Data Types and Enumerations

The data type determines what kind of data can be stored in a variable and the operations that can be on that data.

1. Simple Data Types

We have already used the most common data types:

Type	Size (in bits)	Range	.NET class
int	32	-2147483648 to 2147483647	Int32
char	16		Char
string		Can hold string of any length, including the empty string “”	String
bool	8	true or false	Boolean

Usually `int` will be sufficient for the numbers we are using, however for really large or really small numbers, you can use:

Type	Size (in bits)	Range	.NET class
sbyte	8	-128 to 127	SByte
byte	8	0 to 255	Byte
short	16	-32768 to 32767	Int16
ushort	16	0 to 65535	UInt16
long	64	-9223372036854775808 to 9223372036854775807	Int64
ulong	64	0 to 18446744073709551615	UInt64

For decimal fractions, a double is usually precise enough, but there are also:

Type	Size (in bits)	Precision (in digits)	Range	.NET class
float	32	7	1.5×10^{-45} to 3.4×10^{38}	Single
double	64	15-16	5.0×10^{-324} to 1.7×10^{308}	Double
decimal	128	28-29	1.0×10^{-28} to 7.9×10^{28}	Decimal

2. Structured Data Types

Structs

The *struct* (short for structure) is a data structure that is composed of several pieces of data and no methods.

For example, suppose I want to create a phone list *struct* where I store names and phone numbers.

```
private struct phoneList
{
   public string name;// public scope so able to be used outside the struct
   public int number;
}

private void button1_Click(object sender, EventArgs e)
```

```
{
   phoneList myPhoneList;
   myPhoneList.name = textBox1.Text;
   myPhoneList.number = Convert.ToInt16(textBox2.Text);
}
```

Structs are sometimes called a *value class*, but **DO NOT USE**!! Use classes instead.

Arrays

Arrays are use to store several values of the same type without having to use a different variable for each value. In the following example, the following *OnClick* handler puts values into an array, sums them, and outputs the sum into an edit box.

```
private void button1_Click(object sender, EventArgs e)
{
    int[]  myArray = new int[5];         //declare a local array

    for (int i = 0; i < 5; i++)
    {                                    //initialise array by filling with random
         myArray[i] = random.Next(100);  //numbers
    }

    int sum = 0;                         //initialise variable to hold sum

    for (int i = 0; i < myArray.Length; i++) //sum up the values in the array
    {
          sum += num;
    }

    textBox1.Text = Convert.ToString(sum) //write the result to the TextBox
}
```

Lists

A List is a data structure, similar to array in that they are both linear collections. The advantages of a List are that they resize dynamically and automatically counts the elements it contains.

To create a List of integers:

```
private List<int> myList;              //declaration
```

In the constructor, create the list of integers:

```
myList = new List<int>();              //create the list
```

To add an integer value to the List:

```
myList.Add(4);                         //use the list
```

To count the number of values contained in the List, note, Count is 1-based:

```
myList.Count;
```

To clear the List, so that it no longer contains any values:

```
myList.Clear();
```

Enumerations

Enumerations are used to store a fixed set of values. Unless specified an enumerations will be stored as a zero-based integer value. For example, you might want a direction type that can store one of the values north, south, east or west.

```
public enum eDirection
{
  NORTH = 1,
  SOUTH = 2,
  EAST = 3,
  WEST = 4
}
```

or

```
public enum eDirection
{
  NORTH,
  SOUTH,
  EAST,
  WEST
}
```

3. Complex Data Types

Visual controls, eg Button.

4. Abstract Data Types

eg Stack, Heap… you will meet these in second year.

Task 1: Playing Cards 1

Write a program to simulate the drawing of a random card from a deck of 52 cards (13 cards for each of the 4 suits). This is a useful utility for building card games like poker or solitaire.

1. Create a new project. Put *CardsVersion2* files into the Resources folder. Create enumerations for eSuit and eRank.

2. Place a *Button*, *PictureBox* and a *TextBox* on the *Form*. Set the *Text* property of the *Button* to “Deal a Card” (or equivalent).

3. Create a *Card* class to represent a single playing card. A playing card has a suit and a rank, both of which should be an *enumeration* and an image. The Card class needs a *ToString()* method for writing to the form. This will need to *override* the standard *ToString()* method.

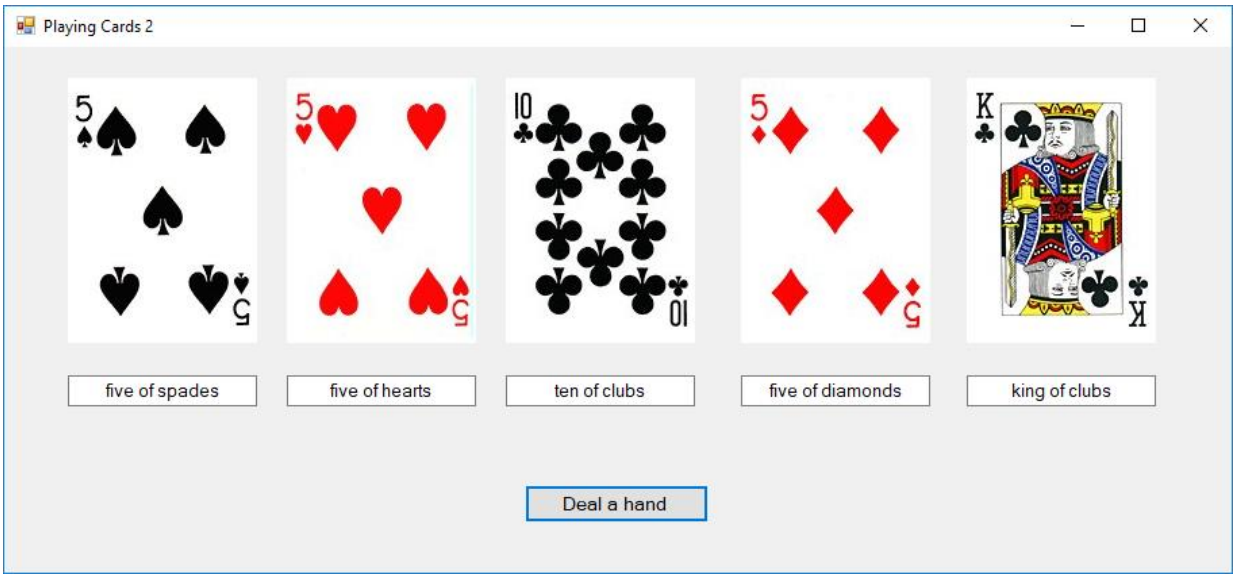
4. Create a *Deck* class which will hold all the possible 52 cards. The constructor will be used to populate the deck. It will also need a *DealACard()* method to choose one card at random and return the chosen *Card*.

5. In the *Form1 class*, create a *Deck* object (an instance of the *Deck* class).

6. The *button1_Click* handler should ask the deck to call its *DealACard()* method. Remember, this method returns a *Card.* Using that card display the card's image in the PictureBox and write the card's String into the TextBox .

Task 2: Playing Cards 2

Write a program to simulate the dealing of a hand of cards, that is, five random cards from a deck of cards.



1. Copy the folder you created for Task1.

2. Place a *Button,* 5 *PictureBoxes* and 5 *TextBoxes* on the *Form*. Set the *Text* property of the *Button* to "Deal a Hand" (or equivalent).

3. In addition to the *Card and Deck* classes, create a *Hand* class to hold the 5 cards (a hand of cards). It will also need a *DealAHand()* method that asks the deck to deal 5 cards from the deck. That is, the deck calls its *DealACard()* method to find a card and the hand stores the card in the cards array in the *Hand* class. Repeat this process 5 times.

4. In the *Form*, make a hand object in the Form, and pass a reference to the deck in the *Hand's* constructor.

5. The *button1_Click* handler should ask the hand to call its *DealAHand()* method. Remember, this method creates a new set of 5 cards. Use the hand to display each card's image in the PictureBox and write each card's String into the TextBox.

6. What is the problem with this implementation?

Task 3: Playing Cards 3

Write a program to simulate the dealing of a five-card hand, that is, five random cards from a deck of 52 cards, with no duplicates. That is, you may get “seven of diamonds” twice in your hand. Real decks of cards don’t do this. Rewrite your program so that it won’t deal duplicate cards. To do this, you need to verify that the suit and rank combination you are about to assign to an array element is not a duplicate of a “card” already selected

---

Refactoring

Refactoring is the process of improving your code, after it has been written, by changing the internal structure but without changing the external behaviour of the code.

Task 4: Playing Cards 4 ******* CHECKPOINT 7 *******

1. Take the code you wrote as Playing Cards 3, add a *Shuffle()* method to the deck class, where 2 cards are randomly selected from the pack and swapped. If this is repeated 20-30 times, the deck will be shuffled.
2. Now to select a hand, we just need to take the first 5 cards.
3. Is this a better implementation? Why?

Task 5: Playing Cards 5

1. Rewrite the previous task using arrays rather than lists.
2. Create a Bridge hand of 13 cards.
3. Create a Dealer class that separates the dealing of cards from the Form1 class.
