Inft2012 Application Programming

Lecture 4

Programming features: constants, formatting output, dates, debugging, exception handling, Math class, strings, Random class, trackbars, timers

Constants

- A *variable* is a name associated with a memory location that can hold different values as the program proceeds
- A *constant* is a name associated with a value that *cannot be changed* as the program proceeds
- If a sporting class won't take members over 90kg, we might say

```
const double dMaxWeight = 90.0;
```

- If a sport has 11 members in a team, we might say

```
const int iTeamSize = 11;
```

- What's the point of constants?

Constants versus literals

- A ‘literal’ value in a program – a number or a string – can appear in many places
- Examples: the GST rate (0.07); a company’s name ("Programming with Style")
- If the value changes (the GST rate goes up to 10%; the company changes its name to "Programs R Us"), every occurrence has to be found and changed
- If we make it a constant (`const string sCoName = "Programming with Style"; const double dGST = 0.07;`) and use the name of that constant throughout the program, only the one occurrence at the beginning needs to be changed

Constants versus variables

- Even if a value is logically a constant, of course it’s possible to program it as a variable – but this is not good practice
- `double dGST = 0.07` might be changed somewhere in the program. It’s unlikely, but if it does happen it will have serious effects.
- `const double dGST = 0.07` *cannot* be changed anywhere in the program. Once set, it remains fixed.
- Some programmers prefer to give their constants completely upper-case names, such as MAXWEIGHT, to distinguish them from variables

Constants versus magic numbers

- The GST rate and a company name might need changing from time to time, but what about this?
```
const double dInchesToCm = 2.54;
```
- The conversion from inches to cm is fixed for ever – so why do we need to make that a named constant?
- To avoid using a 'magic number'
- Magic numbers are numbers in a program that don't have an obvious meaning
- 1.6093 doesn't have an obvious meaning; used in a program, it would be a magic number
- dMilesToKm is a lot clearer; a constant of this name makes the program far more readable

Formatting output

- You might have noticed that some calculations produce results with far more decimal places than could be useful
- When we ask C# to convert a double to a string (as we do before displaying it), it produces as many decimal places as it can, except for 'trailing zeros', extra zeros at the right-hand end after the decimal point
- This is very seldom appropriate, so we need to know how to control the display of numbers
- *Format*, a function method of the String class (with a capital S), can help here. It produces a string (with a small s), but as this is always for display, we will sometimes write about what is displayed

Formatting single numbers

- Format's first argument is a string, and its second is an expression whose value is to be displayed
- In the string, the formatting is controlled by some characters in braces:

```
String.Format("You are {0:d} years old.", iAge)
```

- All the characters outside braces are transferred to the new string
- The characters in braces are replaced in the new string by the value of the following expression (in this case iAge), formatted according to the characters in braces
- If iAge has a value of 17, the example produces the string `"You are 17 years old."`

Numeric format specifiers

- In the braces, the bit after the colon specifies the format for the number; it can be upper or lower case
- n – general number, comma-separated, 2 dec places

```
String.Format("{0:n}", 1234567) gives "1,234,567.00"
```

- d – general integer (decimal number) without commas

```
String.Format("{0:d}", 1234567) gives "1234567"
```

- f – floating point (eg double) number, 2 dec places

```
String.Format("{0:f}", 1234.567) gives "1234.57"
```

- c – currency value, commas, 2 dec places

```
String.Format("{0:c}", 1234.567) gives "$1,234.57"
```

- p – percentage, 2 dec places

```
String.Format("{0:p}", 0.1234567) gives "12.35%"
```

Precision specifiers

- In the braces, the format specifier can be followed by a precision specifier, an integer . . .
- n – number of decimal places

```
String.Format("{0:n0}", 12345) gives "12,345"
```

- d – minimum total places – leading zeros if required

```
String.Format("{0:d7}", 12345) gives "0012345"
```

- f – number of decimal places

```
String.Format("{0:f5}", 1234.567) gives "1234.56700"
```

- c – number of decimal places

```
String.Format("{0:c4}", 1234.567) gives "$1234.5670"
```

- p – number of decimal places

```
String.Format("{0:p4}", 0.1234567) gives "12.3457%"
```


Formatting multiple numbers

- The first character in braces indicates which following expression to use. For historical reasons, 0 means the first one, 1 means the second, and so on. We've been using 0 because we've only had one.
- If we have multiple expressions, we just number them sequentially:

```
String.Format("You are {0:d} years old, {1:n}m tall,
and you weigh {2:n1} kilos.", iAge, dHeight,
dWeight)
```

- will produce something like

```
"You are 17 years old, 1.68m tall, and you weigh
72.7 kilos."
```

- Lec4DemoFormatting illustrates these points

Dates

- C# has a special type, DateTime, for dates and/or times
- A date is not the same as a string, eg "23-Jun-2001", which is treated by C# just as a sequence of characters
- A date is stored not as characters, but using C#'s own internal representation of dates
- A string (for example, input by the user in a textbox) can be converted to a date . . .

```
birthday=Convert.ToDateTime(txbxDate.Text);
```

- . . . but if it's not in the right form, the conversion will fail and the program will crash

Formatting dates with Format

- String.Format handles dates, too:
  - d means the day
  - M means the month (but m doesn't – see below)
  - y means the year
  - (and h, m, and s mean hours, minutes, and seconds)
- In different numbers they indicate different forms, eg
  - yy is the 2-digit year, yyyy is the 4-digit year
  - M is the month number, MMM is the 3-letter abbreviation of the month name, MMMM is the full month name; what do you think MM is?
  - d will display a single digit if appropriate, dd will include a leading zero, ddd & dddd are the day name

Demo – input strings to dates

- Lec4DemoFormatting adds date formats to the number formats discussed earlier
- Enter the input, then click *Make date* to have the inputs combined into a DateTime value
- Examine the code to see how it all works
- Explore different date format settings of your own

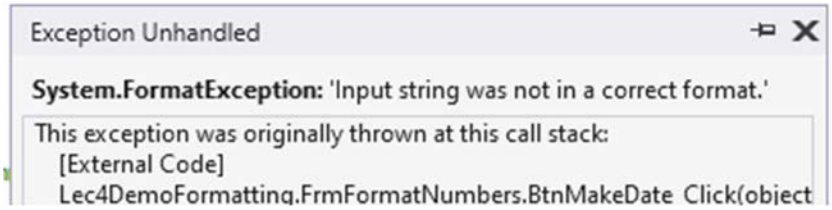
Making dates unambiguous

- Many people write dates as three numbers separated by hyphens or slashes
- These are often ambiguous. When you see 12/11/2021 on a computer, does it mean December 11 (as in the USA) or November 12 (as in the rest of the world)?
- It's wise to avoid ambiguity by using a format that includes the (abbreviated) month name – not just in computer programs, but in the rest of your life
- Nobody confuses 12 Nov 2021 with 11 Dec 2021

Debugging

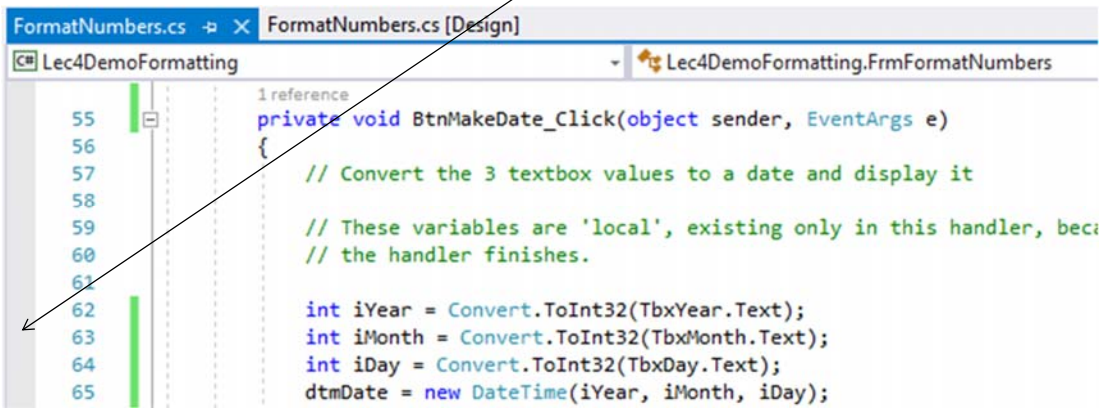
- If you leave a textbox empty in the demo program, or enter invalid data, the program crashes
- The error message and highlit program code can be helpful, but there are other steps we can take to help sort out what's going wrong

```
int iYear = Convert.ToInt32(TbxYear.Text);
int iMonth = Convert.ToInt32(TbxMonth.Text);
int iDay = Convert.ToInt32(TbxDay.Text);
dtmDate = new DateTime(iYear, iMonth, iDay);
```



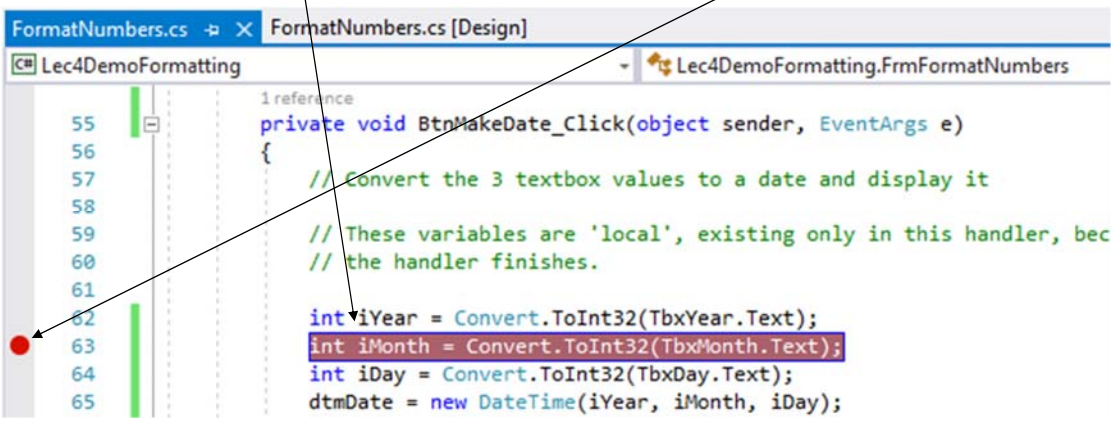
Setting a breakpoint

- If you click in the left grey margin next to a line of code you set a breakpoint



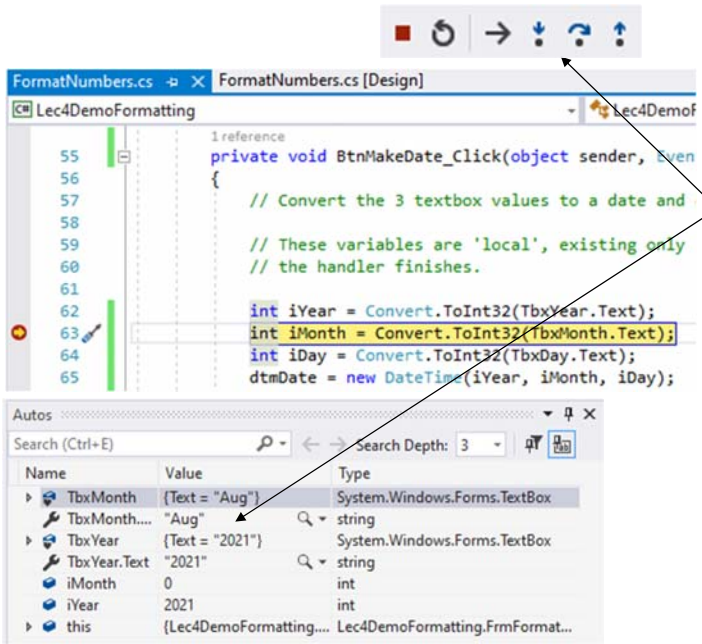
A breakpoint

- The brown bar indicates the breakpoint
- You can have several at the same time
- To remove the breakpoint, click on the brown dot



Debugging

- When the program reaches a break point, execution stops, with the break point highlit



- You can check the values of variables, you can change those values, and you can continue
- The Debug toolbar also has some features that might be useful

Exception handling

- It's really not acceptable for a program to crash when a user enters invalid input
- There are ways of avoiding a crash
- One way is to use the exception-handling tool *try-catch*
  - an *exception* is what it's called when something goes unexpectedly wrong
- If anything goes wrong in the code between *try* and *catch*, instead of crashing, the program executes the code after *catch* – then carries on as if nothing had happened
- If nothing goes wrong, the code after *catch* is ignored
- Let's see how this might work in our program

Try-Catch in the dates demo

- Existing code in MakeDate event handler:

```
int iYear = Convert.ToInt32(TbxYear.Text);
int iMonth = Convert.ToInt32(TbxMonth.Text);
int iDay = Convert.ToInt32(TbxDay.Text);
dtmDate = new DateTime(iYear, iMonth, iDay);
```

- Replacement code:

```
try
{
    int iYear = Convert.ToInt32(TbxYear.Text);
    int iMonth = Convert.ToInt32(TbxMonth.Text);
    int iDay = Convert.ToInt32(TbxDay.Text);
    dtmDate = new DateTime(iYear, iMonth, iDay);
}
catch (FormatException)
{
    MessageBox.Show("Please enter valid numbers for day, month, and year.", "Data format error");
}
```

- Run this with poor data and see the difference!

FormatException?

- Our catch clause has a parameter of type *FormatException*, but we haven't given it a name
- We knew to use FormatException because that was in the title bar of the error message when the code crashed
- We didn't give it a name because at this point we don't need to use the exception
- Here are some variations . . .
- If we accept the default stub (provided by Visual Studio if we press Tab twice after typing *try*), catch will have an unnamed parameter of type *Exception*
- It's also possible to have a catch clause with no following parentheses and no parameter

Different exception types?

- A single try statement can include different catchers for different exception types, each in its own catch clause
- How do we know what exception types to catch?
- The easiest way is to run the code, make it crash, and see what exception type it reports
- Run the program again, entering numbers, but make the day or the month too big
- Now fix the program to catch that exception, too, and to give a different message if it arises

Using the exception object

- If we give a name to the parameter of a catch, we can use the parameter for other purposes
- For example, we might want to give users the C# error message as well as our own (though we'd have to be confident it won't confuse them)
- If the parameter is called exceptionObject, its message is exceptionObject.Message
- We might try something like

```
MessageBox.Show(exceptionObject.Message +
"\nPlease use numbers for day, month, and
year.", "Data entry error");
```

- And a different message for the other exception type

The simplest catch

- If we don't really care what sort of exception can be caused by data entry errors, and we couldn't be bothered finding out,
- we can just write

```
catch or catch(Exception)
```

- instead of

```
catch (<exceptionType> <parameterName>)
```

- When several different exceptions are possible, this doesn't distinguish between them, but so long as we have a suitable error message, that needn't matter

A principle of exception handling

- In general, the more helpful information we can give the user, the better
- The program should now warn the user (and not crash) if they enter non-numeric day, month, or year, or if they enter numeric values that are out of the valid ranges
- But a better program would tell the user exactly which item was at fault, eg "Aug is not a valid month number"
- To do this, you would need each of the three input lines in its own try-catch . . .
- . . . or a single catch that tests all three values and determines which has caused the problem
- Complicated, but very helpful for the user

The date-time picker

- We've seen that getting the user to enter a date using textboxes requires all sorts of checking to make sure it's a legitimate date, and can still lead to problems
- Now try adding a date-time picker to the form, and see how easy it is to use
- You can still type in a date if you want, but the picker does the checking and displays the date unambiguously
- If the date-time picker is called DtpInDate,
  - DtpInDate.Text is a string representation of its date
  - DtpInDate.Value is its actual date, of type DateTime
- A DateTime object has useful properties such as Day, Month, Year, DayOfWeek, and more

Namespaces

- Namespaces are libraries of classes that we can use in our programs
- Classes from a handful of namespaces are automatically available to any new project. These include
  - System
  - System.Drawing
  - System.Windows.Forms
- Some classes provided by C# are not contained in the standard namespaces. To use these, we need to import their namespaces, or to explicitly name their namespaces when naming the class.
- We'll see an example of this soon

Calculations (Math class)

- Most business applications can be written with the mathematical operators we have used so far
- Scientific, engineering, and complex mathematical problems are more likely to require the Math class, with a library of useful functions, such as Sqrt(x), which calculates the square root of its argument
- Either the Math class has to be explicitly named in a calculation, eg `dAns = Math.Sqrt(612.45);`
- or it can be imported right at the start of your program, `using System.Math`, allowing you to simply refer to its functions directly, eg `dAns = Sqrt(612.45);`

Calculations – Math functions

- Some of the Math Class Functions are:
  - Abs(x) – absolute value of x
  - Sin(x), Cos(x), Tan(x) – trigonometric functions for the sine, cosine and tangent of the angle x
  - Ceiling(x) – double equivalent of next integer greater than or equal to x
    - eg Ceiling(13.25) is 14.0
  - Floor(x) – double equivalent of next integer less than or equal to x
    - eg Floor(–9.37) is –10.0
  - Max(x, y) – larger of x and y
  - Min(x, y) – smaller of x and y
  - Pow(x, y) – x raised to the power y

A little more about strings

- We've said that strings are like a cross between primitive types and classes of objects
- They are declared as if they were a primitive type
- But like objects of classes, they have properties that are invoked by <instanceName>.<propertyName>

```
iLength = sStr1.Length;
```

- Likewise, they have methods invoked by <instanceName>.<methodName>, of which we shall soon see a few

String as a collection of characters

- A string can be thought of as a collection of characters, with an index indicating each character's position in the collection:

String:	f	o	u	r	+	t	w	o
Indexes:	0	1	2	3	4	5	6	7

- Actually, *char* is a separate type in C#, one that we don't use; so when we say 'character' here, think 'single-character string'
- So far we've just dealt with whole strings. The 'indexed collection' idea is helpful as we see how to break strings into separate parts.

Transforming strings

- The string class has many useful function methods . . .
- sStr1.ToUpper( ) and sStr1.ToLower( ) produce uppercase and lowercase versions of *sStr1*
- sStr1.Trim( ) produces *sStr1* trimmed of any leading and trailing spaces
- sStr1.Insert(iIndex, sStr2) produces *sStr1* with *sStr2* inserted immediately before the character whose index is *iIndex*
- sStr1.Remove(iStart, iCount) produces *sStr1* with *iCount* characters removed, starting at index *iStart*
- None of these alter *sStr1* – they all produce new strings

Examining strings

- sStr.IndexOf(sStr2) produces an integer indicating at what index *sStr2* is first found in *sStr1*; if it can't be found there at all, it returns –1
- sStr1.LastIndexOf(sStr2) produces an integer indicating at what index *sStr2* is last found in *sStr1*; if it can't be found there at all, it returns –1
- sStr1.Substring(iStart, iCount) produces a string consisting of the *iCount* characters from *sStr1* starting at index *iStart*
- If *iCount* is omitted (sStr.Substring(iStart)), you get the rest of the string from *iStart* onward

There's more

- There are more methods, properties, and functions of strings, but these are enough to achieve lots of serious string processing
- If you want to achieve some particular result, and don't think these methods and properties will do it, see what you can find out about the others
- Lec4DemoStrings demonstrates a number of the features that we've covered here – and one more!
- As with all of the demo programs,
  - read it carefully, to be sure you understand it all; and
  - play with it, seeing what happens if you alter various bits

The car park demo

- We're now going to look at Lec4DemoCarpark to introduce two new controls, the trackbar and the timer
- We'll also introduce random numbers

The Random class

- Generates a stream of 'random' numbers.
- We can declare and instantiate an instance of the class by:

```
Random rndLotto = new Random();
```

Declares the variable *rndLotto* as a name that will be used for an instance of the class Random.
It does not have a concrete instance in memory as yet.

Instantiates: Creates an instance of the class Random in memory and associates it with the name *rndLotto.* While *rndLotto* has properties, it does not yet have any **Values** (it is not ***initialised***).

Random

- Methods of the Random class can be used to produce random numbers. One such method is Next( )

```
int iPick;
iPick = rndLotto.Next(1, 46)
TbxLotto.Text = Convert.ToString(iPick);
```

- Note that this could also be generated by:

```
TbxLotto.Text =
  Convert.ToString(rndLotto.Next(1,46))
```

- Help on Random.Next will tell you that this produces random numbers in the range 1 to 45
- (It starts with a random double from 1 to 45.9999999…, then *truncates* it to give an integer)

Only have one Random

- If you write a program that uses random numbers, only declare one instance of the Random class
- A single instance generates a 'sequence' of random numbers. If you have several instances, each can generate the same sequence, and suddenly the numbers don't look so random
- You can program different variables to draw random values from the same Random instance
- (By the way, they're really 'quasi-random', not random. This means that the sequence of numbers is predictable and in some circumstances repeatable; but that's generally not a problem.)

Trackbars

- A trackbar added from the toolbox provides a graphical display of a number, and permits a quick adjustment of that number
- Its minimum and maximum values can be set at design time using the Properties window, but it might be better to set them from a variable as the program starts
- This is a case for initialisation in the form *constructor*, the method that runs when the form is created
- In the constructor, we can set the trackbar's maximum value directly from, say, the value in a text box:

```
TkbarFullness.Maximum =
  Convert.ToInt32(TbxVacantSpaces.Text);
```

Members: methods and attributes

- A class consists of methods (actions its instances can perform) and attributes (features of its instances, which can have values)
- The car park class (yes, every program is a class) has several event-handling methods and a number of designer-generated methods
- It also has attributes, which we called instance variables
- The methods and attributes of a class are together called its members

Trackbar members

- We're going to use a trackbar called TkbarFullness to indicate and control how full the car park is. A few useful members are . . .
  - TkbarFullness.Minimum – the lowest value represented
  - TkbarFullness.Maximum – the highest value represented
  - TkbarFullness.Value – the actual value currently represented
- All of these can be set or read by the program
  - TkbarFullness.Scroll – the event generated when the user drags along the trackbar using the mouse cursor
- See how these are used in Lec4DemoCarpark

The Timer class

- Instances of classes such as Button are dropped onto a form at design time, and their properties set at design time
- Instances of classes such as Random must be created, and their properties set, at run time; that is, we have to code them
- Timer is another sort of class again. When we drop one on a form . . .
  - at design time it appears in a special 'component tray'
  - at runtime it doesn't appear on the form, but can be accessed by way of code

A timer in action

- To see the timer in action, run the program again, use the trackbar to fill the car park, then press the emergency button and see what happens
- A timer generates its own event, called a Tick. This is one event that isn't triggered by the user
- The Tick event handler for a timer is called every time the Tick event takes place
- While the timer is turned on, the Tick event happens repeatedly, at a specified interval
- We can thus use a timer to program something to happen periodically in our program

Timer members

- Our timer is called TmrEvacuate. A few useful members are . . .
  - TmrEvacuate.Interval – the time between ticks of the timer, in thousandths of a second (milliseconds or ms)
  - TmrEvacuate.Start() – set the timer running
  - TmrEvacuate.Stop() – stop the timer running
- For most uses, the timer interval will remain fixed, but in this example we reduce the interval at every tick, which gives an interesting effect
- See how these are used in Lec4DemoCarpark