Inft2012 Application Programming

Lecture 6

Multiple forms, parameters

The mark of the web

- When you download the demo program for this week and try to run it, you will probably be told "Couldn't process file AboutBox.resx due to its being in the Internet or Restricted zone or having the mark of the web on the file. Remove the mark of the web if you want to process these files."
- This is a security measure, perhaps a little overblown
- In the folder of files for the project, find the file AboutBox.resx; right-click on it and choose Properties; at the bottom of the General tab, find the checkbox labelled Unblock, and tick it
- (Or do the same to the zip file before unzipping it)
- Re-open the project, and it should now run

Multiple forms

- When you start a C# project, it automatically has a single form
- Sometimes you might want other forms to appear and disappear, for example when a user clicks a button
- For this you need to know how to
  - add a form to a project
  - get the form to appear
  - get the form to disappear

Adding a new form

- Select Project / Add Windows Form and enter a suitable name for the new form
- Add controls to the form, and code behind the controls, just as you've been doing with a single form

Getting the form to appear

- In the class for your main form, you need to declare and instantiate an instance of your new form:

```
InputForm FrmCollector = new InputForm();
```

- When you want the form to appear, call its Show method:

```
FrmCollector.Show();
```

- This makes the new form appear
- If you want the user to be unable to access the main form while this one is active, instead write

```
FrmCollector.ShowDialog();
```

- As a side-effect, this shows the form in a 'modal' manner, which has the effect mentioned above

Getting the form to disappear

- Unless you disable this facility, the form will disappear when the user clicks the close button in its top right corner
- However, it's sometimes better to provide an explicit button with text that makes it clear this form will close and return the user to the other form. Examples might be Close, Submit, Return, Back, etc
- The actual line of code that closes the form is `this.Dispose();`
- Of course there might be other things to do just before closing the form, such as registering any data that's been input

OK and Cancel buttons

- If your form has OK and Cancel buttons, and you want your program to recognise which was pressed, first set the DialogResult property of each button
- When you show the form with the ShowDialog method, it will return a value of Dialog.OK or Dialog.Cancel (or other Dialog results) according to which button was pressed
- If you want the user to have keyboard shortcuts, in the properties of the form choose an AcceptButton (which is considered clicked when the user presses Enter) and a CancelButton (which is considered clicked when the user presses Esc(ape)

Naming convention for controls

- We've told you to give each control an informative name preceded by an abbreviation of the type of control, eg TbxName
- There are variations on this convention, but they should be used with care; in particular, try not to use the name followed by the control type, eg NameTextbox
- AcceptButton and CancelButton are properties of a form; if you use the same names for the buttons themselves, the button names replace the property names, so the properties no longer work

An InputBox form

- Some languages have a useful feature called an InputBox. Like a MessageBox, it displays a message; but it also has a textbox, and returns whatever text the user types into the textbox.
- C# doesn't have such a feature, but it's easy to write one
- It's a separate form, which is Shown as required, and has a property (sInputValue) that contains the Text from the textbox
- Nothing in its code explicitly disposes of the form; when a form's AcceptButton and CancelButton properties are set, these buttons do that automatically

Multiple constructors

- We have seen that a form's constructor is the method that is called when the form is initialised (with *new*)
- We have seen that the constructor has the same name as the form's class
- In the InputBox we see that a class can include multiple constructors, so long as they have different signatures (different combinations of parameters)
- When *new* is called to initialise a form, C# checks the arguments, and chooses the constructor with corresponding parameters
- Every constructor must start by creating the form itself; that is, it must start with InitializeComponent();

Properties

- Objects (including forms) can have properties, sort of like variables, accessible from outside the object
- To outside methods they appear like public instance variables, but they are structured quite differently
- They have inbuilt methods called *get* and *set*
- *get* and *set* are never called by name, the way most other methods are called
- Instead, *get* is called when an external method uses the property in an expression (ie gets its value),
- and *set* is called when an external method uses the property on the left of an assignment (ie sets its value)

Parameters and arguments

- Reminder: for a method to use the values passed in, it has to have names for them; these are the *parameters* we specify when writing the method:

```
private void Flag(int iLeft, int iTop, int
                                    iSize)
```

- When we call a method we provide corresponding *arguments*:

```
Flag(100, 90, 180);
```

- C# then executes the Flag method, with
  its first parameter (iLeft) taking the value 100,
  its second parameter (iTop) taking the value 90,
  and its third parameter (iSize) taking the value 180

Returning multiple values

- A function method takes one or more values (as parameters) and returns a single value
- What if we want to return more than one value?
- For example, the roots of the quadratic equation

$y = ax^2 + bx + c$

are given by $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

- A *quadratic equation* simply means an equation of the form $y = ax^2 + bx + c$
- The *roots* are the values of x for which y is 0
- There are 2 roots, one found by taking the + of $\pm$, the other by taking the –

A function can't return 2 values

- We might like to try this, but it won't work:

```
private double double QuadRoots(double a,
        double b, double c) {
   return (-b+Math.Sqrt(b*b-4*a*c))/(2*a);
   return (-b-Math.Sqrt(b*b-4*a*c))/(2*a);
    }
```

- We can't specify the method type as "double double"
- There's no point in having two return statements, because as soon as the first is executed, the method stops running and 'returns control' to the part of the program that called it

We've been using value parameters

- In the methods we've seen so far, when the method is called, the *value* of each argument is given to the corresponding parameter
- If the method changes the value of the parameter, that change stays within the method; the parameter behaves like a local variable
- Even if the arguments are variables, changing the values of the parameters doesn't change the values of the arguments
- Remember the example on holes in scope
- Passing parameters by value is the default in C#, but it's not the only way

An alternative – out parameters

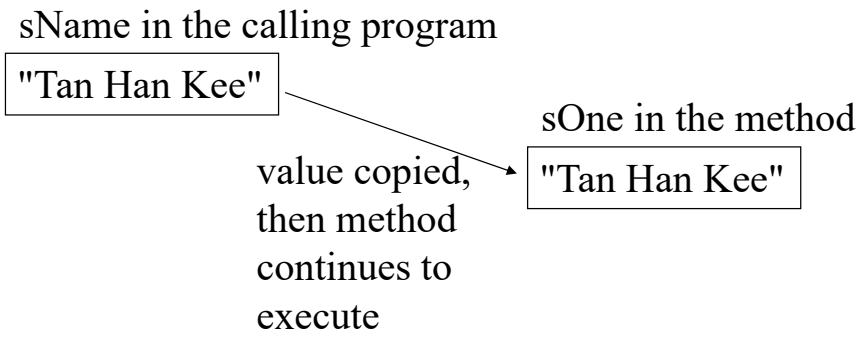
- An *out* parameter is a parameter that will be 'passed out' to the corresponding argument when the method finishes
- The argument corresponding to an out parameter must be a variable, because it must be able to take on the value that is passed out
- Both the parameter and the argument are marked with the word "out"
- There's no limit on the number of parameters, so if we want to return more than one value we can do it using out parameters

Value parameters

- Method declaration:
  ```
  private void MyMethod(string sOne)
  ```
- Method call:
  ```
  MyMethod(sName);
  ```
- When the method starts executing (its code is performed), *sOne* (the parameter) takes the value of *sName* (the argument) as its own initial value
- If the method is called with
  ```
  MyMethod("Tan Han Kee");
  ```
  then the method is executed (its code is performed) with *sOne* (the parameter) taking "Tan Han Kee" (the argument) as its own initial value

Value parameters in pictures

- Method declaration:
  ```
  private void MyMethod(string sOne)
  ```
- Method call:
  ```
  MyMethod(sName);
  ```

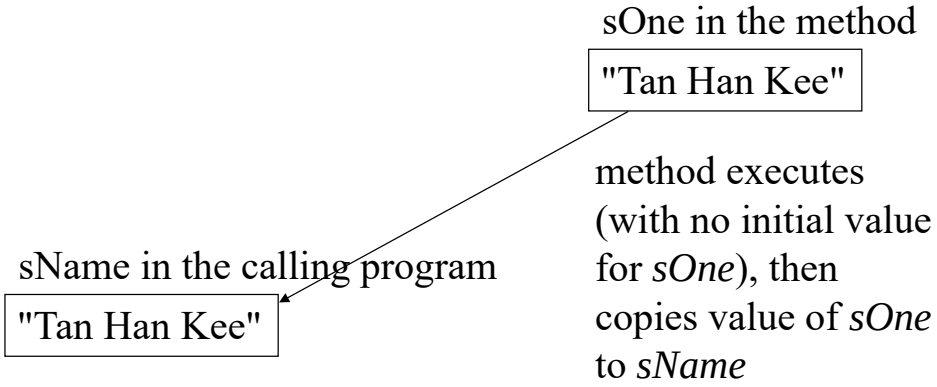


Out parameters

- Method declared:
  ```
  private void MyMethod(out string sOne)
  ```
- Method called:
  ```
  MyMethod(out sName);
  ```
- When the method finishes executing, *sName* (the argument) is assigned the value of *sOne* (the parameter)
- The method cannot be called with
  ```
  MyMethod(out "Tan Han Kee");
  ```
  because "Tan Han Kee" (the argument) is a literal, not a variable that can be assigned a value

Out parameters in pictures

- Method declaration:
  ```
  private void MyMethod(out string sOne)
  ```
- Method call:
  ```
  MyMethod(out sName);
  ```



And more – reference parameters

- A third kind of parameter is a reference parameter: when the method is called the parameter becomes a *reference* to the argument – another name for the same variable
- Therefore any change made to the value of the parameter is also made to the value of the argument
- Both the parameter and the argument are marked “ref”
- When we pass arguments by reference, they must be variables, so that their values can be changed
- A reference parameter is thus a way for a method to change or update the value(s) of one or more variables

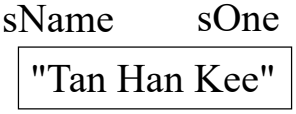
Reference parameters in pictures

- Method declaration:

```
private void MyMethod(ref string sOne)
```

- Method call:

```
MyMethod(ref sName);
```



2 names (references) for the same variable, then method continues to execute

Returning two values

```
private void QuadRoots(double a, double b,
   double c, out double x1, out double x2)
   {
   x1 = (-b+Math.Sqrt(b*b-4*a*c))/(2*a);
   x2 = (-b-Math.Sqrt(b*b-4*a*c))/(2*a);
    }
```

- As a general rule, use value parameters to pass values into a method, out parameters to pass values out, and reference parameters to update or change the arguments
- As a general rule, if only one value is to be passed out, use a function method, not a void method with an out parameter

Aside – programming a formula

- Note that the expression

```
x1 = (-b+Math.Sqrt(b*b-4*a*c))/(2*a);
```

  says just what the formula says
- Don't become one of those programmers who break it up so that it bears no resemblance to the formula . . .

```
x3 = b*b;
x4 = 4*a;
x5 = x4*c;
x6 = x3-x5;
x7 = Math.Sqrt(x6);
x8 = x7-b;
x9 = 2*a;
x1 = x8/x9;
```

The traditional reference example

- Imagine we have two variables, *iBig* and *iSmall*, and we discover that *iBig* is smaller than *iSmall*
- We want to swap them
- We want to make many such swaps, so we write a method

```
private void Swap(ref int i1, ref int i2)
   {
       int iSwapper;
       iSwapper = i1;
       i1= i2;
       i2= iSwapper;
    }
```

- Examine this, and be sure you see how it works

Overloading

- That Swap method swaps two integers
- If we want a method that swaps two doubles, we can use the same name, Swap, and just define our parameters and our local variable as double
- C# doesn't mind that we have two methods of the same name – which is called *overloading* the name
- When we call Swap, it checks the signature of the call (its name *and* the number and types of its arguments) and finds the method with the same signature (its name and the number and types of its parameters)
- Lec6DemoMultipleFormsParameters illustrates everything covered in this lecture so far (and the rest)

Reference types and parameters

- There is an apparent exception to the rule that value parameters are simply initialised with the values of their corresponding arguments
- Remember, there are simple types, such as int and bool; and reference types, more complex types such as objects
- When a variable of a reference type is passed as a value parameter, what is passed is the reference to the object
- The parameter is then initialised to the value of that reference – so it becomes another reference to the same object
- This means that reference types are effectively passed by reference even when apparently passed by value

Good programs

- From a user perspective, good programs
  - satisfy a user's requirements
  - are easy/intuitive to use
  - are aesthetically pleasing
- From a programmer perspective, good programs
  - are readable
  - are reusable
  - are consistent
  - all of which make them easy to maintain

  (Maintenance is an important issue: programs may need to last a very long time and so may need adjustment to meet changing requirements)

Programming style

- Programming style is how to write good programs
  - It's not the same as how to write programs that work;
  - but it is easier to get programs working properly if they're written with good style
- We've been teaching good style from the start of the course. This is a quick revision of some aspects of good style – aspects that we hope you already apply
- A few further style pointers are thrown in for good measure
- Your programming assignment will be expected to display good programming style

Choosing names

- Intelligent naming should be applied to forms, controls, variables, properties, methods, classes, projects, C# files, etc
- That means an informative name
- Knowing the object type and giving the object a reasonable name can help us in our coding. For example TbxFamilyName tells me that I can use code like:

```
TbxFamilyName.Text = "Cornforth";
TbxFamilyName.ForeColor = Color.Red;
```

- `double a;` is not a well-named variable. Why not? (unless you're writing a solver for quadratic equations)

Renaming projects

- If you realise you've named a project poorly,
  - right-click on its name in *Solution Explorer*
  - select *Rename* and type the new name
- Then, to be really thorough . . .
  - in the project's properties, *Application* tab, change the name where it appears in *Assembly name* and *Root namespace*;
  - with the project closed, manually rename the top-level .sln and .suo files; but not the folder that's with them, or you'll corrupt the project
- Conclusion: name your project well at the beginning!

Indentation

- Program indentation shows structure: which statements are in the body of which statements
- Indentation is semi-automatic in C#; when it fails to keep up, re-align your code with Edit / Advanced / Format document, which has a shortcut of either ctrl-E + D or ctrl-K + D depending on your software version

```
private void BtnNestedFor_Click(object sender, EventArgs e)
{
    // Simple illustration of a loop within a loop
    int iFactorial;
    int iLimit = LimitingValue();
    TbxOutput.Clear();
    for (int i = 1; i <= iLimit; i++)
    {
        iFactorial = 1;
        for (int j = 1; j <= i; j++)
        {
            iFactorial = iFactorial * j;
        } // end for j
        TbxOutput.AppendText(String.Format("{0:d}! is {1:d}\r\n", i, iFactorial));
    }  // end for i
}  // end BtnNestedFor_Click
```

Keeping your code neat

- We've seen several tips for keeping your code neat . . .
- A blank line between methods helps us to see them more easily
- A blank line between distinct blocks of code within methods can be equally helpful
- Regions give an easy way to tuck code out of sight while we're not working on it – see ParameterDemo.cs in Lec6DemoMultipleForms
- *Always* delete accidental handlers, which appear when you double-click accidentally on the form or a control
- Be consistent in ordering the parts of your code; eg constants, then variables, then methods

Methods and functions

- A method should perform a distinct task
- If that task involves calculating a single value (numeric, string, object . . .), make it a (non-void) function
- Event-handlers perform the task of handling that event;
  - other methods perform what the programmer has decided is a single task
- If a method is too long, it can be hard to read
- Don't break it into parts *just* to shorten it,
  - but if it has obvious sub-tasks, consider making each sub-task a method in its own right

Methods replace duplicated code

- Whenever you find that you have the same code in two or more places . . .
- . . . you should put that code into a method, and replace the repeated code with calls to the method
- The program immediately becomes easier to read; also, if you now need to change anything in the code, you only need to change it in one place
- If the code has small variations, for example each instance uses different variables . . .
- . . . make these variables parameters to the method, and each time you call the method pass in whichever actual variables you're dealing with on this occasion

Comments

- You are not going to be the only person working on and looking at your code
- Others will be maintaining (and marking) your programs. Even *you* might have trouble reading them 6 months from now. So make them understandable!
- At the least
  - each file (each Class) should have header comments: name, date, history, purpose, special instructions
  - each method should have purpose comments
  - any complex (part of) algorithm should have explanatory comments about what it does and perhaps even the reason for including it

How much to comment?

- Some comments are patently unnecessary

```
iCounter = iCounter + 1;   // Add 1 to iCounter
```

- Comments (possibly quite brief) should be added to explain less obvious parts of the code

```
// Set up increasing powers of 2 in each bit
iBit0 = 1;
iBit1 = 2;
iBit2 = 4;
iBit3 = 8;
// etc
```


Sensible calculations

- When a value has to be calculated, it can be done in one expression

```
y = 5 * x * x - 3 * x + 2;
```

- or in multiple assignments

```
d1 = x * x;
d2 = 5 * d1;
d3 = 3 * x;
d4 = d2 - d3;
y = d4 + 2;
```

- As a general rule, break up a calculation *only* if it makes it clearer; normally you'll find it's clearer when left in its 'original' form.

Understanding booleans

- A programmer who understands booleans never compares them with true or false:
  - not `if (chbxLicensed.Checked == true)`
    but `if (chbxLicensed.Checked)`
  - not `while (bGameOver == false)`
    but `while (! bGameOver)`
- . . . and assigns them directly to boolean expressions:
  - not
```
if (iAge >= 18)
        bCanDrink = true;
    else
        bCanDrink = false;
```
    but `bCanDrink = iAge >= 18;`

(some programmers prefer `bCanDrink = (iAge >= 18);` for added clarity)

Nested ifs

- 'Nested ifs' generally means putting an *if* in the *then* part of an outer *if*
- (For some reason, the phrase isn't used for putting an *if* in the *else* part of an outer *if*, ie writing *else if*s)
- There is nothing fundamentally wrong with nested *if*s
- However, their logic can be confusing (especially for beginning programmers and, more importantly, for maintenance programmers), so use comments where appropriate to clarify them
- Alternatively, consider rewriting them using simpler logic flows (such as *else if*s) or more complex boolean expressions in a simple *if*

A couple more points for *if*

- Don't write an *if* statement with nothing after the condition and something after the *else*; use the opposite condition, put the body after the condition, and have no *else*
- Don't assign a variable the same value it already has (eg $x = x$); this is usually done because a programmer wants to write something for every possible case, but there is one case where nothing is to be done. So ... just leave out that case!
- If a statement has a really big *then* part (do all this stuff) and a really small *else* part (display this error message), use the opposite condition and reverse their order. That way, when you get to the *else*, you still remember what the condition was.

Documentation

- Documentation refers not only to comments in a program, but also to external descriptions such as program designs, user manuals, test schedules, etc
- Many programmers dislike writing documentation, but it is essential
- Fortunately, if you take the time to understand the problem, and actually design a solution, the program and the documentation both follow naturally
- Then the program can be maintained far more easily
- The longer a maintenance programmer takes to understand a program, the greater is the maintenance cost (in terms of time, effort and money)