Inft2012 Application Programming

Lecture 3

Methods, selection

Remove accidental handlers

- You've probably discovered that it's easy to accidentally create event handlers you didn't want, by double-clicking on the form or its controls in design mode
- Don't leave them there – remove them!

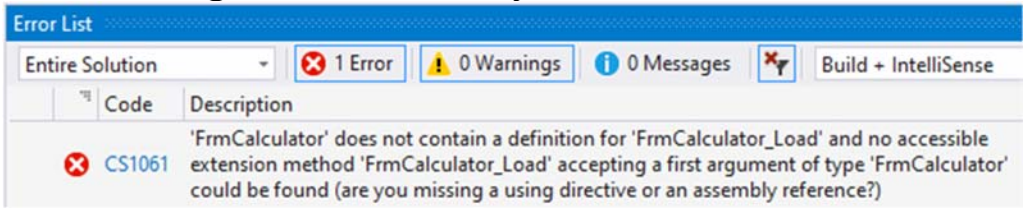
```
1 reference
private void FrmCalculator_Load(object sender, EventArgs e)
{

}

1 reference
private void LblFirst_Click(object sender, EventArgs e)
{

}

1 reference
private void TbxResult_TextChanged(object sender, EventArgs e)
{

}
```

Remove accidental handlers

- As soon as you've created an accidental handler, *go back to the form design* and press the Undo button
- The accidental handler will disappear
- If you press Undo *while you're still in the code window*, it will undo all of the changes since your last Save All. You don't want this!
- Fortunately, if you try it, you'll get a pretty clear warning. Heed it!

If you leave it till later . . .

- Removing accidental handlers by just selecting the code and deleting it can lead to syntax errors.



- If this happens, double click on the error; you'll be taken to the offending line of code, which will be in the designer file of code created by C# . . .

```
this.Load += new System.EventHandler(this.FrmCalculator_Load);
```

- delete that line of code, then close the designer file, because you don't normally go there

Renaming controls too late

- The same sort of error can arise if you start writing the event handler for a control, then go back and rename the control
- Alternatively, you might end up with an event handler that looks perfectly normal, but isn't executed when the event happens
- In either case, delete the offending line and the event handler . . .
- . . . then start the event handler again from the renamed control

Drawing a flag

- Let's draw a simple flag
- The Aboriginal flag is 3 units wide and 2 units high (whatever unit we choose to use)
- So what are its components, where are they located (from top left), and how big are they?
- The black rectangle?
- The red rectangle?
- The yellow circle?

  Remember, C# needs not its centre but the top left of its enclosing square.

From problem solution to code

- If we want a program to draw the flag, we have to work out these values (in units), then translate them into pixels
- There is *no point* in trying to write the program until we've done this!
- Programs should be designed and then written, not written and then aimlessly fiddled with
- Look at Lec3DemoFlags, the Draw Flag button
- Look at its code, and the rest of the visible code
- Notice that there are 3 'regions'
- Putting our code in regions can help when we want to work on one bit at a time and then tuck it away

Drawing many flags

- Clear the screen, click the first Flag Pattern button, then look at its code
- Even once the pattern had been designed (which of course was done before the code was written), how much effort went into writing this code and calculating all the numbers correctly?
- Now that the program is written, how much effort would it take to read and understand it?
- Yet if we want to change it (eg to change the location or size of one or more flags) we must understand it, to know which bits to change, and what to change them to

Drawing many flags

- Clear the screen, click the second Flag Pattern button, then look at its code
- Exactly the same amount of effort went into writing this code and calculating all the numbers correctly . . .
- . . . but it takes a lot less effort to read, and therefore to understand
- But look at all the 'almost-repetition', repetition with minor changes
- If only there were a draw-flag procedure/method; we could just specify the location and size, and let it do all the calculations
- OK – let's write one!

A method that doesn't handle events

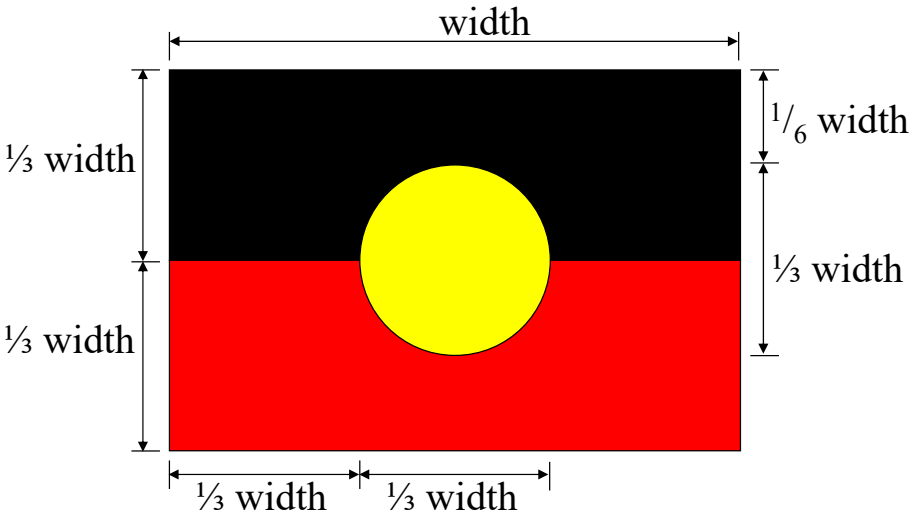
- All the methods we have written so far handle events, generally button clicks. It might not be obvious, but the last few bits of the first line tell you so.

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

- The method we want now will be called on by an event handler, but it isn't itself an event handler, so it won't have *object sender* and *EventArgs e* in its parentheses
- C# makes up reasonable names for our event handlers, so we generally accept them
- We have to make up our own names for methods that don't handle events. What about 'Flag'?

Flag design: pictorial version

- This diagram tells us all we need to know about the sizes of the flag's components . . .



Parameters in methods

- The quantities on the diagram are a lot easier to deal with if we have names for them . . .
- . . . so we specify *parameters* when writing the method
- A parameter is a marker that says "When this method is called, a value of this type will be provided; for the purposes of this code, we'll give it this name."
- We want our method to be given starting coordinates (horizontal and vertical) and a size, so we'll give it three integer parameters

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

- (We'll explain 'void' soon)

Arguments in method calls

- When we call a method (ie tell C# to do it), we provide arguments – the values that it needs to do the job
- There should be as many arguments in the call as parameters in the method definition; and the arguments must be of the same types as the parameters, in the same order as the parameters
- When we call

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

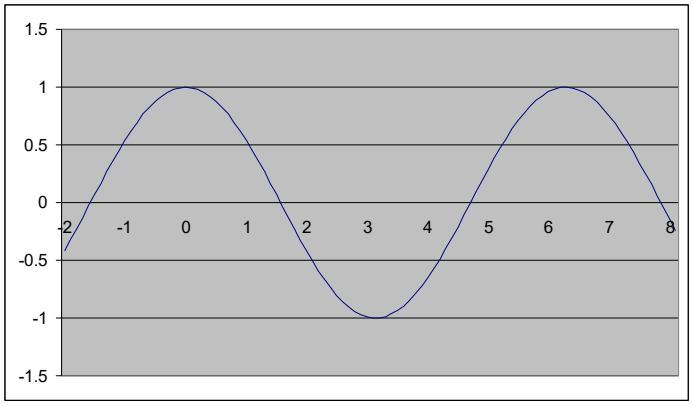
  C# executes the Flag method, with
  its first parameter (iLeft) taking the value 100,
  its second parameter (iTop) taking the value 90,
  and its third parameter (iSize) taking the value 180
- Now look at the code for the third flag pattern button

Advantages of using methods

- Some of the advantages of using methods . . .
- The amount of code is reduced dramatically, with each working section of code replaced by a single call
- The method is more general, so we're more likely to comment it helpfully
- The method calls are virtually self-documenting – they almost explain themselves
- The programmer only has to know the start coordinates and size of every flag – the method calculates the various numbers needed for each rectangle and the circle
- Conclusion – methods can be extremely useful!

The idea of a function

- A function in maths can be thought of as a graph. Here is the *cosine* function, abbreviated *cos*:



- When we apply the function to a specific value, we get a specific value: cos(0) is 1, cos (π/2) is 0, etc
- This is also how we think of a function in programming: something that, given a value, returns another value

Functions in programming

- A function in a program is a method that takes one or more values and returns a value calculated using those values
- As with our flag method, the starting values are provided as arguments
- A function method is called by writing the function name, then its arguments in parentheses, exactly where we want to use the value
- The method calls we've seen so far (eg `Flag(100,90,120);`) are whole statements; a function method call is part of a statement . . .

```
y = 3 * cos(x) - 5 * sin(x * x);
```

In C# they're not called functions

- In fact we would never write cos and sin functions – along with many others, they are already provided
- But C# doesn't have every function we might want

- And by the way, in C# they're not called functions, they're simply another sort of method
- In some other languages, such as Python, they are called functions
- As a compromise, we'll generally call them function methods, to show that we're aware of the difference between these and other methods

A function for the area of a triangle

- If we know the lengths (a, b, c) of the three sides of a triangle, the area of the triangle is

$$A = \sqrt{s\,(s-a)\,(s-b)\,(s-c)}$$

$$\textit{where} \quad s = \frac{1}{2}(a+b+c)$$

- You're not expected to know or learn this formula – just to see what we do with it
- Let's write a TriangleArea function method, which takes the three lengths as its arguments and returns the area

An extra variable

$$A = \sqrt{s\,(s-a)\,(s-b)\,(s-c)}$$
$$\text{where} \quad s = \frac{1}{2}(a+b+c)$$

- Notice *s*
- It doesn't have to be there, but it makes the top expression much shorter
- The same applies in programming: if we calculate it once, calling it *s*, then use *s* four times in calculating the area . . .
- . . . it's easier and less messy than calculating it four times in the expression for the area
- So in addition to the three lengths (which will be parameters), our function requires a variable *s*

A local variable

- This extra variable is required only in the TriangleArea function method; it's no use anywhere else
- We therefore declare it within the method; it is a *local* variable that exists only in this method

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

- Note the return statement. This is very important: it's how the method 'sends back' the calculated value
- The rest will be explained over the next few slides

The scope of a name

- We make up names for our variables, our methods, our classes, and more
- The *scope* of a name describes where the name is valid, where the variable, method, etc, can be used
- If a variable is declared at the start of a class, its scope is the whole class; it's called an *instance* variable
- Likewise, a method in a class can be called from anywhere within the class
- On the other hand, if a variable is declared within a method, its scope is just the method – when that method isn't running, it doesn't exist
- A parameter's scope is also the method it is in

Holes in scope

- What if a local variable or a parameter has the same name as a variable declared at the class level?
- The outside variable suffers a 'hole' in its scope
- That is, it cannot be accessed while we are in the method
- Any use of the name within the method refers to the local variable or the parameter
- Once the method ends, its local variables and parameters cease to exist, and any outside variables of the same name can be used again
- However, to avoid confusion, it's generally best not to use the same name in different contexts

A function must have a type

- Our triangle area function method would be called in a statement such as

```
dTotalArea = dTotalArea + TriangleArea(35,50,24.7);
```

- Because we use the function call where we would use the value it produces, the function must have a type – the same type as that value
- Its type must also be the type of the expression that follows the word "return" in the function
- In this case, the type is double
- When defining a function, we must specify not only the type(s) of its parameter(s) but also the type of the function itself

```
private double TriangleArea(double a, double b,
  double c)
```


What does void mean?

- It's time to find out what void means
- We've just said that every function method must have a type
- In fact every *method* must have a type, simply because all methods are declared in the same way
- But methods like Flag don't need a type like int or double, because they don't calculate and return a value
- void is the 'type' that we give to non-function methods, methods that don't need a type and don't return a value

Functions vs void methods

- Summarising, all methods have types, but there's a big difference between the ones whose type is void and the ones whose type is anything else (which we're calling function methods or simply functions)
- A function method has a return statement, which returns a value of the same type as the method
- A void method is called as a complete statement, eg
  `Flag(100,90,120);`
- A function method is called within a statement, at the point where you want to use the returned value, eg
  `dTotalArea = dTotalArea + TriangleArea(35,50,24.7);`

Meaningful variable names?

- We said earlier that we expect you to use meaningful variable names with appropriate prefixes
- Single-letter names can be meaningful, if they are exactly the name used for the quantity in 'the real world' outside the program
- This formula for the area of a triangle is always expressed in terms of a, b, c, and s
- To express it otherwise (eg dSide1, dSide2, dSide3, dHalfPerimeter) would actually make the program less clear
- As the point of longer names is to make the program clearer, we don't use them in these circumstances

A string function

- Functions don't have to be mathematical

```
private string FullCourseName(string
     sPrefix, int iCode, string sName)
 {
   string sCode = Convert.ToString(iCode);
   return sPrefix + sCode + " " + sName;
 }
```

- So, for example,

```
TbxCode.Text = FullCourseName("Inft",
       2012, "Application Programming");
```

would display

```
Inft2012 Application Programming
```

in the TbxCode text box

this

- We've said that to call a method of an object we write ObjectName.MethodName(arguments)
- The procedures/methods covered in this lecture are all methods of the form that is the basis of our program
- Within the code for a class of object, we don't need to specify the object or class – we can just write the method name, as we have been doing
- If we wish, though, we can write this.MethodName, eg

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

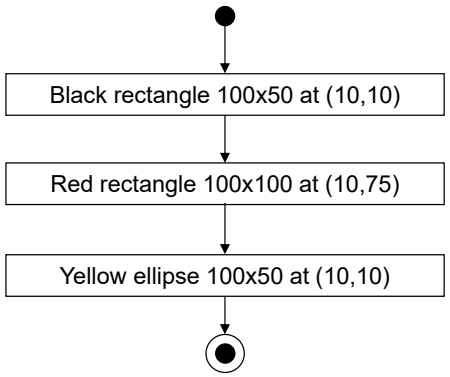
- This is not at all necessary, but some programmers like to do it for the sake of completeness – and we'll see it in the automatically generated code

Sequence, selection, iteration

- Sequence, selection, and iteration are the building blocks of the sort of programming we're covering in this course
- When a method is executed, it goes through each of its statements in turn, ignoring any comments and blank lines
- *Sequence* – doing statements in the order they appear – is the simplest order of operations in a program

Flowchart showing sequence

- We sometimes use 'flowcharts' to show the flow of a piece of code
- Execution starts at the top of the diagram and follows the arrows through each statement in turn, ending at the bottom
- The statements don't have to be correct C# code



Selection

- In programming, *selection* is choosing which statements to execute
- Instead of executing all the statements in a method, from start to finish, we reach a point where we want to do one set of statements in some circumstances and possibly a different set in other circumstances
- This means that we need a way of programming "in some circumstances" and "in other circumstances"
- For this we need the data type *bool*
- It's an abbreviation of boolean, named after George Boole, who formalised its use in 1848

Boolean data type

- The bool data type is the simplest data type in programming
- The int data type, which is pretty simple, has 4,294,967,296 possible values, ranging from
  – *2,147,483,648* to *2,147,483,647*
- The bool data type has only two possible values: *true* and *false*
- Boolean data was designed for working with formal logic, but in programming it's astonishingly useful in selection and in some other constructs

Boolean variables

- Boolean variables tend to represent things that we generally think of as being true or false:
  - underage; licensed; overweight; game over
- Simple boolean assignment statements . . .

  bLicensed = true;

  bGameOver = false;

If statement

- The simplest selection statement is the *if* statement:

```
if (<boolean expression>)
  { <statements to be executed if true> }
else
  { <statements to be executed if false> }
```

- The angle brackets are not part of programming; they are used by us to describe parts of a program statement.
  - <boolean expression> means "something that has a boolean value", such as a boolean variable
  - <statements> means "some program statements"
- The braces (curly brackets) are normally on lines of their own, but it's hard to fit that on this slide

Example of if statement

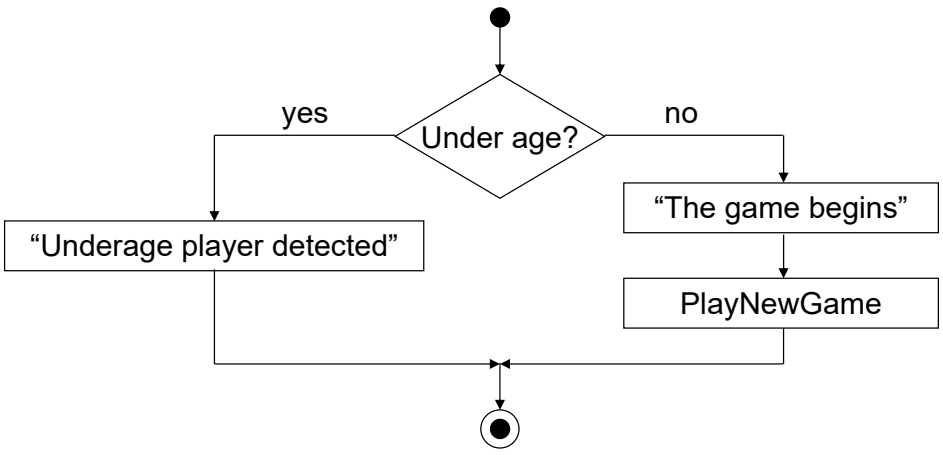
- A simple if statement in a method might look like this:

```
if (bUnderage)
  {
   MessageBox.Show("Too young to play",
      "Underage player detected");
  }
else
  {
   MessageBox.Show("Prepare to lose!",
      "The game begins");
   PlayNewGame();
  }
```

- If the value of bUnderage is true, the first statement is executed; if it's false, the alternative two statements are executed

Flowchart for if statement

- The boolean test is in a diamond-shaped box
- Its branches are clearly labelled *yes* and *no* or *T* and *F*
- Immediately after the branches, control returns to a single flow.



Boolean expressions

- An expression is a set of operators and operands that is evaluated to give a value:

  5 * 3 + 27 evaluates to give the value 42
- A boolean expression is an expression whose value is either true or false:

```
bUnderage
iAge > 30
TbxEmail.Text != ""
TbxEmail.Text == "simon@newcastle.edu.au"
```

- Each of these expressions evaluates to either true or false. The equality test doubles the equals symbol to avoid confusion with the assignment statement

Comparison operators

- Boolean expressions often use the comparison operators (<, >, ==, <=, >=, !=). The program tests the left side of the operator against the right side and checks whether the outcome is true or false
- Used between numbers (generally of the same type), the meanings are pretty obvious
- Strings can be tested for equality (==) and inequality (!=), but not with the other comparison operators
- Checking if one string comes alphabetically before another requires a method that we'll meet later

Ifs with comparisons

- If statements often use boolean expressions built with comparison operators

```
if (iShoeSize > 13)
   { MessageBox.Show("No boots big enough",
             "Error: cannot ski")}
else
  {
   MessageBox.Show("Boot hire $15/day",
         "Equipment issue complete");
   IssueEquipment();
   CollectPayment();
  }
```


Don't compare booleans

- It's possible to compare booleans

  if (bGameOver == true) . . .
- But this is silly, because bGameOver == true always has exactly the same value as bGameOver!

boolean expression	*value if bGameOver is true*	*value if it isn't*
bGameOver	true	false
bGameOver == true	true	false

- It is very poor programming style to check
  if *(something == true)* (just check if *(something)*) or
  if *(something == false)* (just check if *(!something)*)

If statements with else if

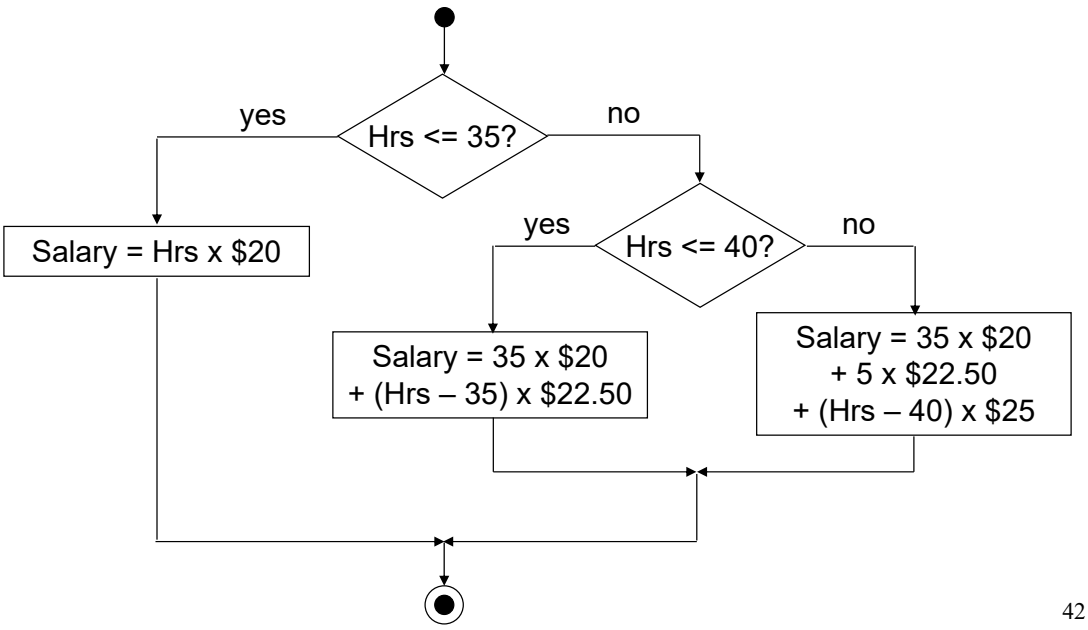
- This code uses a given formula to work out the week's payment for a worker based on the hours they worked . . .

```
if (iHoursWorked <= 35)
   {dPayment = iHoursWorked*20;}
else if (iHoursWorked <= 40)
   {dPayment = (35*20)+(iHoursWorked-35)*22.5;}
else
   {dPayment = (35*20)+(5*22.5)+(iHoursWorked–40)*25;}
```

- You should desk check this code to be sure that you understand which formula will be used for what ranges of hours worked
- You should also be clear that each time this statement is executed, it will apply only one formula

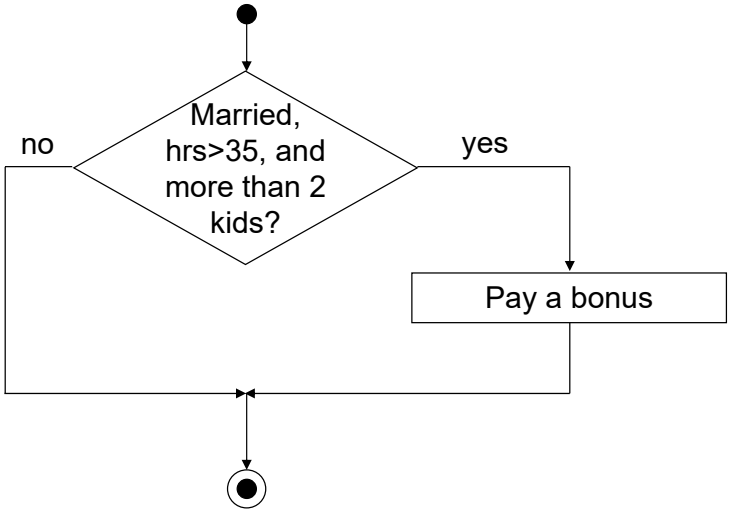
Flowchart for if . . . else if

- Notice how the else if fits within the if as another if to be done if the answer to the first test is no



Flowchart for if with no else

- Notice that an if statement doesn't need an else part
- We show the no branch even though it has no action



Boolean operators

- +, *, % etc are arithmetic operators – used between two arithmetic operands, they form an arithmetic expression that can be evaluated
- && (and) is a boolean operator – between two boolean operands, it forms a boolean expression that can be evaluated (and is true if both of the operands are true)
- || (or) is also a boolean operator – between two boolean operands, it forms a boolean expression that can be evaluated (and is true if either of the operands is true)
- ! (not) is a unary boolean operator – used before a boolean operand, it produces the opposite boolean value (ie ! true is false and ! false is true)

Boolean operators, boolean operands

- The way we sometimes use *and* and *or* in English, we might be tempted to use them between arithmetic operands . . .

  if (iAge == 18 || 21) . . .

- This won't work because 18 and 21 are numbers, not booleans. Instead we need

  if (iAge == 18 || iAge == 21) . . .

- Always be sure that &&, ||, and ! have boolean operands; that is, the bit that follows ! must be boolean, and the bits before and after && and || must both be boolean

Input validation

- It's always a good idea to check whether user input is valid before using it in calculations
- If we know that the minimum weekly work is 4 hours and the maximum is 60, we can accommodate that
- As always, we should desk check to be sure that the tests in an if statement work correctly together . . .

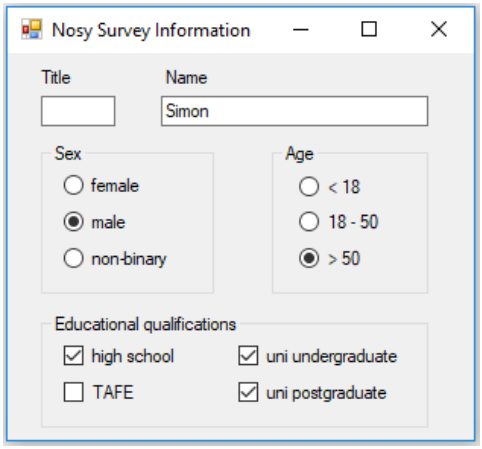
```
if (iHoursWorked < 4 || iHoursWorked > 60)
  {MessageBox.Show("Hours not in permitted range");}
else if (iHoursWorked > 40)
  {dSalary = (35*20)+(5*22.5)+(iHoursWorked–40)*25;}
else if (iHoursWorked > 35)
  {dSalary = (35*20)+(iHoursWorked-35)*22.5;}
else
  {dSalary = iHoursWorked*20;}
```

All those braces!

- By now our programs have multiple methods, sometimes with ifs within them
- At some points, the program is just a sequence of closing braces
- It's a good idea to put a comment with each closing brace, to help you keep track of which is which
- By the way, the braces in a selection statement aren't always necessary; they're needed if the body is more than one statement, but not if it's a single statement
- It's best to use them all the time, though: it can be confusing if you remove them for a single statement, then later add more statements but forget the braces

Grouping controls – group box

- The GroupBox control (from *Containers* in the toolbox) acts as a container for other controls such as radio buttons and check boxes (see next slides)
- GroupBox is one example of a class called *Container*; it's probably the only one we'll use



Uses of group box

- Group boxes are clearly useful for showing the user that certain controls are related, and for giving them a name (eg, Sex, Age, Educational qualifications)
- This would be a good enough reason to use group boxes, helping to make a form easier for the user to understand
- However, they do have another important use, which we'll see when we consider radio buttons

Checkboxes

- The checkbox is a simple control that the user can tick or untick by clicking on it
- Its Text property is a message that appears alongside it
- Its Checked property is a boolean that is typically used in if statements . . .

```
if (ChbxAnchovy.Checked)
  {
  // Add anchovy to this pizza
  }
if (! ChbxCheese.Checked)
  {
  // Don't put cheese on this pizza
  }
```

Radio buttons

- Radio buttons are another select/deselect control . . .
- . . . but if there are several in a GroupBox or other container, only one can be selected at any one time
- Selecting a radio button in a container automatically deselects any other radio button that was selected

```
if (RbtnSmall.Checked)
  { /* Give this pizza a small crust */ }
else if (RbtnMedium.Checked)
  { /* Give this pizza a medium crust */ }
else
  { /* Give this pizza a big crust */ }
```

- Because only one can be selected, it makes sense to use else-ifs to find out which one
- Note the new form of comment; why was it used here?

The *CheckedChanged* event

- The *CheckedChanged* event of a radio button can be used to take some action whenever the radio button is selected or deselected
- However, if you write a simple program to, say, display an appropriate message box when a radio button is clicked, you will note that you generally get two messages for each click – because selecting one radio button deselects another, so they both change!
- When using *CheckedChanged* as an event handler, it might be wise to add in a test to see if the radio button is currently *Checked*, and do nothing if it isn't. That way, you only get the one that has just been selected.

Reminder – understanding booleans

- Many programmers (and some book authors) write

```
if (RbtnChoice1.Checked == true) . . .
if (RbtnChoice4.Checked == true) . . .
```

- This shows us that they don't seem to understand that

```
if (RbtnChoice1.Checked == true) . . .
```

means exactly the same as

```
if (RbtnChoice1.Checked) . . .
```

- Why stop at one? They could write

```
if (RbtnChoice1.Checked == true == true == true)...
```

and it would still mean exactly the same

And assigning booleans

- Also, you very seldom need an *if* statement to assign a boolean variable
- You might be tempted to write

```
if (iAge < 18)
{
    bUnderage = true;
}
else
{
    bUnderage = false;
}
```

- But it's much more logical to write

```
bUnderage = iAge < 18;
```