before you begin the larger lab, answer the following practice questions by writing code in Visual Studio. You can use the instructor's demo project or create a new Console application.

1. Read a string from the keyboard and print it as a banner, between asterisks (*). For example, if the user types "Hello!", print "**** Hello! ****".
2. Read a string from the keyboard and convert it to an int (use **int.Parse**, as shown in the video). Then, print double the value of that number.

Lab

In this lab, you will develop a mortgage (משכנתא) calculator. With the housing crisis and all, it's important to know exactly how much you are going to pay for a mortgage.

Create a Console application. In the **Main** method, write code that displays a message to the user asking for the house price: "Please enter the house price: ".

Next, write code to read the house price from the user and store it in a string variable. Convert the string variable to an integer variable called **price** using the **int.Parse** method shown in the video. If the user's input is not a valid integer, for example the string "foo foo", this line will fail and cause your program to exit. That's OK for now, but in another module you will learn how to handle errors like these and recover from them.

Repeat these steps to ask the user for the duration of the mortgage, in years. Store it in an integer variable called **years**.

Repeat these steps to ask the user for the yearly interest rate, in percent. Store it in an integer variable called **interestPercent**.

Now, paste the following line of code to create a new variable of type **double**:

```
double interestRate = 1 + (interestPercent / 100.0);
```

The **/** sign stands for division. For example, **7.2 / 2** is the result of dividing 7.2 by 2, which is 3.6. Similarly, the **+** sign stands for addition, and ***** for multiplication. If you have a variable called **price** and you want to triple its value, you can write **price * 3** to multiply it by 3.

Doubles are more precise than ints. You can store fractions (such as 4.27) in double variables, but you cannot store a fraction in an int. This is why we use a double here -- if the interest percent is 8%, then the interest rate is 1.08, and we couldn't put 1.08 in an integer variable.

You're all set to calculate the amount to be paid on the mortgage. The formula is:

$$total = price \times interestRate^{years}$$

To translate this formula to code, you need an exponentiation (העלאה בחזקה) function. The C# function for this is **Math.Pow**. For example, **Math.Pow(1.35, 20)** returns 1.35^{20}, which is approximately 404.

Armed with this knowledge, you should be able to create a new double variable called **total** with the total amount to be paid on the mortgage. Display that value to the user.

Module 3 - Flow Control

Time Breakdown

Night #	Objectives
4	Watch all videos Complete practice questions
5	Work on lab (guessing game)

Goals

In this session you will start working with sophisticated language constructs for branching and looping. These can really make the computer do your bidding, and your programs are going to be so much more interesting!

Videos and Assessments

Branching with the if Decision Statement and the Conditional Operator (19m)
Operators, Expressions, and Statements Duration (14m)
For Iterations (13m)

Practice Questions

1. Create an int variable and put some value in it. Write a decision statement (**if**) that prints the number if it is positive, or prints the negated number if it is negative. In other words, print the number's absolute value (ערך מוחלט).
2. Write a **for** loop that goes over all the numbers from 1 to 100 and prints "Buzz!" whenever the number is divisible by 3. To test whether a number is divisible by 3, use the **%** operator: **if (number % 3 == 0) ...**
3. Write a **for** loop that starts from 2 and prints the first 10 powers of 2: 2, 4, 8, 16, etc.
4. There is a well-known story about the famous 18/19th century mathematician Carl Friedrich Gauss that tells how he, is a kid, came up with the formula to calculate the sum of the numbers 1 to 100. Well, you have the **for** loop, so you don't need Clever Carl's formula (for now). Write a **for** loop that calculates the sum of the numbers 1 to 100.

Lab

In this lab, you will develop a simple guessing game. Your computer will think of a number, and the user will have to guess it.

Create a Console application and paste the following code into the **Main** method:

```
int number = new Random().Next(1, 10);
```

This line of code generates a number between 1 and 9, and stores it in a variable called **number**. The user will have to guess this number, so now we will ask the user for input.Write code to read the user's guess from the keyboard and store it in a variable. The variable should be an int, so use **int.Parse** to convert the user's input.

Now, write code to check if the user's guess is the same as the number your program has thought of. If so, display a nice message: "You won! Way to go!". If the guess was wrong, display another message: "Try again".

At this point, your program only tells the user if she guessed right, but she doesn't get an opportunity to guess again. Write a **for** loop that lets the user make up to 5 guesses at what the number is. Make sure that the line assigning a random value to the **number** variable is *outside* the loop. Think about it -- the computer thinks of a number *once*, and then the user takes multiple guesses at it.

As a bonus, if the user guesses right, terminate the loop by using the **break** statement instead of keeping at it for the full 5 attempts.

If you have time, make the game a little easier for the user. Tell the user whether her guess is smaller than or greater than the number. Use the **<** and **>** operators.

Module 4 - Arrays, Methods, and Files

Time Breakdown

Night #	Objectives
6	Watch the first two videos (arrays and helper methods) Complete practice questions
7	Watch the third video (loops and files) Work on the lab (shopping application)

Goals

Up until now, your program could only interact with the user. In this session, you will learn how to read data from text files, which will make your programs much more useful. You will also learn about arrays, which can store multiple values of the same type. Arrays are useful when you have a large number of values and you don't want to declare a separate variable for each one. Finally, you will also learn how to create helper methods that split your program's logic into multiple pieces that are easier to write and read.

Videos and Assessments

Creating Arrays of Values (19m)
Creating and Calling Simple Overloaded Helper Methods (20m)
While Iterations and Reading Data from a Text File (16m)

Practice Questions

1. Create an array with 10 ints and initialize it with some values. Write a helper method that returns the sum of all the elements in the array.
2. Write a helper method that returns the maximum number in the array.
3. Write a helper method that returns the minimum number in the array.
4. Write a helper method that returns a new array where each element is the square of the original element. For example, if you pass in an array with 1, 2, 3, you get back an array with 1, 4, 9.

Lab

In this lab, you will build a shopping application that the user can use to create a shopping cart and then check out. The shop's catalog will be stored in a text file and loaded by the application.

Create a Console application. Write a helper method above the **Main** method with the following signature:

```
static string[] ReadCatalogFromFile()
```

This method should read the product catalog from a text file called “catalog.txt” that you should create alongside your project. Each product should be on a separate line. For example:

```
milk
bread
butter
diapers
```

Use the instructions in the video to create the file and add it to your project, and to return an array with the first 200 lines from the file (use the **StreamReader** class and a **while** loop to read from the file). If the file has more than 200 lines, ignore them. If the file has less than 200 lines, it’s OK if some of the array elements are empty (null).

Write another helper method with the following signature:

```
static bool IsProductInCatalog(string product, string[] catalog)
```

This method should determine whether the specified product is part of the catalog. The return type, **bool**, can have one of two values: **true** or **false**. The method should go over all the elements in the catalog array (hint: use **foreach**) and compare each element to the product parameter. If an element is equal to the product parameter, return **true** immediately. If you couldn’t find a match, return **false** at the end of the method.

Now it’s time to implement the rest of the program. In the **Main** method, begin by reading the catalog into a local variable by calling **ReadCatalogFromFile**. Then, create an array of 5 strings called **cart**. Create a **while** loop that asks the user for a product to buy, and persists until 5 products have been added. Use the **IsProductInCatalog** method to determine if the product is in the catalog. If it’s not, display an appropriate message. If it is, add the product to the **cart** array.

Finally, after the user provided 5 valid products and you added them to the cart, print the contents of the cart.

Here is an example of what the console could look like after a successful interaction with the user (user input is marked in green -- it doesn’t have to be green in your program, the color is just to make the image clearer):

```
*** Hello! Welcome to the shopping application. ***
Enter a product: red wine
Sorry, the product “red wine” is not in the catalog. Try another product.
Enter a product: salad bowl
```

```
$$$ “salad bowl” has been added to your shopping cart.
Enter a product: avocado
$$$ “avocado” has been added to your shopping cart.
Enter a product: MacBook Air
$$$ “MacBook Air” has been added to your shopping cart.
Enter a product: decanter
$$$ “decanter” has been added to your shopping cart.
Enter a product: mineral water
$$$ “mineral water” has been added to your shopping cart.
*** You’re ready to check out! Here are the products in your shopping cart: ***
salad bowl
avocado
MacBook Air
decanter
mineral water
```

Here are some ideas for improvement if you have spare time:

- Some users might want fewer than 5 items in their cart. If the user provides “quit” as the product name, exit the loop early even though the cart isn’t full yet.
- Sort the shopping cart alphabetically before displaying it during checkout. You can use the **Array.Sort(cart)** method for this.
- Some users might want more than 5 items in their cart. Ask the user for the maximum size of the shopping cart before starting the main loop.

Module 5 - Strings and Dates

Time Breakdown

Night #	Objectives
8	Watch all videos Complete practice questions Start working on lab (on-time flight performance planner)
9	Complete the lab If time permits, move to the next module or implement bonus parts

Goals

In this session you will learn how to perform some advanced operations on strings and dates. Most of the session, though, will be a lab where you analyze flight on-time performance to make more informed decisions during your vacation planning.

Videos and Assessments

Working with Strings (26m)
Working with DateTime (17m)

Practice Questions

1. Write code to ask the user for her name and then print it in uppercase. (Hint: use one of the methods on the **String** class.)
2. Initialize a new **DateTime** variable with today's date. Write a loop that prints the day of the week (Sunday, Monday, etc.) 10 years ago today, 9 years ago today, and so on.

Lab

In this lab, you will help yourself and your friends plan a vacation in the United States in September, during our holidays. As you probably know, flights within the United States are very often delayed, and knowing in advance about expected delays can help plan your trip accordingly.

Download and unzip the file **airline-on-time-performance-sep2014-us.zip**. It contains a single text file[1] (with the .csv extension, which stands for Comma-Separated Values). Here are a few lines from the .csv file:

```
CARRIER,ORIGIN_CITY_NAME,DEST_CITY_NAME,DEP_DELAY,ARR_DELAY,CANCELLED,DISTANCE,
```

[1] The source of the information you used in this lab is the United States Department of Transportation. If you are interested, you can produce similar tables for other date ranges and include additional fields in the report.

```
AA,New York NY,Los Angeles CA,-9,-26,0,2475,
AA,New York NY,Los Angeles CA,2,0,0,2475,
AA,New York NY,Los Angeles CA,-11,5,0,2475,
AA,New York NY,Los Angeles CA,-8,-37,0,2475,
AA,New York NY,Los Angeles CA,-7,-19,0,2475,
AA,New York NY,Los Angeles CA,-4,2,0,2475,
AA,New York NY,Los Angeles CA,-4,-13,0,2475,
AA,New York NY,Los Angeles CA,-4,-6,0,2475,
AA,New York NY,Los Angeles CA,-2,20,0,2475,
```

It is basically a table, like a spreadsheet. You can in fact open the .csv file in Excel or Google Spreadsheets. Each row contains seven values:

1. The airline operator (carrier). For example, “AA” stands for American Airlines.
2. The origin airport city.
3. The destination airport city.
4. The departure delay, in minutes. A positive value means the flight departed later than it was scheduled. A negative value means the flight departed ahead of time. For example, the first flight in the table departed 9 minutes early, and the second flight departed 2 minutes late.
5. The arrival delay, in minutes. This is probably what you care about the most when planning a trip. It’s not that bad if a flight leaves late if it still arrives on time, and it’s very bad if a flight leaves early and still arrives late! For example, the third flight in the table departed 11 minutes early but arrived 5 minutes late.
6. 1 if the flight was cancelled, 0 otherwise.
7. The distance between the origin and destination airports, in miles.

Create a Console application. Write code that asks the user for the origin and destination airports for their September flight.

Write a helper method with the following signature:

```
static void PrintAverageDelay(string origin, string destination)
```

This method should open the .csv file with the flight performance information, and look for all records matching the origin and destination airports specified by the user. It should calculate the number of such records and the sum of the arrival delays, in minutes, and display the average -- which is the sum of arrival delays divided by the number of records. If there were no flights between the two airports, display an appropriate message.

Here is an example of what the console could look like after a successful interaction with the user (user input is marked in green):

```
Please enter the origin airport: Chicago IL
Please enter the destination airport: Denver CO
There were 705 flights between these airports, and the average delay was 16 minutes.
```

You can reuse the code you wrote in previous labs for reading lines from the .csv file. However, this time you need to parse the line to extract specific parts from it. To split a line of values separated by commas (,) into its parts, use the **Split** method, as follows:

```
string[] parts = line.Split(
         new char[] { ',' }, StringSplitOptions.RemoveEmptyEntries);
```

Note that some rows might be invalid, and have fewer than seven parts. You should check that right after splitting, and simply ignore these lines.

Here are some ideas for improvement if you have spare time:

- Display the maximum delay in addition to the average.
- Some people say not all airlines are equally bad. Display the results grouped by airline, so you can make more informed decisions.
- Determine which is the worst airport to fly from, and which is the worst airport to fly to.

Module 6 - Classes

Time Breakdown

Night #	Objectives
10	Watch all videos Complete practice questions
11	Read the CodeProject article Work on the lab (Blackjack game: the Card class and the Deck class)
12	Work on the lab (Blackjack game: the Game class and the main loop) If time permits, implement the bonus

Goals

In this session, you will become acquainted with the principles of object-oriented programming. You will learn about classes, methods, and properties, and understand the idea of encapsulating state and behavior in an object. Inheritance, which is an important tool in object-oriented programming, is also introduced. Finally, in the lab, you implement an interactive Blackjack game -- your first game the computer actually plays against you and you can win or lose!

Videos and Assessments

Understanding and Creating Classes (28m)
More About Classes and Methods (19m)
Working with Classes and Inheritances in the .NET Framework Class Library (34m)

In addition to these videos, you might find it useful to read this CodeProject article about object-oriented programming. The article is rather long, so you don't have to read the whole things. We recommend the following sections:

- Basic concepts, classes, and abstract classes: 4.3, 4.4, 4.5, 4.6, 4.7, 4.8, 4.9, 4.10, 4.11
- Inheritance and polymorphism: 4.16, 4.17, 4.20

It's OK if you don't understand the whole article, but it should help reinforce some of the things covered in the video. If you end up even more confused, don't worry -- you are going to write a bunch of classes in the lab and these ideas will become clearer to you.

Practice Questions

1. Write a **SunTimes** class that has two **DateTime** properties: **SunriseTime** and **SunsetTime**. Write a constructor that initializes the two properties from parameters. Write a method called **DaylightMinutes** that returns the number of minutes of daylight (time between sunrise and sunset).

2. Write a **ShoppingCart** class that has two int properties: **TotalPrice** and **ProductCount**. Write a method with the following signature that adds a product to the cart and updates these properties accordingly: **public void AddProduct(int price)**. Finally, add a method **Checkout** that prints the total price, the product count, and the average product price.

Lab

In this lab, you will implement an interactive Blackjack game. Blackjack is a card game with very simple rules. You can find the full rules in the Blackjack Wikipedia article. The basic idea is the following: players draw playing cards from the deck, which are associated with a numeric value (Ace = 1, Jack = 11, and so on). If a player hits 21, it's a Blackjack and that player wins. If a player exceeds 21, that player loses. If neither player reached 21, and both players don't want to draw another card, the player with the higher score wins.

Our game is going to proceed as follows:

1. The computer (dealer) draws two cards from the deck. If the computer has a Blackjack (e.g., drew a 10 and a Jack (11) = 21), the computer wins. If the computer has more than 21 points, the user wins.
2. The user draws two cards from the deck. If the player has a Blackjack, the user wins. If the player has more than 21 points, the computer wins.
3. The computer always draws another card (even if the computer has 20 already, in which case few players would draw another card). As before, if it's a Blackjack -- the computer wins, if it's above 21 -- the user wins.
4. The user can choose whether to draw another card or not. If the user draws a card, the previous rules apply. If the user doesn't draw a card, and the computer already has a higher score, the computer wins. Otherwise, the computer keeps drawing cards.

This all might sound terribly complicated to program, but we are going to break it into very digestible parts. By using classes to model cards, the playing deck, and the card game itself, we will be able to implement and test each part separately before combining all the parts to a fully-functional game.

The Card Class

First, implement a **Card** class.

The class should have the following *static* methods:

```
public static string[] ValidSuits()
```

The method should return an array of valid suits: "Hearts", "Spades", "Diamonds", and "Clubs".

```
public static string[] ValidRanks()
```

The method should return an array of valid ranks: “Ace”, “2”, …, “10”, “Jack”, “Queen”, “King”.

The class should have the following properties:

- **Suit** -- of type string. Valid values are what **ValidSuits** returns.
- **Rank** -- of type string. Valid values are what **ValidRanks** returns.

The class should also have a constructor that takes a rank and suit:

```
public Card(string suit, string rank)
```

This constructor should initialize the **Suit** and **Rank** properties. As a bonus, verify that the suit and rank have valid values. If not, print a message and/or exit the program by calling **Environment.Exit(1)**.

Finally, the class should have two methods:

```
public int GetValue()
```

The value returned should be from 1 to 13, depending on the rank (Ace = 1, King = 13).

```
public string GetFace()
```

The value returned should be a combination of the suit and rank. For example, “Jack of Spades”, “10 of Diamonds”, “Queen of Clubs”. As a bonus, use symbols to return a string that looks like a card. For example:

```
+-------+
|10     |
|       |
|  ♡    |
|       |
|    10 |
+-------+
```

Test your **Card** class by creating a few cards and verifying that the value returned from **GetValue** and **GetFace** is valid.

The Deck Class

Next, implement a **Deck** class. This class will be initialized with the 52 playing cards, and will let the user and the computer draw cards from the deck. Importantly, after a card is drawn from the deck it has to be removed so you don’t accidentally return it again.

The **Deck** class should have two properties:

```
public Card[] Cards { get; set; }
public Random RandomGenerator { get; set; }
```

The **Deck** class should have a constructor that doesn't take any parameters:

```
public Deck()
```

This constructor has several very important jobs to perform:

1. Set the **Cards** property to a new array of size 52.
2. Initialize each element in the array to the appropriate playing card. The best way to do this is using a **foreach** loop that goes over **Card.ValidRanks()** with another **foreach** loop inside going over **Card.ValidSuits()**. As a result, every combination of valid rank and valid suit (all 52 cards) is added to the array.
3. Set the **RandomGenerator** property to a new object of type **Random**. We will use the **Random** class to draw random cards from the deck, as if the dealer has shuffled the deck before we got it.

Finally, the **Deck** class should have a single method:

```
public Card DrawCard()
```

This method should draw a random card from the deck. Generally, the following line of code produces a random number between 0 and 51:

```
int randomNumber = RandomGenerator.Next(0, 52);
```

To keep track of which cards were already drawn from the deck, you will set the appropriate index in the **Cards** array to null after drawing the card (do not do this yet). It means that when **DrawCard** is called, it is possible that the first card number you generate was already drawn from the deck. Therefore, you should use a loop that repeats the above code until you get a card number that was not already drawn from the deck.

Try to figure this out on your own. If you're stuck, try the following code[2]:

```
int randomNumber = RandomGenerator.Next(0, 52);
while (Cards[randomNumber] == null)
{
```

[2] There is a better way of getting the same result -- with a **do...while** loop. As a bonus, try to find out how the **do...while** loop works in C# and replace this code with a **do...while** loop.

```
  randomNumber = RandomGenerator.Next(0, 52);
}
```

Once you get a card number that was not already drawn from the deck, put it in a new variable called **drawnCard**. Then, set the appropriate index in the **Cards** array to null. Finally, return the **drawnCard** variable from the function. (Make sure you understand now why we needed the temporary variable. Think about what would happen if we first set the index in the **Cards** array to null and then returned it.)

Test the **Deck** class by drawing a few cards. Then, write a **for** loop that draws all 52 cards and prints their faces. Make sure you don't see a card drawn more than once.

Note that our current implementation has an issue. If all 52 cards were drawn and the user tries to draw another card, the **DrawCard** method will enter an infinite loop! Try to understand why this happens. As a bonus, think about how to fix it. (Hint: the **Deck** class can keep track of how many cards were already drawn.)

The Game Class

Finally, the **Game** class will keep track of the state of the game -- the user's score and the computer's score. The **Game** class will also determine if the game is over and who the winner is.

The **Game** class should have the following properties:

- **UserWon** -- of type bool. Starts as false and is set to true if the user wins.
- **ComputerWon** -- of type bool. Starts as false and is set to true if the computer wins.
- **UserScore** -- of type int. Starts at 0 and keeps track of the user's score.
- **ComputerScore** -- of type int. Starts at 0 and keeps track of the computer's score.
- **Deck** -- of type **Deck** (yes, a property can have the same name as its type).

Implement a constructor on the **Game** class that doesn't take any parameters. The constructor should set the **UserWon**, **ComputerWon**, **UserScore**, and **ComputerScore** properties to their initial values. The constructor should also initialize the **Deck** property with a new deck of cards:

```
Deck = new Deck();
```

The **Game** class should also have two methods with the following signatures:

```
public void ComputerMove()
public void UserMove()
```

These methods have very similar logic, except one makes a move for the computer and the other for the user. For example, the **ComputerMove** method should do the following:

1. Draw a card from the deck.
2. Add the card's value to **ComputerScore**.
3. Print the face of the card drawn and the updated score. For example: "Computer drew Jack of Spades, and has 11 points now."
4. If **ComputerScore** is 21, set **ComputerWon** to true.
5. If **ComputerScore** is greater than 21, set **UserWon** to true.

The **UserMove** method should have a symmetric implementation.

Test the **Game** class by calling **ComputerMove** a few times in a row. Then try to intersperse some calls to **ComputerMove** with some calls to **UserMove**.

The Main Method

You're all set to connect the dots and make an awesome blackjack game! Go back to the **Main** method (in the Program.cs file) and implement the game flow. Whenever the game ends, print an appropriate message (such as "Computer won! You'll get lucky next time.").

Try to figure out the game flow yourself first -- on a piece of paper. If you get stuck, here is what it should look like:

1. Create a new **Game** object.
2. Call **ComputerMove** twice. If **UserWon** is true, or **ComputerWon** is true, the game ends.
3. Call **UserMove** twice. If **UserWon** is true, or **ComputerWon** is true, the game ends.
4. *In a loop* (until the game ends):
    a. Ask the user if she wants to draw another card.
    b. If the user says yes, call **UserMove**. If **UserWon** is true, or **ComputerWon** is true, the game ends.
    c. In any case, check the score. If **ComputerScore** is greater than **UserScore**, the game ends.
    d. Finally, call **ComputerMove**. If **ComputerWon** is true, or **UserWon** is true, the game ends.

Here is an example of what the console could look like after a game (user input in green):

```
Welcome to Blackjack!
The house draws Jack of Spades, and has 11 points now.
The house draws 3 of Hearts, and has 14 points now.
You draw 7 of Hearts, and have 7 points now.
You draw 8 of Clubs, and have 15 points now.
Would you like to draw another card (Y/N)? Y
```

```
You draw 4 of Spades, and have 19 points now.
The house draws 2 of Clubs, and has 16 points now.
Would you like to draw another card (Y/N)? N
The house draws 5 of Hearts, and has 21 points now.
BLACKJACK! The house wins.
```

For simplicity, if the user provides invalid input (neither “Y” nor “N”), assume she doesn’t want another card.

As a bonus, you can make the computer play smarter. For example, if the user has 18 and the computer has 20, the computer should probably avoid testing its luck. On the other hand, if the user has 20 and the computer has 19, there’s no harm in drawing another card.

Module 7 - Collections and LINQ

Time Breakdown

Night #	Objectives
13	Watch the first two videos (enumerations and exceptions) Complete practice question #1 Watch the third video (collections) Complete practice questions #2 and #3
14	Watch the fourth video (LINQ) Work on the lab (querying flight records)
15	Work on the lab (querying flight records) Work on the lab (word count)

Goals

In this session, you will learn about another kind of C# type (enumerations), and how to interact with them. You will also see how C# programs can handle errors gracefully without having the whole application crash. In the second part of the session, you will see how to manage large amounts of data in collections and how to perform queries to retrieve only the parts of the data you care about.

Videos and Assessments

Enumerations and the switch Decision Statement (19m)
Gracefully Handling Exceptions (18m)
Working with Collections (34m)
Filtering and Managing Data Collections Using LINQ (23m)

Practice Questions

1. Create a **DayOfWeek** enumeration with a value for each of the seven days of the week. Write a helper method that takes a **DayOfWeek** value and prints a happy message if it's the weekend.
2. Create a **List<string>** and put some of your friends' names in it. Write a helper method that takes this list, asks the user for a person's name, and prints whether that person if your friend.
3. Create a **Dictionary<string, int>** and put some your friends' names and ages in it. Write a helper method that prints your youngest and oldest friends' names.

Lab

Querying Flight Records

In this lab, you are going to experiment with LINQ over a data set that you already familiar with: the on-time flight performance information from Module 5. This time, instead of treating the data

as strings, each row in the data set will be represented as a C# class, which you can query using LINQ.

To save you some time, you can use the **FlightInfo class**, which has properties for each of the data set's fields and a static method **FlightInfo.ReadFlightsFromFile** that returns a **List<FlightInfo>** that you will use to write your queries.

Create a new C# Console application and read all the flights from the file into a local variable called **flights**. Now, for each of the following questions, write a LINQ query that answers that question and print its results using a **foreach** loop. You can use the following skeleton:

```
var flights = FlightInfo.ReadFlightsFromFile(
              "airline-on-time-performance-sep2014-us.csv");
var query = from flight in flights select flight.Origin;
foreach (var row in query)
{
  Console.WriteLine(row);
}
```

Here are the questions you need to answer using LINQ queries (questions marked with (*) are more challenging and can be considered a bonus):

1. What were the arrival delays of all the flights from Boston, MA to Chicago, IL?
2. What were the airports with direct flights to Monterey, CA? (Hint: use **query.Distinct()**)
3. What was the origin and destination of the 10 flights that had the largest arrival delay? (Hint: use **orderby** and **query.Take(10)**)
4. What was the destination airport for the first 20 flights? (Hint: use **flights.Take(20)**)
5. What was the average arrival delay across all flights? (Hint: add **.Average()**)
6. What was the origin and destination of the shortest flight in the data set, and what is the distance of that flight? (Hint: use **orderby** and **query.Take(1)**)
7. What was the longest flight out of San Francisco, CA?
8. The weighted arrival delay of a flight is its arrival delay divided by the distance. What was the flight with the largest weighted arrival delay out of Boston, MA?
9. How many flights from Seattle, WA were not delayed on arrival? (Hint: use **query.Count()**)
10. (*) What were the top 10 origin airports with the largest average departure delays, including the values of these delays? (Hint: use **group by**)
11. (*) What is the airline with the worst average arrival delay on flights from New York, NY?
12. (*) Which origin airport is the worst to fly from in terms of average departure delay if you are flying American Airlines (airline code "AA")?
13. (**) For all the 1-stop flights between Boston, MA and Los Angeles, CA (not including direct flights), what was the flight path with the lowest average arrival and departure delay? (Note that this query can take a very long time to run.)

Word Count

In this lab, you are going to experiment with .NET collections by writing a simple application that displays the most frequent words in a large text file.

Begin by visiting the Project Guthenberg top 100 books and downloading your favorite book as a text file. Take a look at the text file in a text editor (such as Notepad on Windows).

Create a C# Console application. First, implement a helper method with the following signature:

```
static List<string> AllWords(string filename)
```

This method should read the entire file line-by-line and return a list of all the words in the file. To read the file line-by-line, you can either reuse the **StreamReader** approach from one of the previous labs, or use the **File.ReadLines** helper method. To split each line into words, use the **Split** method on strings. It's also a good idea to clean up the words a little bit by removing punctuation, and to convert them to lowercase. For example, "You!" should be stored as "you", "--well." should be stored as "well", and so on.

Next, implement another helper method with the following signature:

```
static Dictionary<string, int> WordFrequencies(List<string> words)
```

This method should take a list of words and create a dictionary that counts the number of times each word occurs.

Finally, in your **Main** method, call the two helper methods to obtain a dictionary of frequencies. Print the top 100 words and their frequencies.

Module 8 - WPF

Time Breakdown

Night #	Objectives
17	Watch all videos Complete practice question #1
18	Read the WPF tutorial Complete practice question #2 If time permits, start working on lab (interactive Blackjack)
19	Work on lab (interactive Blackjack)
20	Work on lab (interactive Blackjack)
21	Work on lab (interactive Blackjack)

Goals

In this module you will learn how to create applications with a graphical user interface (GUI). At the end of the session you will be able to create basic interactive programs where a user can interact with the application using buttons.

Videos and Assessments

Understanding Namespaces and Adding References to Assemblies (27m)
Understanding Scope and Utilizing Accessibility Modifiers (23m)
Understanding Event-Driven Programming (28m)

The last video introduces Windows Presentation Foundation (WPF), a framework for building rich Windows applications with a graphical user interface (GUI). For the final lab, you will need to do a little reading on WPF in addition to watching the video. You will be using the WPF Tutorial, which has more than 100 pages of WPF-related content. But as always, you don't have to read the whole thing! Here are the pieces you need to go through:

- About WPF
    - What is WPF?
- Getting started
    - Hello, WPF!
- XAML
- A WPF application
    - The Window
- Basic controls
- Panels
- Dialogs

- The MessageBox

You can read the whole thing for fun of course, but maybe it's best to do so after completing the final lab.

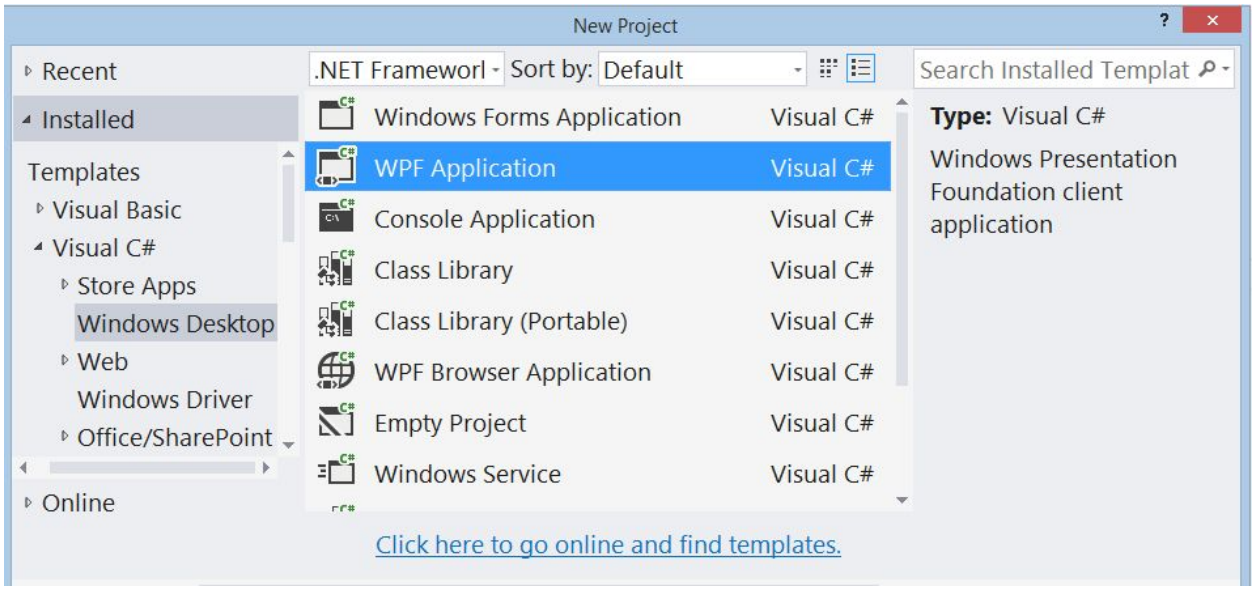
Practice Questions

1. Replicate the WPF application used by the instructor in the demo. It should have the following UI elements: a **TextBox** where the user inputs her name, a **Button** titled "Greet me", and a **Label** that starts out empty. When the user clicks the button, change the label's text to a motivational greeting. (For example: "Dana, you rock!")
2. Add a **PasswordBox** to your WPF application. When the user clicks "Greet me", and the specified password isn't "shecodes", display a **MessageBox** that says "Invalid password!". If the password matches, keep the behavior from the previous step.

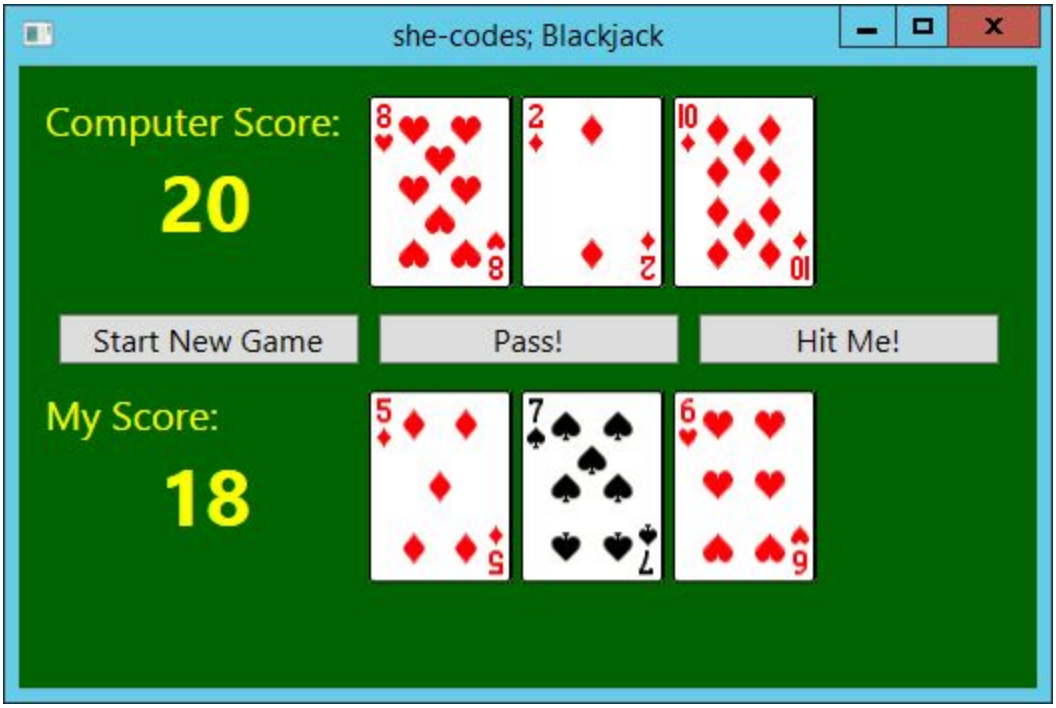
Lab

In this final lab, you will build a GUI application -- a graphical interface to your Blackjack project from Module 6. For example, instead of "printing" cards to the black console with weird characters like - and +, you will display actual images of cards, and instead of getting the user's input from boring Y or N strings, the user will hit a bit button titled "HIT ME!".

Create a new C# WPF Application:



Your application will have only one window. Double click the MainWindow.xaml file if it isn't already open. You can design the user interface with some degree of creativity, but here is a suggested design if you get stuck:



Here are the controls you will need to implement this design:

- "Hit Me!" is a **Button**
- "Pass" is a **Button**
- "Score" and the actual score are **TextBox**es
- The card images are displayed inside a **StackPanel** with horizontal orientation
- The cards themselves are **Image** controls

You can download 52 card face images for free from our GitHub repository.[3] The card face images are named 1.png, 2.png, and so forth (and the organization is by rank -- 1-4.png are aces, 5-8.png are twos, and so on). You will need to write a helper method that takes a **Card** object and returns the appropriate file name.

You should reuse the **Card**, **Deck**, and **Game** classes from your console Blackjack game. **Card** and **Deck** can be used without any modification! In the **Game** class, make the **UserMove** and **ComputerMove** methods return the drawn **Card**. Also, these methods shouldn't print to the console anymore -- there is no console :-)

Instead of a **Main** method that drives these classes from a console, now most of the fun happens in the "Hit Me!" and "Pass!" buttons' click event handlers. Let's recap what needs to happen:

1. The application starts and the user clicks "Hit Me!".

[3] These images are free for personal use. Source: http://www.jfitz.com/cards/

2. The application draws two cards for the computer and two cards for the user.
3. If the user clicks “Hit Me!”, the application draws another card for the user and another card for the computer.
4. If the user clicks “Pass”, the application draws another card for the computer, unless the computer already has a higher score, in which case it wins.

As before, you need to check after each step whether the game is won or lost by one of the players. When the game is over, display a **MessageBox** with the winner’s details, and clear the game board.

About the Authors

Dina Goldshtein is a Senior Software Engineer at BrightSource Energy, a solar energy company. Dina builds infrastructure components in C#, C++, and JavaScript, and spends a lot of time improving processes, tools, and operations.

You can read more about Dina's work on her blog: http://blogs.microsoft.co.il/dinazil/

Sasha Goldshtein is the CTO of Sela Group, a Microsoft C# MVP and Azure MRS, a Pluralsight author, and an international consultant and trainer. Sasha is the author of numerous training courses including .NET Debugging, .NET Performance, Android Application Development, and Modern C++.

You can read more about Sasha's work on his blog: http://blogs.microsoft.co.il/sasha/