Inft2012 Application Programming

Lecture 2

Objects, variables, arithmetic

The drawing project

- Last week we created a new project that did some simple drawing
- We hope that you spent some time trying to work out what the code actually did, but now we'll explain it in some detail
- For our purposes at this stage, there are two important parts to the project
  - the form design: what the form looks like, and the names of the controls on it
  - the code behind the form: the C# code that is intended to run when particular events happen
- Be sure you know how to get to both of these

Event handler for the button

- This is the code that we entered between the braces of the event handler:

```
Graphics graPaper = PicbxDrawing.CreateGraphics();
Pen penBlack = new Pen(Color.Black);
graPaper.DrawRectangle(penBlack, 10, 10, 100, 50);
graPaper.DrawRectangle(penBlack, 10, 75, 100, 100);
Pen penBlue = new Pen(Color.Blue, 3);
graPaper.DrawEllipse(penBlue, 10, 10, 100, 50);
SolidBrush brshGreen = new SolidBrush(Color.Green);
graPaper.FillEllipse(brshGreen, 10, 75, 100, 100);
```

- The name of the event handler reminds us that this is the code that will be executed when the user clicks *BtnDraw*

Objects: declare, instantiate, initialise

- When we need an object in a program, we have to
  - *declare* it: make up a name for it and indicate what type of object it is
  - *instantiate* it: link the name to an actual object (an *instance*) of the right type
  - *initialise* it: give this actual object an initial value (a value that we might choose to change as the program runs)
- Declaring an object is done by indicating what type the object is, then giving the name we choose for the object

```
Graphics graPaper;
Pen penBlack;
```

Choosing names for variables

- Just as with controls, when we name variables we choose a name that is helpful and informative
- Before the bit that tells us what the variable is used for, we have a prefix that tells us what type it is
- The prefix will generally be abbreviated, but will be enough to tell us the type . . .
- *i* for int, *gra* for Graphics, *pen* for Pen, etc
- Prefix naming is a standard practice in Visual Basic, but not in C#
- Even so, we require you to use it in this course, as it can be incredibly helpful in writing and debugging code

Variable names vs control names

- Did you notice the difference?
- For the form itself, and the controls we added to it, we used Pascal case – names starting with a capital letter
- For variables we use 'camel case' – names starting with a lower-case letter
- These are standard conventions in C#
- Also, C# style prefers method names to start with a capital letter, and this becomes easier if control names start with a capital letter

Instantiating objects

- Once we've said `Pen penBlack`, the name penBlack is recognised as being the name of an object of type Pen, but the name doesn't yet have an object it can call its own
- To give it an object we need an assignment:
  ```
  penBlack = new Pen();
  ```
- The word *new* says to create a new object of the specified type; the "=" then assigns this to penBlack
- At this point the new object is still 'empty'; it has no value; but it is an object of type Pen
- Likewise we could instantiate our paper with
  ```
  graPaper = new Graphics();
  ```


Initialising objects

- Now we are ready to give actual values to our objects
- Let's give our pen a colour value of Black:
  ```
  penBlack.Color = Color.Black;
  ```
- We'll give our paper a tricky value: whatever's produced by the CreateGraphics() method of our picture box, which we called PicbxDrawing:
  ```
  graPaper = PicbxDrawing.CreateGraphics();
  ```
- Because this assignment overwrites the previous value, we could forget the previous statement and use this initialisation to both instantiate and initialise

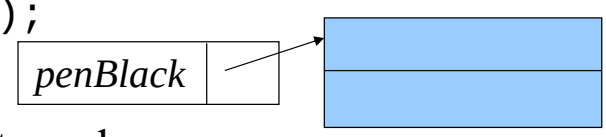
Declare, instantiate, initialise

- For those who prefer pictures to words . . .
- declare – create a name and an associated type

```
Pen penBlack;
```

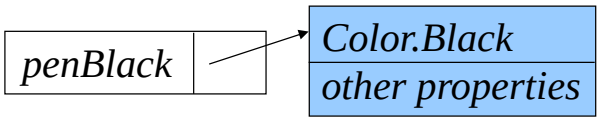
- instantiate – link to an object of that type

```
penBlack = new Pen();
```



- initialise – give the object a value

```
penBlack.Color = Color.Black;
```



Declare, instantiate, initialise together

- It turns out that we can declare, instantiate, and initialise an object in a single statement . . .

```
Graphics graPaper = PicbxDrawing.CreateGraphics();
Pen penBlack = new Pen(Color.Black);
```

- If you do this, try to remember that it's actually three distinct steps
- Note: what has been described on the preceding slides is the general process. It happens that you can't write `Pen penBlack = new Pen();` – the way the *new* operation is defined for Pens, it must have some useful information in the parentheses. However, the description still works as an explanation of the process.

Objects and classes

- Every object is an *instance* of some *class*. We loosely used the word ‘type’ to mean ‘class’ earlier
- Pen and Graphics are classes, and our objects penBlack and graPaper are instances of those classes
- Classes have specified methods (things they can do)
- To get an object to do one of the methods of its class, we write the object name, then a dot, then the method name
- If the method requires any further information, it is provided in parentheses after the method name; that is, we give it *arguments*

Some methods we used

- CreateGraphics() is a method of the PictureBox class, requires no arguments, and returns a Graphics object:

```
graPaper = PixbxDrawing.CreateGraphics();
```

- DrawRectangle() is a method of the Graphics class; its arguments are a Pen, the coordinates of the top left of the rectangle, and the width and height required:

```
graPaper.DrawRectangle(penBlack, 10, 10, 100, 50);
```

- FillEllipse() is a method of the Graphics class; its arguments are a Brush, the coordinates of the top left of the rectangle that encloses the ellipse, and the width and height of that rectangle:

```
graPaper.FillEllipse(brushGreen, 10, 75, 100, 100);
```

Classes without objects

- Some classes exist to provide useful methods and/or properties, but it doesn't make sense to create objects of those classes
- The class Color is an example of this. There's no need to create objects of type Color
- But the class has some very useful properties, which are accessed by writing the *class* name (not an object name because there is no object), then a dot, then the property name
- For example, we used Color.Black, Color.Blue, and Color.Green

Simpler pens and brushes

- We don't actually have to declare and instantiate pens and brushes. We did it that way to help illustrate declaration, instantiation, and initialisation.
- There are Pens and Brushes classes that take a colour as their argument and can be used directly
- Instead of

```
Pen penBlack = new Pen(Color.Black);
graPaper.DrawRectangle(penBlack, 10, 10, 100, 50);
SolidBrush brshGreen = new SolidBrush(Color.Green);
graPaper.FillEllipse(brshGreen, 10, 75, 100, 100);
```

- we can write

```
graPaper.DrawRectangle(Pens.Black,10,10,100,50);
graPaper.FillEllipse(Brushes.Green,10,75,100,100);
```

What if it's gone wrong?

- Visual Studio tries to help you when you're typing code
- It underlines some obvious mistakes in squiggly red (like a spelling checker); if you hover the cursor over one, a tip tells you very briefly what it thinks the problem is
- The problem is also listed in the Error List at the foot of the screen; if you double-click on the error in the Error List, the relevant piece of code will be highlighted
- It can take lots of practice to learn to understand the messages. To start with, just compare what you've written very closely with what you should have written

More help from Visual Studio

- You might also have noticed that when you're typing code, Visual Studio offers help to complete some parts
- As you start a word, a drop-down menu can offer a list of options. When the right one is selected, pressing Tab will complete it.
- When you type a method name and the opening parenthesis, a box will list the possible arguments for that method, with a brief explanation of what they mean
- When you hover the cursor over pretty much anything in the code window, a box will give you lots of information about it

Have syntax errors, won't run

- By now you know that a program with syntax errors, errors in the form of what you've written, won't run
- If you want to run programs, you learn to find and get rid of syntax errors
- As you should also know, getting rid of syntax errors is often not the end of the story
- A program that runs but does the wrong thing is still wrong
- Getting rid of all errors, so that the program runs *and* does what was intended, is called 'debugging'

Primitive variables

- Not every data item in a program is an object
- Simpler data items such as numbers are called *primitive types*
- C# has many types of number, but at this stage we only need two:
  - int (ie integer): eg –13, 0, 234742, 7
  - double: numbers that can have a fractional part, such as –13.3, 0.00001, 234742.99, 7.0
- Note that 7.0 is a double, not an int, even though its fractional part is 0
- ints and doubles are stored differently, so when we want a variable we have to tell C# which type it is

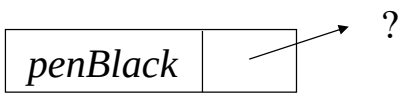
Primitive variables: declare, initialise

- Objects must be declared, instantiated, and initialised
- Primitive variables only need to be declared and initialised
- Declaring them tells the computer what name to use and how much storage to set aside for it
- Initialising them gives them a value. As with objects, we initialise with an assignment statement, whose right side can be any appropriate (eg arithmetic) expression.
- *Remember: an assignment statement takes the value of the expression on the right of the equals sign, and gives that value to what's on the left of the equals sign*

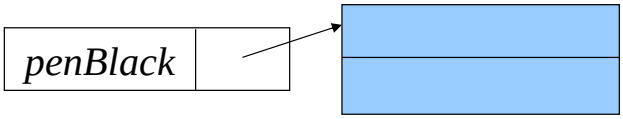
Why instantiate objects?

- Every declared name has a little bit of associated storage.

- Objects are complex and can require varying amounts of memory, so what goes in the storage is a 'pointer' to another part of storage where the object is stored.

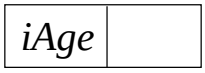


- Instantiating an object sets aside the area where it's actually going to be stored, and sets the pointer to point to that storage.

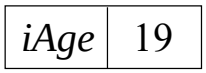


Why not instantiate primitives?

- Every declared name has a little bit of associated storage



- Primitives are much simpler than objects, and require a small fixed amount of memory, so their value can actually go in the little bit of associated storage



- Declaring a primitive automatically sets aside the storage it needs, and we don't need an instantiation step to do this

Declaring variables

- To declare a variable, first give the type of the variable, then give the name you've chosen for the variable

```
int iAge;
double dWeight;
```

- We can declare several variables of the same type in the same declaration

```
double dBoatLength, dBoatWidth, dBoatDepth;
```

- We normally only do this if the variables are logically related
- We can declare and initialise a variable in a single statement, remembering that it's actually two steps:

```
double dWeight = 61.75;
```

Arithmetic operators

\+ addition: 6 + 3 is 9

\- subtraction: 5.3 - 2 is 3.3

* multiplication: 3.1 * 2 is 6.2

/ division

If either operand is a double, so is the answer:

5.0 / 2 is 2.5

If both are integers, so is the answer:

9 / 2 is 4, 3 / 7 is 0

% modulo (the remainder after integer division):

9 % 10 is 9, 12 % 10 is 2, 12 % 7 is 5, 7 % 3 is 1

10 into 93 goes 9 remainder 3, so 93 / 10 is 9 and 93 % 10 is 3

Operator precedence

- We need to know the order in which operators in an expression will be applied
- 3 levels of 'precedence': parentheses; division & multiplication; addition & subtraction
  - 2 + (45 + 4) * 3 is calculated as (2 + ((45+4) * 3))
  - the number of right parentheses ")" must match the number of left parentheses "("
  - AND they need to be in the right places
- If two operations are of the same precedence, execute them from left to right:

  3 * 4 / 2 is calculated as (3 * 4) / 2

Sensible use of parentheses

- In programming, 'round brackets' are called *parentheses*, 'square brackets' are called *brackets*, and 'curly brackets' are called *braces*
- Brackets are not used in expressions, but parentheses can be 'nested', set within each other, as required . . .
  - 15 * (3 + 8 * (17 – 9)) + 4
- Some programmers like to add extra parentheses. Simon thinks this shows a poor understanding of precedence:
  - (((x + 7 + 10) * (1000 – 8)) / (900 + 90 + 2)) – 17
  - (x + 7 + 10) * (1000 – 8) / (900 + 90 + 2) – 17
  - This formula illustrates a fine verse from Lewis Carroll's *Hunting of the Snark*

String – an in-between type

- String is a type we will use a lot
- A string is a sequence of characters, such as a person's name
- We think of strings as a primitive type; they're not really, but they behave like one in many ways
- Like primitive types, strings need to be declared and initialised, but not instantiated
- When we write a literal string, we enclose it in quotation marks:

```
string sOtherNames = "Han Kee";
string sSurname = "Tan";
string sFullName = "Simon";
```

Concatenation of strings

- We can do almost as many things with strings as we can with numbers, but for now we'll settle for concatenation – joining together
- Strings are concatenated using the operator +; a poor choice, because it means something so different in maths
- Don't expect C# to insert spaces the way a word processor will sometimes do; you must write them in

```
sName = sOtherNames + " " + sSurname;
sMessage = "You've eaten " + sCount +
            " ice creams.";
```


Input and output

- Most computer programs have *input*: information provided in some way by the user
- The many ways of providing input include clicking on controls and typing text in particular controls
- (Other ways include getting the program to read a file)
- All (useful) programs have *output*: information provided in some way to the user
- The many ways of providing output include placing text in particular controls and displaying it in separate windows
- (Other ways include playing sounds, controlling devices, producing and storing files)

Output via message boxes

- A message box is a small window used to output information
- The typical message box has a title, a message, and an OK button – though there are other configurations
- When a program displays a message box, the program waits until the OK button has been clicked before continuing
- MessageBox is another class with methods but no objects; the method we're interested in is Show()

```
MessageBox.Show("Message", "Title");
MessageBox.Show("Hello, World!",
                "Example");
```


Output via text boxes

- The TextBox control, like most controls, has a Text property; this property determines what is displayed in the box
- We've changed properties such as Text at design time, using the Properties window
- Properties of controls can also be changed by the program, using an assignment statement
- This is one way that a program can produce output:

```
tbxSurname.Text = "Tan";
tbxOtherNames.Text = sOtherNames;
```

- *Remember: an assignment statement takes the value ... and gives that value to ...* (Please fill in the blanks)

Input via text boxes

- Just as a program can put text in a text box, it can also 'read' the text that is there
- One way of doing this is by assigning it to a variable, which it can then use as required

```
sOtherNames = tbxOtherNames.Text;
sSurname = tbxSurname.Text;
```

- Note that assignment must be to a variable or a property; it wouldn't make sense to try something like

```
"Tan" = tbxSurname.Text;
```

- This is because assignment changes the value of the *variable* or *property* on its left side; "Tan" isn't a variable or property and we can't change its value

Input of numbers from text boxes

- If the user types their age into a textbox, we can't get that age with `iAge = tbxAge.Text;` because iAge is an integer and tbxAge.Text is a string
- We have to find the integer corresponding to the string and store that in the integer variable
- *Convert* is a class containing methods that we can use to convert between various types
- The conversion doesn't change the original value (for example, it doesn't somehow change the textbox text from a string to an int): it returns the corresponding value of the required type (it gives us the int equivalent of the textbox text)

Some useful conversions

- Convert a string to an int

```
iAge = Convert.ToInt32(tbxAge.Text);
```

- Why Int32? int is a type specific to C#, while Int32 is a more generic type for all of the Visual Studio languages
- Convert a string to a double

```
dWeight = Convert.ToDouble(tbxWeight.Text);
```

- Convert an int to a string

```
tbxAge.Text = Convert.ToString(iAge);
```

- Convert a double to a string

```
tbxWeight.Text = Convert.ToString(dWeight);
```


Converting between numbers

- Sometimes we need to convert one type of number to another
- For example, we want to divide two integers and get a double result, so we need to convert one of the integers to a double; or we want to discard everything after the decimal point of a double
- We could use Convert, but a more traditional method is called 'casting', in which we put the desired type in parentheses before the value to be converted

```
dRatio = (double) iLength / iWidth;
iDollars = (int) dCost;
```

- Note that (int) doesn't round, it truncates

Combined assignment operators

- We often find the same variable on both sides of an assignment statement:

```
iCount = iCount + 1;
dStrength = dStrength * 1.1;
sToppings = sToppings + ", anchovy";
```

- C# provides a shorthand form of these assignments that avoids repeating the variable name:

```
iCount += 1;
dStrength *= 1.1;
sToppings += ", anchovy";
```


Combined assignment operators

- If the combined assignment operators confuse you, don't use them! The standard forms are more explicit and clearer.
- The standard forms are also less prone to tricky errors. If you mean to write

```
iCount += 1;
```

but accidentally write

```
iCount =+ 1;
```

what will C# do? And how easy will it be to find and correct this error?

Comments in code

- By now you should know the importance of comments
- A comment in C# is anything following a pair of slashes on the same line; Visual Studio displays it in green, and otherwise completely ignores it
- Comments should be written . . .

– at the start of each program, to say who wrote it, and when, and why (ie what it does)

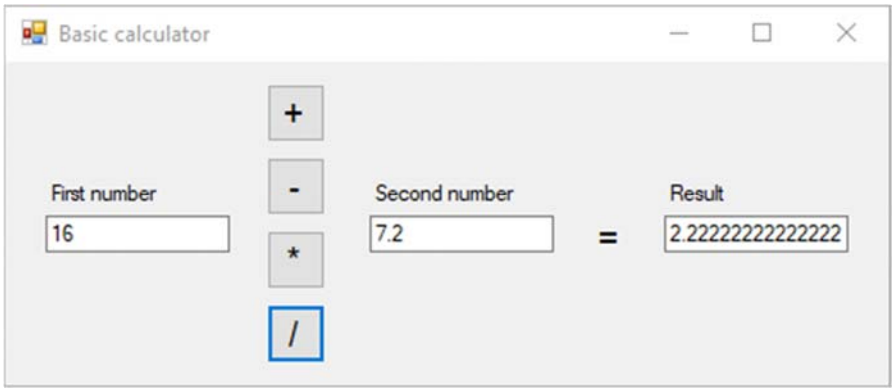
– at the start of each method, to say what the method does

– within the code, to explain aspects that might not be obvious to the reader

- Comments should *not* be written to explain what is completely obvious

Example: a calculator

- Let's put much of this together into a simple example
- Add, subtract, multiply, or divide numbers in textboxes
- For good measure, output a message box as well
- Examine the program well

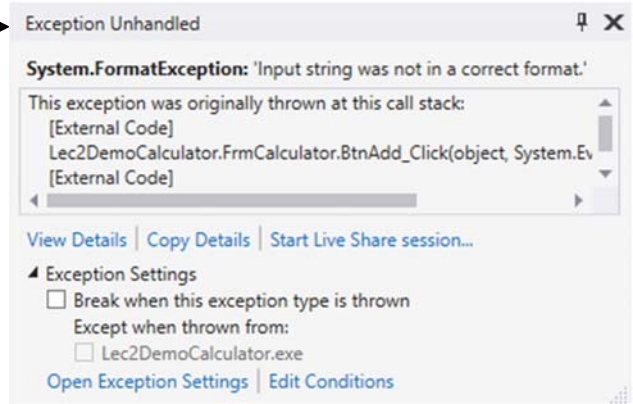


Points to note

- All of the buttons and textboxes are well named
- Lots of helpful comments
- Each event handler uses a different combination of variables and text boxes to do the job; look carefully at each one to see how they work
- We wouldn't normally output results in a textbox AND a message box – message boxes slow everything down; this is just for illustration
- What happens if we put a non-number in a textbox?
- What happens if we click a button when an input textbox is empty?
- What happens if we divide by zero?

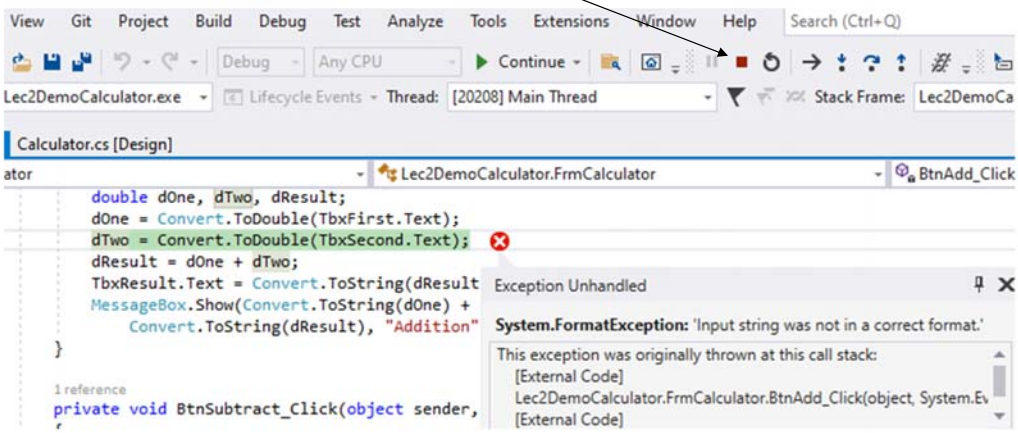
Dealing with runtime errors

- A runtime error occurs when the program has no syntax errors, runs correctly, but can't handle something that it encounters while running
- The program stops and an error window is displayed
- Be sure to read the top lines, which tell you what the error was



Stop the program

- After a runtime error, the program appears to have stopped, but it's still debugging and waiting to see what you do next
- Click the red square (Stop Debugging) to stop it properly and return to the code window



Then find and fix the problem

- At this point many student programmers try changing things almost randomly in the hope that eventually the problem will go away
- This might eventually get the program running, but the chances are that it won't be doing what it's supposed to do
- Debugging is much faster and more effective if you take the time to read the error message and understand what it's telling you
- If you can't do this by yourself, ask your tutor for help – and try to learn from what the tutor shows you

It's not enough to get it running

- Remember, getting a program running isn't enough
- It also has to produce correct output
- Take a look at Lec2DemoIntegerDivision
- What's going on here?
- The Demo name should give you a hint
- When you have a program running, it's always important to check the output that it's producing and make sure that it's the output it should be producing