C# for Beginners

Windows Desktop Applications using Windows Forms

.NET

Contents:
- Windows Forms and Controls
- Event-driven programming
- Properties, Events and Event Handlers

Farid Naisan, University Lecturer, Malmö University, farid.naisan@mau.se

Windows Applications

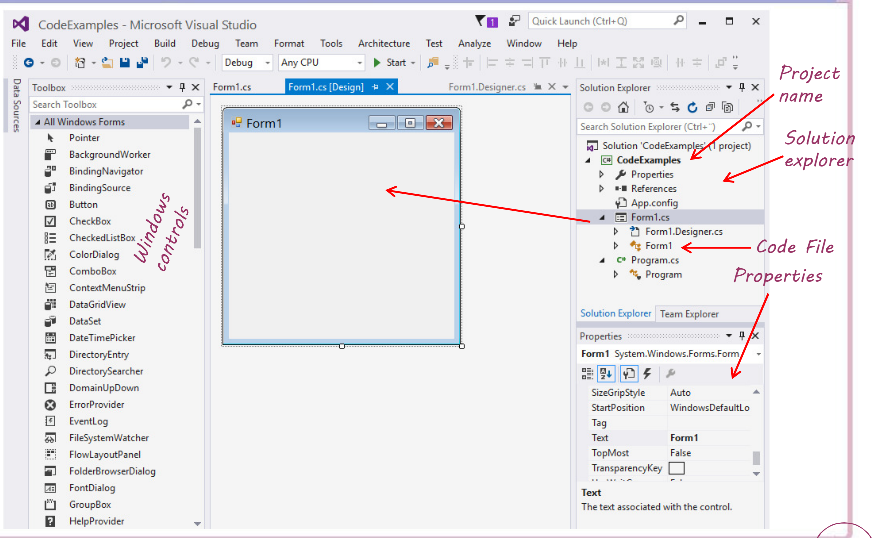
.NET

- .NET includes all the classes and features needed to develop desktop applications.
- It contains many namespace to categorize related classes and types.
- System.Windows.Forms is a namespace hosting a large selection of types for designing and writing applications with graphical user interface (GUI).
- The GUI components are grouped as **Forms** and **Controls**. They are all windows based and have a parent class called System.Windows.Forms.Control.
- A Control is a graphical component with many features.
  - TextBox, Label, Button, Icon are all controls.

Windows Forms

- A Form is a class and we need to create an object of this class before we can access its feature-rich members.
- You can think of a form as a container for other controls, although you write text or draw graphics directly on its surface.
  - However, there are other components that are particularly designed for such purposes.
- When you create a Windows Forms Application in VS, an object of the Form class is created by VS with the default name “Form1”, ready for you to begin designing your GUI.

An Example Project

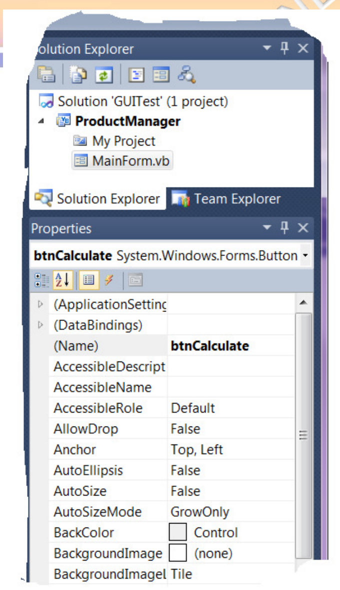


Windows Forms – starting object

- When working with GUI applications using Visual Studio, Visual Studio creates automatically a start up class with a Main method (named Program.cs by default).
- In this method, it creates Form1 which becomes the staring object meaning that when you run the application, Form1 will be loaded and displayed to the user.
- Every application usually has multiple Forms. One of these has to be selected as the start up Window. If you wish to select another Form as your starting object open the folder Properties inside VS, in the Project window.
- Designing a Windows Forms Application is very easy using Visual Studio, but it can be developed using an ordinary text editor like Notepad and of course the .NET SDK.

Working with Forms and Controls

- All Forms and Controls are instances of classes
- They have:
  - Fields
  - Properties and methods
  - Events
  - Event-handlers
- An event is an action that occurs during the program usage at run time.
  - Clicking on a button fires the Click

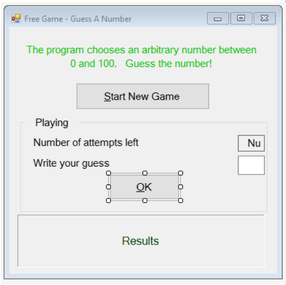


btnCalculate System.Windows.Forms.Button

(ApplicationSettinc	
(DataBindings)	
(Name)	btnCalculate
AccessibleDescript	
AccessibleName	
AccessibleRole	Default
AllowDrop	False
Anchor	Top, Left
AutoEllipsis	False
AutoSize	False
AutoSizeMode	GrowOnly
BackColor	☐ Control
BackgroundImage	☐ (none)
BackgroundImageL	Tile

Event-handler method

- An event-handler is a method that is connected to an event inside the code and is automatically executed when the event occurs at run time.
- The **btnOK_Click** method below is an example of such a method. It is connected to the OK button. btnOK is the Name-property of the button

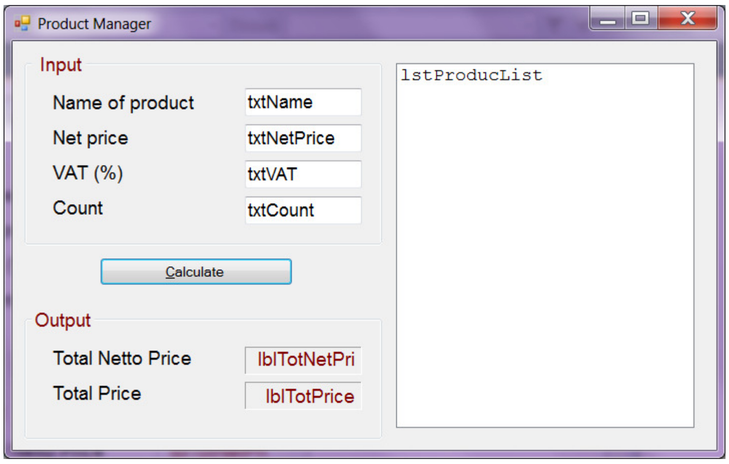

Free Game - Guess A Number

The program chooses an arbitrary number between 0 and 100.   Guess the number!

```
//Every time the user clicks the OK button, this method gets executed.
//Sender: is the object that notifies about the clicking, here it is
//the btnOK button.  The parmeter e is a structure that contains
//information about the clicking (whether right or left button is used)
1 reference
private void btnOk_Click (object sender, EventArgs e)
{
    //All code here will be executed automatically.
}
```

GUI Example – The Product Manager

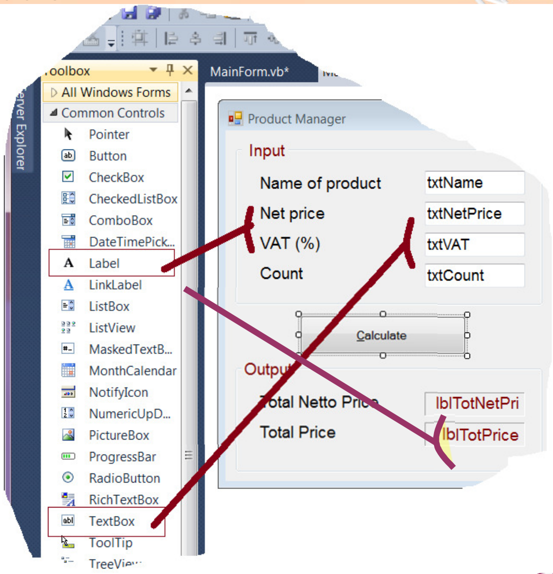
- Same naming rules as variables apply also for controls.


Product Manager

Input	
Name of product	txtName
Net price	txtNetPrice
VAT (%)	txtVAT
Count	txtCount
Output	
Total Netto Price	lblTotNetPri
Total Price	lblTotPrice

Textboxes and labels

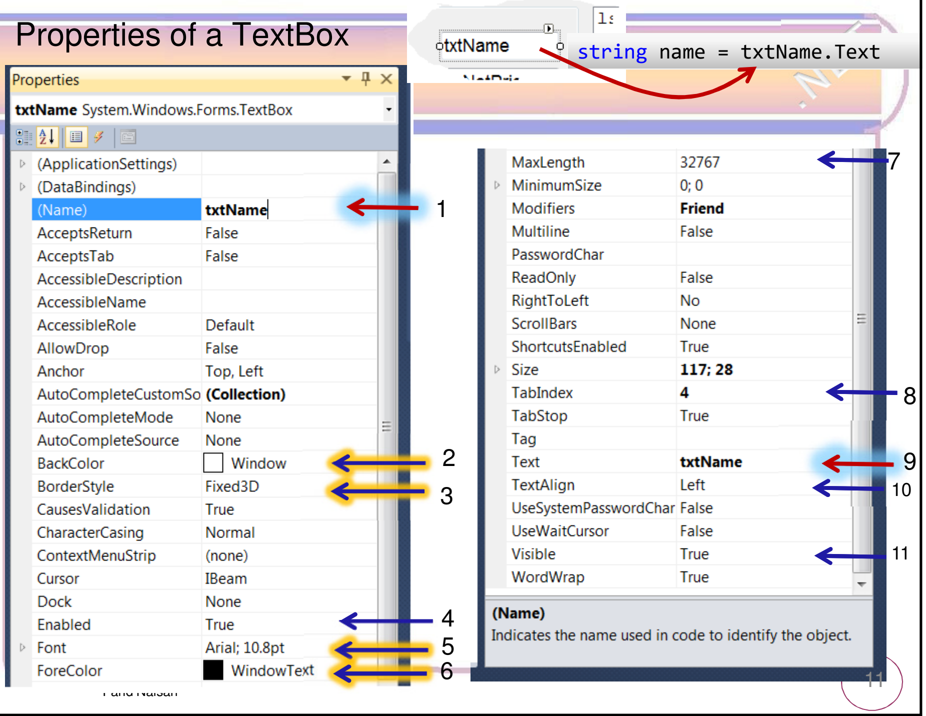
- TextBoxes are used for editable text (input)
- Labels are used for read-only text (heading, info, output).



Properties

- Every Control has many properties that you can affect the look and feel of the control..
- Some of the properties can be changed only at design time, some only at run-time (from code) and some either way.
- Certain properties are not possible to change. These are read-only.
- Each property has a value of a certain type. The **Text** property (ex textBox1.Text) for example is a string, while the **Enabled** and **Visible** properties are Booleans.
- Some properties have values that are constants and should be chosen from an Enumeration.

Properties of a TextBox



```
string name = txtName.Text
```

Properties

txtName System.Windows.Forms.TextBox

(ApplicationSettings)	
(DataBindings)	
(Name)	txtName
AcceptsReturn	False
AcceptsTab	False
AccessibleDescription	
AccessibleName	
AccessibleRole	Default
AllowDrop	False
Anchor	Top, Left
AutoCompleteCustomSo	(Collection)
AutoCompleteMode	None
AutoCompleteSource	None
BackColor	⬜ Window
BorderStyle	Fixed3D
CausesValidation	True
CharacterCasing	Normal
ContextMenuStrip	(none)
Cursor	IBeam
Dock	None
Enabled	True
Font	Arial; 10.8pt
ForeColor	⬛ WindowText
MaxLength	32767
MinimumSize	0; 0
Modifiers	Friend
Multiline	False
PasswordChar	
ReadOnly	False
RightToLeft	No
ScrollBars	None
ShortcutsEnabled	True
Size	117; 28
TabIndex	4
TabStop	True
Tag	
Text	txtName
TextAlign	Left
UseSystemPasswordChar	False
UseWaitCursor	False
Visible	True
WordWrap	True

(Name)
Indicates the name used in code to identify the object.

Methods

- Forms and Controls have also many useful methods that you can use in your code.
- For example some controls have a method called **Focus** which means that it is ready to receive the user's input.
- The focus can be set to a control in code using the Focus method:

```
      txtNetPrice.Focus();  //method call
```

- This method sets the focus to the textbox named txtNetPrice.
- Note:
  - Properties are used exactly as variables.
  - txtProductName.Text = “Egg”; //no parenthesis after .Text
  - Methods are used as ordinary method calls, as above.

Event-Driven Programming

- An event is an action that takes place within a program, such as the clicking of a button.
- An Event is an action performed by the user interacting with an application, or by code in your application.
- Applications respond to an Event by providing methods that are automatically called.
- Events and event-handlers are the key concepts in the event-driven mechanism.
- An event-handler method is connected to an event easily using VS.

Events and event-handler methods

- All Visual C# .NET controls are capable of detecting a set of predefined events.
- You can handle an event by writing code in a special method connected to a control.
- To handle an event means that you will write code that will instruct the program what actions to take whenever a specific event is triggered.

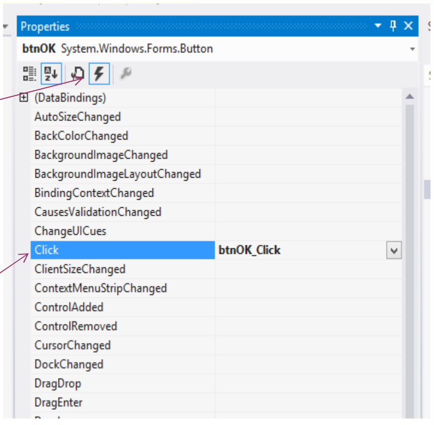
```
//Event-handler method connected to the Click event
//of the button btnOK.
//Sender: the object sending the event
//e: object containing relevant info about the event.
private void btnOK_Click(object sender, EventArgs e)
{
    //Write code to do things when the OK button is clicked.
}
```

Events and event handlers

- Forms and Controls decide which event they make available.
- Events have names. The click-event of a control has the name **Click**.
- Other examples are
  - **Load, Unload, MouseDown, MouseUp, MouseMove, MouseDoubleClick, KeyPress, KeyDown, KeyUp, SizeChanged**, etc.
- Each event has also a set of parameters that are made available to the event-handler, as in the previous slide.

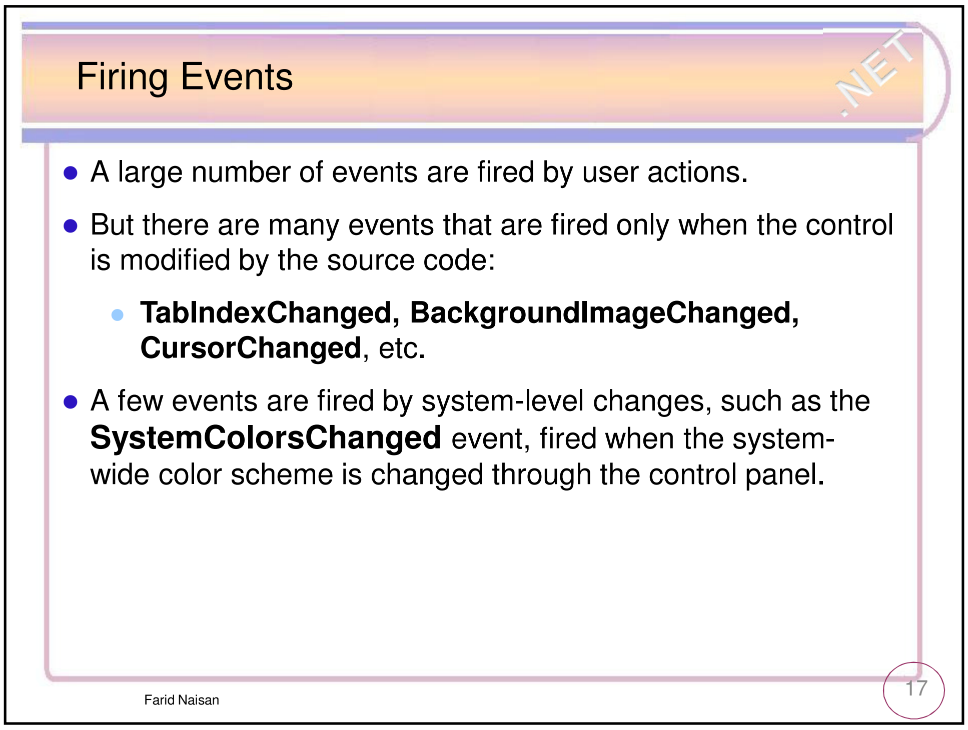
Example – Button Events

- To see all events for a certain control, select the control, click on the lightning icon.
- To create an event-handler method for a control, double-click on the event.
- The figure shows the Click-event for a button.



btnOK System.Windows.Forms.Button

(DataBindings)	
AutoSizeChanged	
BackColorChanged	
BackgroundImageChanged	
BackgroundImageLayoutChanged	
BindingContextChanged	
CausesValidationChanged	
ChangeUICues	
Click	btnOK_Click
ClientSizeChanged	
ContextMenuStripChanged	
ControlAdded	
ControlRemoved	
CursorChanged	
DockChanged	
DragDrop	
DragEnter	

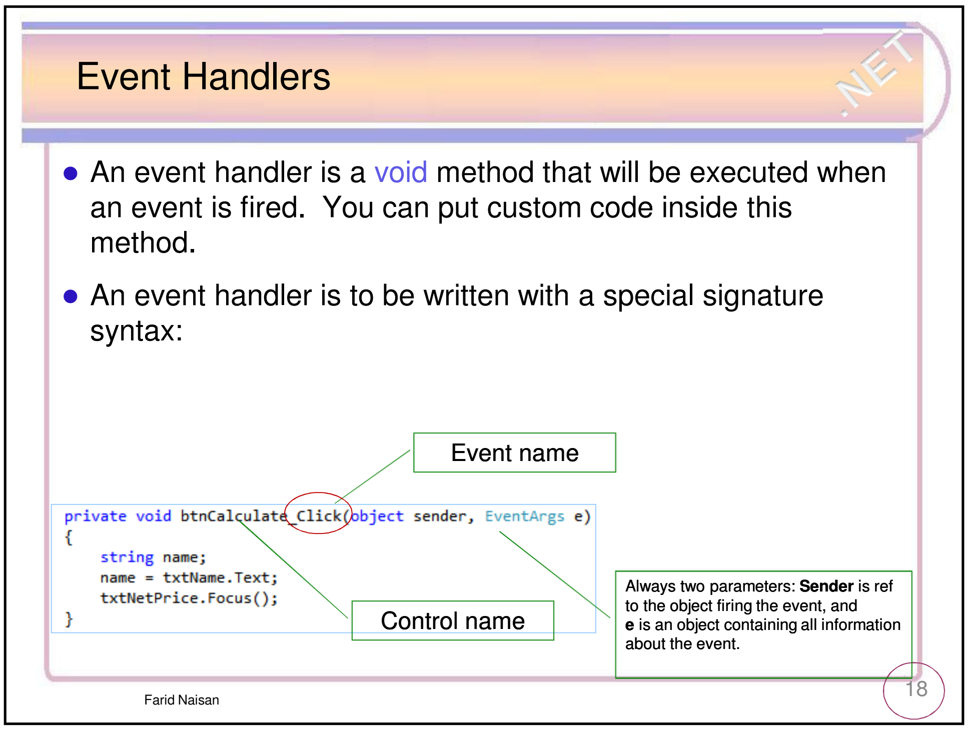


Firing Events

- A large number of events are fired by user actions.
- But there are many events that are fired only when the control is modified by the source code:
    - **TabIndexChanged, BackgroundImageChanged, CursorChanged**, etc.
- A few events are fired by system-level changes, such as the **SystemColorsChanged** event, fired when the system-wide color scheme is changed through the control panel.

Farid Naisan

17



Event Handlers

- An event handler is a void method that will be executed when an event is fired. You can put custom code inside this method.
- An event handler is to be written with a special signature syntax:

Event name

```
private void btnCalculate_Click(object sender, EventArgs e)
{
    string name;
    name = txtName.Text;
    txtNetPrice.Focus();
}
```

Control name

Always two parameters: **Sender** is ref to the object firing the event, and **e** is an object containing all information about the event.

Farid Naisan

18

Event Delegates

- A delegate is a type in .NET that can store the address of one method or a number of methods in the same or different objects.
- Delegates are used very frequently by Windows Forms.
- VS takes care of all the necessary coding to let a delegate remember for example when to invoke a handler method when an event is triggered (clicking on a button).
- All event-handler methods are invoked by delegates.
- VS uses delegates to attach an event-handler to an event of a control.
- The delegate remembers the method and calls it when the attached event is fired.

Use of Delegate - example

- The code in the figure is generated by VS when you double-click on the button btnOK at design time.

```
this.btnOK.Click += new System.EventHandler(this.btnOK_Click);
```

- The method **btnOK_Click** of the **MainForm** will be automatically invoked when the **btnOK** fires the **Click** event, when the user clicks on the button.
- The **+=** symbol adds the method in the list of methods that their objects want them to be executed when the Click event is fired.
- This way objects ”subscribe” to an event of another object.

Events in The Life Cycle of a Form

Event	Occurs …
Move	when the form is being created but before the Load event.
Load	when the Form is created but before it is displayed for the first time.
VisibleChanged	the value of the Visible property changes.
Activated	when the form is activated by the source code or by the user.
Shown	when the Form is displayed for the first time.
Paint	when the control is redrawn.
Deactivated	when the form loses focus and is not the active form.
Closing	just before the form is closed.
Closed	when the form is being closed.

Summary

- A Windows Form is used to design a Graphical User Interface (GUI).
- There are numerous controls under the System.Windows.Forms namespace available for a programmer.
- These are shown in the ToolBox in VS.
- Write event-handler methods to your controls to perform tasks.
- A handler method connected to an event of a control will automatically be executed when the event is fired by the control.
  - The event is fired as a result of user interaction or if the event is triggered from somewhere in the code.
  - When you click on a button on the form, the Click event will be fired. If you have written code in the buttons Click-event handler method, that code will be executed then.

C# For Beginners

Enumerations

Contents:

- The type enum
- The object Enum (with capital E)
- When to use enum

Farid Naisan, Univerisity Lecturer, Malmö University, farid.naisan@mau.se

"enum" is a type

- The type "enum" (with a lowercase 'e') is one of the five core C# types:
  - **class, interface, delegate, struct, enum**
- .NET Framework has hundreds of predefined enums
  - DaysOfWeek, DialogResult, etc are examples.
- Enums are used to group a set of named constants that are related to each other
  - Days of a week, countries of the world.

Example

- An enum is defined in much the same way as a class.
  - An access modifier
  - The keyword enum
  - An enum name
  - Members are separated by a comma
- Comparing to a class definition, the keyword class is replaced by the keyword enum.
- An enum cannot have fields or methods.
  - It holds only a group of constant names with an underlying value.

```
public enum CardSuits
{
    Club,       //=0
    Spade,      //=1
    Diamond,    //=2
    Heart       //=3
}
```

```
public class Car
{
}
```

Member data type and initialization

- The default underlying type for the members is **int** (system.Int32)
- Other integral types can also be used.
  - byte, int, long,uint, ulong,etc

```
public enum SomeBigStuff : long
{


}
```

```
public enum CharCards
{
    Ace  = 1,
    Jack = 10,
    Queen,     // = 11
    King      // = 12
}
```

```
public enum Countries
{
    Afghanistan = 93,
    Sweden = 46,
    UnitedKingdom = 44,
    UnitedStatesOfAmerica = 1
}
```

Declaring and using enums

- A variable of an **enum** type can be declared exactly in the same way as you declare other value types, such int and double.

  private **CharCards** fortuneCard = **CharCards.Ace**;

- enums can be nested inside a class as a part of the class when it relates to the class.
- But if an enum is common between different classes, it should be declared public or internal just as a class.
  - Save the enum in its own code file no matter the size.

Farid Naisan, Malmö University

5

Conversion between enum and integral types

- An enum value can be converted to its corresponding underlying value and vice-versa.
- Different ways but using type-casting is simple.
- To convert an int variable to its corresponding enum member:

```
Countries country = (Countries)cmbCountry.SelectedIndex;//int to enum

double price = GetCallRatePerMinuteFor(country);
```

Note·countries is an enum,
electedIndex is an integer

- To convert an enum variable to the number that it represents:

```
cmbCountry.SelectedIndex = (int)country;          //enum to int
```

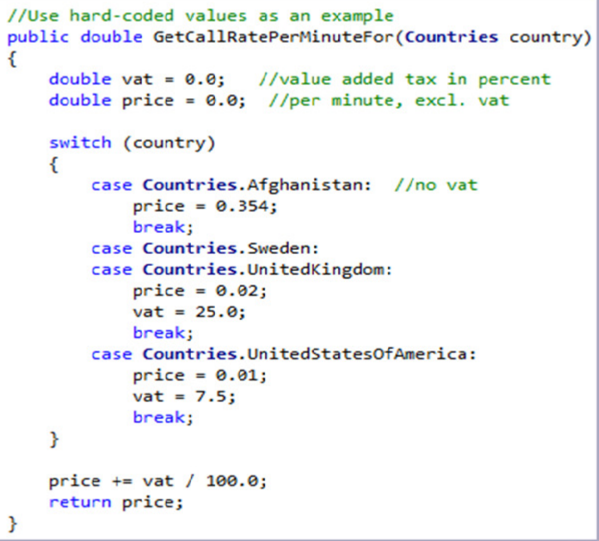
Farid Naisan, Malmö University

6

Use of enum variables

An effective way of using enums in switch statements!

Make a note of how the enum-members are used in code (Countries.xxx)



```
//Use hard-coded values as an example
public double GetCallRatePerMinuteFor(Countries country)
{
    double vat = 0.0;   //value added tax in percent
    double price = 0.0;  //per minute, excl. vat

    switch (country)
    {
        case Countries.Afghanistan:  //no vat
            price = 0.354;
            break;
        case Countries.Sweden:
        case Countries.UnitedKingdom:
            price = 0.02;
            vat = 25.0;
            break;
        case Countries.UnitedStatesOfAmerica:
            price = 0.01;
            vat = 7.5;
            break;
    }

    price += vat / 100.0;
    return price;
}
```

The object Enum

- .NET Framework provides a ready-to-use object called **Enum (capital E)** that can efficiently be used together with enums to manipulate its values.
- The **Enum** object has several useful methods:
  - GetName
  - GetNames
  - GetValue
  - GetValues
- These methods can be specially useful when working with GUI components.

Example with a combobox

- In the code example here, **cmbNumber** and **cmbSuit** are both **ComboBoxes** (Windows Forms controls), while **CardSuits** is an enum type.

```
private void InitializeGUI()
{
    cmbSuit.DataSource = Enum.GetValues(typeof(CardSuits));
    cmbSuit.SelectedIndex = (int)CardSuits.Heart;

    for (int number = 0; number < 10; number++)
    {
        string strNo = string.Format("{0,2}", number + 1);
        cmbNumber.Items.Add(strNo);
    }
    cmbNumber.Items.AddRange(Enum.GetNames(typeof(CharCards)));
}
```

Summary

- enum is one of the five types that the whole C# language is built on.
- enum is used to group a set of related named constants.
- Variables of an enum type can only hold values (named constants) that are defined as members.
- The underlying data type for the members is int by default.
- If you do not specify a value to each member, the members are initialized to 0, 1, 2, n.
- **Use enums very much**. Make also use of the object **E**num with your enums.

C# For Beginners

Methods

Contents:

Farid Naisan, University Lecturer Malmö University, farid.naisan@mau.se

Class - an introduction

- A class describes (defines) a group of things (objects) with similar properties.
- A class has two major parts:
  - **Fields** (instance variables) holding data,
  - **Methods** performing operations on the data.
- An object is an instance of (a unique instance of) a class and is created with the keyword new.

Methods perform tasks

- Every good program is divided into smaller parts, i.e. classes.
- Every good class performs its task in smaller pieces, i.e. methods.
- Every good method should perform a single well-defined task and not mix different tasks.
  - It should do it correctly and as independent as possible.
- Methods are also called functions or procedures in other languages.

Method declaration

- Every method has two parts:
  - A method definition (method header and method signature).
  - A method body:

```
<modifier> <return type> MethodName (parameters)
{
    /*bod*/
}


private bool CheckValue(double amount)
{
     /*...body...*/
}
public double Interpolate ( /*... parameters*/    )
{
    /*... body... */
}
```

Static methods

- A `static` method, also called a class method, is a method that belongs to a class rather than specific instances of the class.
  - A `static` method is called by using the class name without instantiation:

    ClassName.*MethodName*(*arguments*)
  - `Math.Round (),` (Math is a class)
  - All methods of the `Math` class are `static`
    - `Math.Max( var1, var2 );`
- You do not create an instance of the Math class to call its methods.

Method main

- The **Main** method is the entry point of a C# console application or windows application.
- When the application is started the Main method is the first method that is invoked.
- A class containing the Main method is one that is run at the very beginning of an application and therefore there is no possibility, for CLR, of instantiating the class.
- Therefore, the method **Main** is declared `static` so it can be called without creating an object of the class containing Main .
- Any class can contain a **Main** method, but you must select one class containing a Main method as a start class. You must compile your program with the **/Main** compiler option

The Main method – four ways to write

```
static void Main ( )
{
    //...
}

static void Main (string[] args)
{
    //...
}

static int Main ( )
{
    //...
    return 0;
}

static int Main (string[] args)
{
    return 0;
}
```

args are the command line arguments – string values that can be passed to the application when starting the application

`void` Methods and Value-Returning Methods

- A void method is one that simply performs a task and then terminates.

```
Console.WriteLine ( "Hi!" );
```

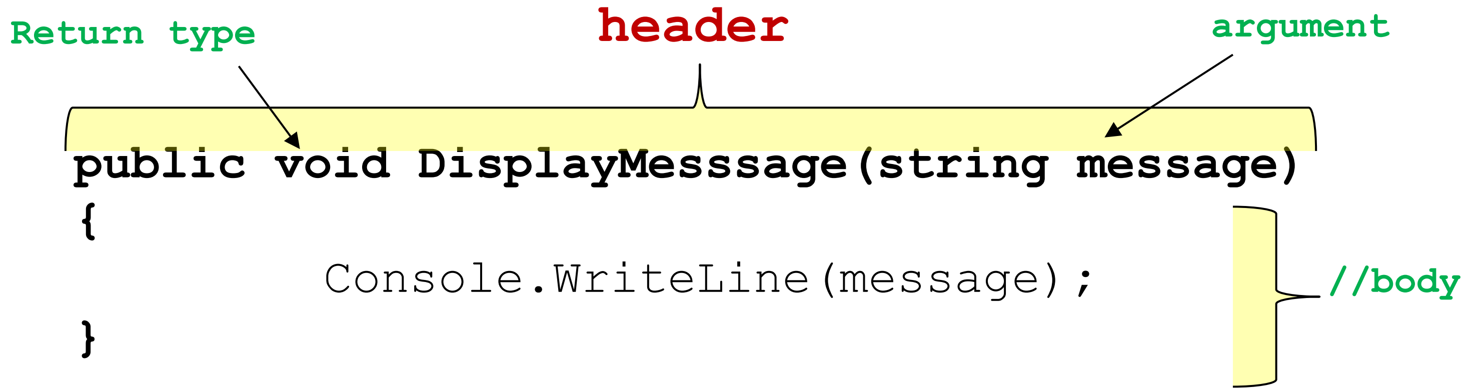
- A value-returning method not only performs a task, but also sends a value back to the caller method.

```
int number = Covert.ToInt32 ( txtNumber.Text );
```

- In this example the return value of the method **ToInt32,** belonging to the struct **Convert,** is saved in the variable number (in the caller method).

Defining a `void` Method

- To create a method, you must write a definition, which consists of a header, and a body.
- The method header lists several important things about the method, including the method's name.
- The method body is a collection of statements that you program to be performed when the method is executed.



```
public void DisplayMesssage(string message)
{
        Console.WriteLine(message);
}
```

Parts of a Method Header

- Method **modifiers:**
  - The modifiers private, protected, public, that determine the accessibility of the method outside the class.
  - A static - method belongs to a class, not a specific object.
- **Return** type: a data type for the value that a method returns. void methods do not return any value.
- Method **name**: name that is descriptive of what the method does.
  - The same naming rules as for variables apply.
- **Parentheses:** - contain nothing or a list of one or more variable declarations if the method should have parameters for receiving input, sending output or both.

Calling a Method

- A method executes when it is called.
- A void method is called in a separate statement as below:

```
DisplayMessage();   //call to a void method
```

- **DisplayMessage** is a void method – no return value.
- A value-returning method is called by using an assignment statement (1) or can be used directly like a variable (2):

```
double number = ReadADouble ( );   //(1)
double result = ReadADouble ( ) / 2.0;  //(2)
```

Passing Arguments to a Method

- Values that are sent to a method are called arguments.

```
Console.WriteLine("Hello");

number = Convert.ToInt32(str);
```

- The data type of an argument in a method call must correspond to the variable declaration in the parentheses of the method declaration. The parameter is the variable that holds the value being passed into a method.
- By using parameter variables in your method declarations, you can make your methods to accept data this way.

```
public String FormatValutaString(decimal kronor)
{
    //code
    return aStringValue;
}
```

Passing a value

```
DisplayValue(5);  //call the method passing a value
```

The argument 5 is copied into the parameter variable ***num***.

```
public static void DisplayValue (int num)
{
      Console.WriteLine ( "The value is {0}!", num );
}
```

Output from the method:

```
The value is 5!
```

Arguments and Parameter Data Type Compatibility

- When you pass an argument (value) to a method, be sure that the argument's data type is compatible with the parameter variable's data type.
- C# will automatically perform widening conversions, but narrowing conversions will cause a compiler error.

```
double d = 1;    //  1 (int) is converted 1.0 (double) --> widening
DisplayValue(d);      //Error! Can't convert double to int (narrowing)
```

- Widening is the case when a smaller size numeric data is converted to a bigger size.
  - When an **int** value is converted to a **long** value.
- Narrowing is the opposite of widening.
  - When a **int** value is converted to an **short**.

Arguments Passed by Value

- In C#, arguments of the standard data types can be passed by value, which means that only the argument’s value is passed to method.
- A method’s parameter variables are distinct from the variables that are used in a method call in caller method, even if they may have same names.
- If a parameter variable is changed inside a method, it has no effect on the original argument.

```
string strFormat = FormatValueSting ( amount ); //method call

  public String FormatValutaString(decimal amount)
  {
      //code
      return aStringValue;
  }
```

The two “amount” variables are distinct and separate

Pass by value

Passing Multiple Arguments

*The argument 5 is copied into the **num1** parameter.*
*The argument 10 is copied into the **num2** parameter.*

```
ShowSum(5,10);
```

NOTE: Order matters!

```
public static void ShowSum (double num1, double num2)
{
    double sum; //to hold the sum
    sum = num1 + num2;
    Console.WriteLine ( "The sum is {0}", sum );
}
```

Arguments Passed by Value

- In C#, arguments of the standard data types can be passed by value, which means that only a copy of an argument's value is passed into a parameter variable.
- A method's parameter variables are separate and distinct from the arguments that are listed inside the parentheses of a method call.
- If a parameter variable is changed inside the callee method, it has **no** effect on the original argument in the call list, in the caller method.

Pass by value

```
public String FormatValutaString(decimal kronor)
{
    //code
    return aStringValue;
}
```

Arguments Passed by Reference

- In C#, arguments of the standard data types can be passed by reference.
- Reference parameters are declared like regular variables, except the key word ref is in front of the method parameter
- If a reference parameter variable is changed inside the callee method, it **affects** the corresponding variable in the call list, in the caller method.

```
public string FormatValutaString(ref decimal kronor)
{
    //code
    kronor += 50.0;   //change affects the corresponding variable in the caller method as well!

    return aStringValue;
}
```

Arguments Passed with out-modifier

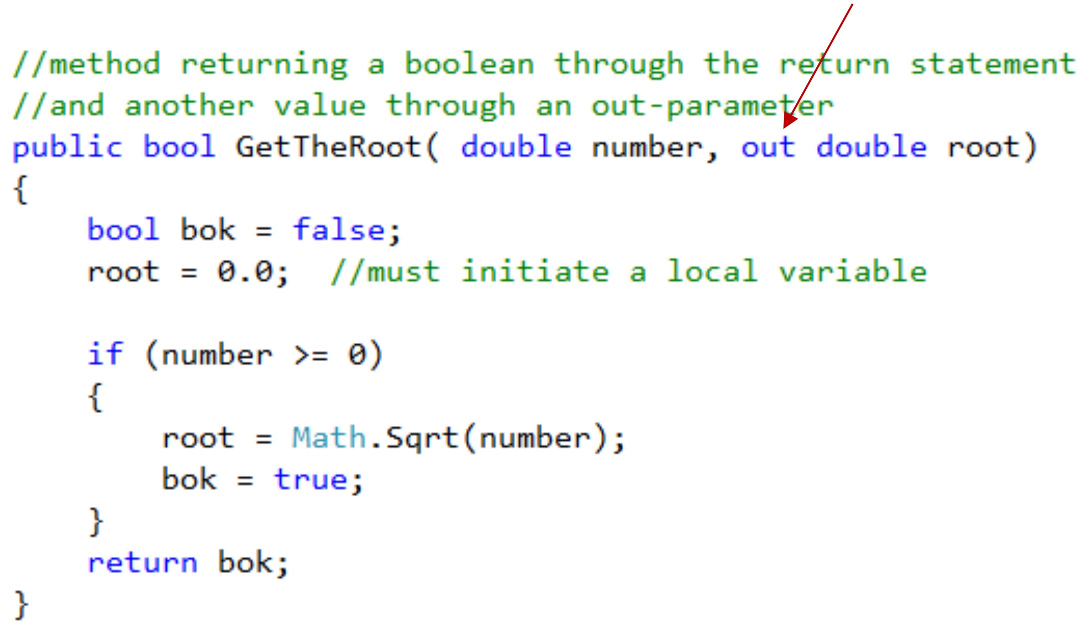
- C# also has output parameters that are declared with the **out** keyword.
- Output parameters, which are similar to reference parameters, are used when the method yields a new value for the parameter without modifying the original parameter.
- When calling a function with output parameters or reference parameters, be sure to include the key words out or ref before the data type as well.

```
decimal amount = 700.0m;
string strIn = FormatValutaString(ref amount);

bool goodNumber = double.TryParse ( Console.ReadLine ( ), out amount );
```

The out-prameter

- The out parameter is used to send values from a callee method to the caller method in addition to the return value.
- A method can send more than one value to the caller.



```
//method returning a boolean through the return statement
//and another value through an out-parameter
public bool GetTheRoot( double number, out double root)
{
    bool bok = false;
    root = 0.0;  //must initiate a local variable

    if (number >= 0)
    {
        root = Math.Sqrt(number);
        bok = true;
    }
    return bok;
}
```

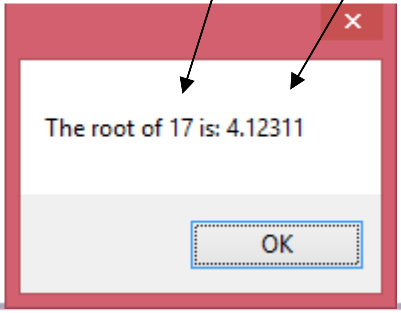
Calling a method with out-parameter

- When calling a method with out parameter, the keyword out must be given in the call.

```
public void TestRootMethod()
{
    double root = 0.0;
    double testNumber = 17.0;

    bool bok = GetTheRoot(testNumber, out root);

    if (bok)
        MessageBox.Show(String.Format("The root of {0} is: {1:f5}", testNumber, root));
    else
        MessageBox.Show("The root cannot be calculated!");
}
```


The root of 17 is: 4.12311

Difference between out and ref

- Both out and ref parameters are passed by reference, i.e. only the address of the variable is passed.
- However, there is at least two big differences.
  - The ref parameter must be definitely assigned a value in the caller method before being passed to the callee method.
  - The out parameter does not need to be initiated or have a value assigned in the caller method before being passed to the callee method.
  - The ref parameter does not need to be assigned a new value in the callee method.
  - The out parameter must definitely be assigned a new value in the callee method.

Passing string Object References to a Method

- Recall that a variable having a class as data type does not hold the actual data item that is associated with it, but holds the memory address of the object.
- A variable associated with an object is called a **reference variable**.
- When an object, such as a string is passed as an argument, it is actually the reference to the object that is passed.

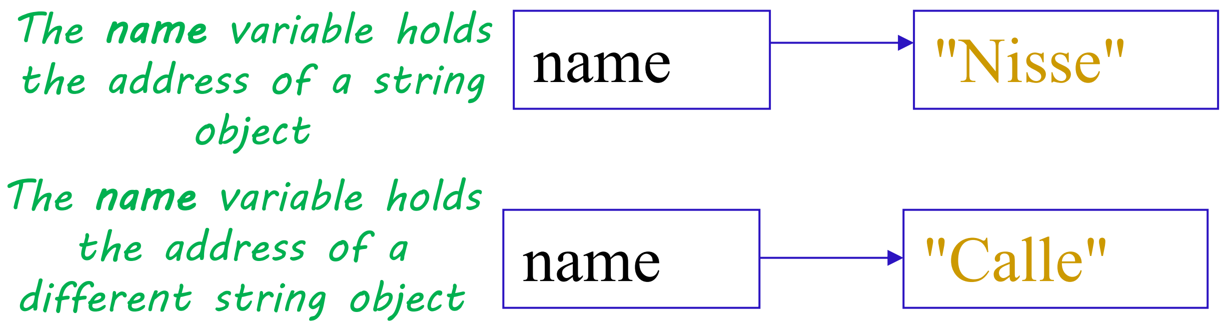
```
string name = "Nisse";
ShowLength(name);
```

strings are Immutable Objects

- Strings are immutable objects, which means that they cannot be changed. When the line

  ```
  name = "Calle";
  ```

  is executed, the **name** object being an immutable object, cannot change, so when copying a new string to **name**, a new string object is created. This is done automatically behind the scene, and you don't have to worry about it.


The name variable holds the address of a string object
name
"Nisse"
The name variable holds the address of a different string object
name
"Calle"

The **params** keyword

- The keyword params can be used to specify a method parameter that takes a variable number of arguments..
- The parameter declared with params must be the last one in the argument list.
- The parameter must be declared as an array (ex int [ ], or double [ ]).
- You can then call a method and send a comma separated list of arguments of the type specified in the parameter declaration.
- You can also send an array of the specified type.
- You can also choose to send no arguments for the parameter.

More About Local Variables

- A local variable is declared inside a method and is not accessible to statements outside the method .
- Different methods can have local variables with the same names because the methods cannot see each other’s local variables.
- A method’s local variables exist only while the method is executing. When the method ends, the local variables and parameter variables are destroyed and any values they store are lost.
- Local variables are not automatically initialized with a default value and must be given a value before they can be used.

Returning a Value from a Method

- Data can be passed into a method via parameter variables.
- Data may also be returned from a method, back to the statement that called it.

```
int num = Convert.ToInt32 ( "700" );
```

- The string "700" is passed into the ToInt32 method.
- The int value 700 is returned from the method and stored in the **num** variable .

Defining a Value-Returning Method

```
public static int Sum (int num1, int num2)
{
   int result;
   result = num1 + num2;
   return result;
}
```

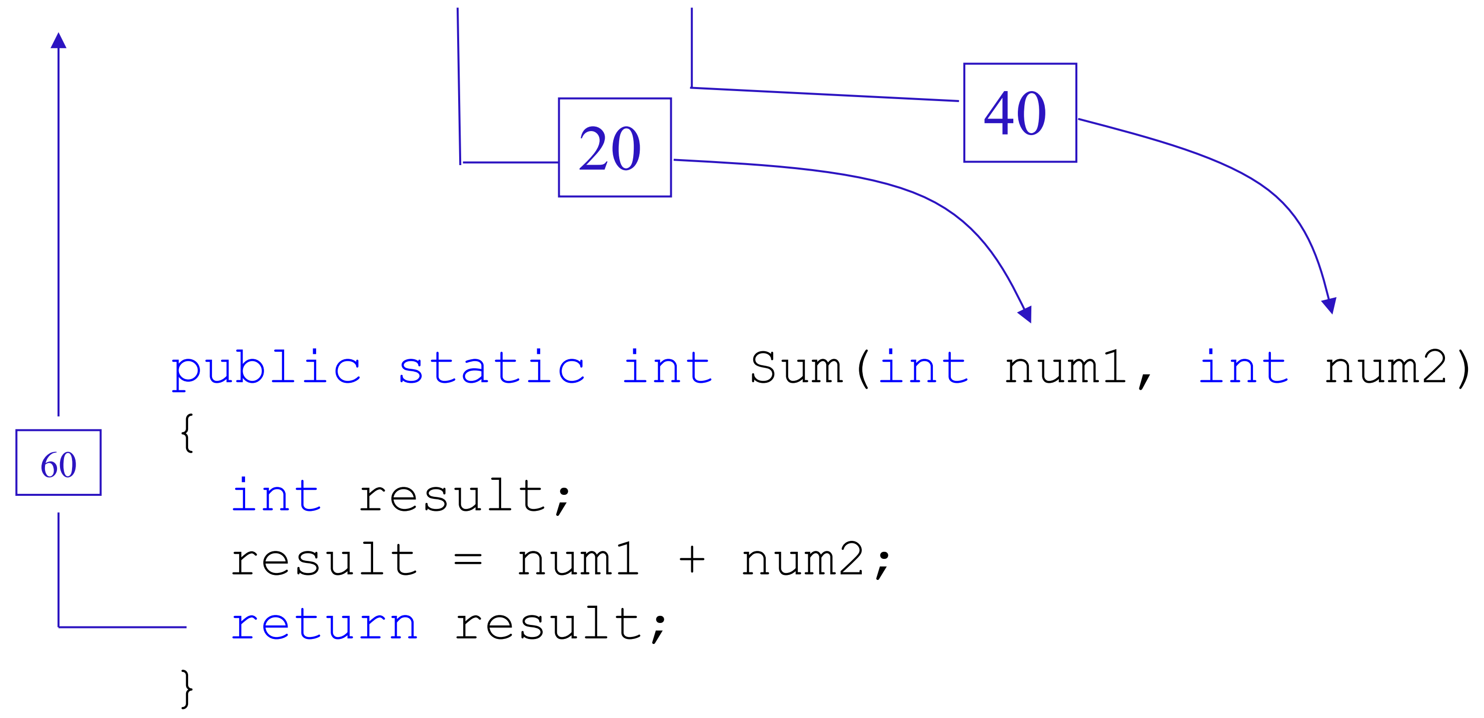
Return type

The return statement causes the method to end execution, and it returns a value back to the statement that called the method.

This expression must be of the same data type as the return type

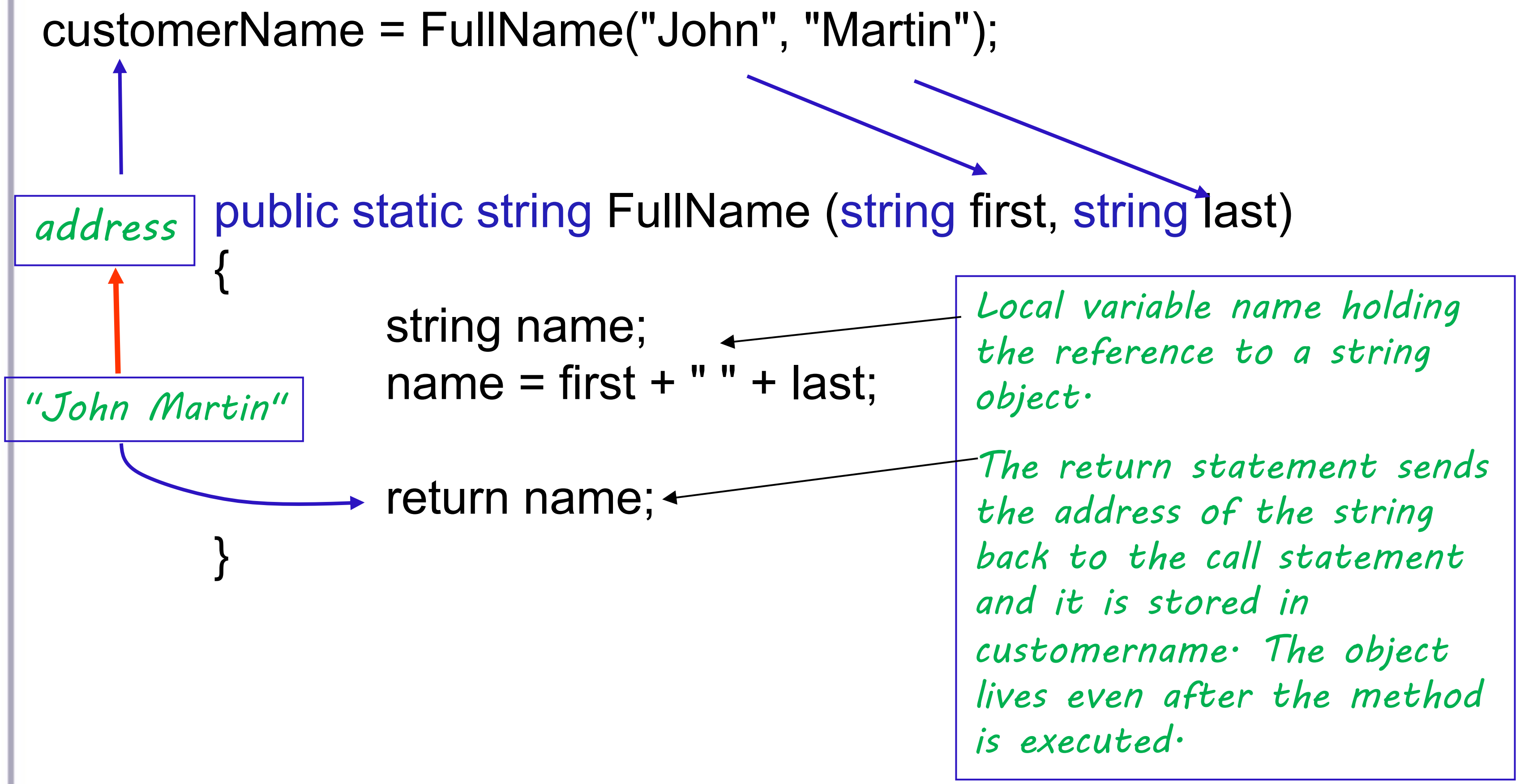
Calling a Value-Returning Method

total = Sum(value1, value2);



```
public static int Sum(int num1, int num2)
{
  int result;
  result = num1 + num2;
  return result;
}
```

Returning a Reference to a `string` Object



```
customerName = FullName("John", "Martin");

public static string FullName (string first, string last)
{
        string name;
        name = first + " " + last;

        return name;
}
```

Local variable name holding the reference to a string object.

The return statement sends the address of the string back to the call statement and it is stored in customername. The object lives even after the method is executed.

Method overloading

- Method overloading imply that a class can ha several methods with the same name but with different parameter constructions according to the following rules:
  - Different number of parameters
  - Different parameter types or parameters types in different order
  - Different passing method, i.e. pass by value or by ref an out.
- Methods that only differ in return type are not allowed and thus cannot be overloaded.
- When an overloaded method is called, the compiler selects the one with parameters that matches best with the arguments.

Overloading - examples

- The example below shows a method overloaded in several ways.
- The **Console.WriteLine** method has been overloaded in many forms and therefore you can call the method with different types of argument.

```
Console.WriteLine(intValue);   //integer
Console.WriteLine("Hej")        //string
//Mixed types
Console.WriteLine("Num1: " + i + "  Num2: " + sum);
```

```
void AMethodName ( int num1 ){ ... }
void AMethodName ( ref int num1 ){ ... }
void AMethodName ( int num1, long num2 ){ ... }
void AMethodName ( long num1, int num2 ){ ... }
void AMethodName ( int num1, long num2 ){ ... }
void AMethodName ( char num1 ) { ... }
```

Summary

- A large, complex problem can be solved one piece at a time by methods.
- Each method should do only one specific task.
- As good programming, the name of a method should express the task. Use longer but more expressive names than short names that don’t say much.
- A method should always be documented by writing comments that appear just before the method’s definition. The comments should provide a brief explanation of the method’s purpose.
- Never write the same code twice. "Copy-paste" code is bad style. Write a method and call the method instead – as many times as needed.
- DO NOT write long methods!

How do we know if we need parameters in a method? Why do we need them?

As yourself the following question, when writing a method with or without parameters:

Does the method have access to the data it needs to perform its task?

- If yes:
  - No parameters
- Else
  - Which value(s) the method needs to have to do its task.
    - For each value, determine the type and then declare a parameter for it.
    -

Example: BMICalculator

We have to calculate BMI and for that we need to know the height, the weight and the name of the person whose BMI is to be calculated. Height and weight can have decimal values (double type) and name is text (string). We write a class **BMICalculator**.

We write one or more methods to calculate BMI. These methods MUST have the above values because they must come from the **MainForm** (or another class that uses **BMICalcualtor**).

Two ways to go:

1. We write methods with parameters to receive the above.
   ```
   public  double CalculateBMI (string name, double height, double weight)
   {
     //code
   }
   ```

   Every time the method is called, the values must be provided by the caller.

2. We let the class have instances variables to store the values. Once the values are in an instance of the class, then all methods of the **BMICalculator** will have access to them and can use them in their calculations.

Which alternative to choose? The second one is easier and better if the data are input to the class. They belong to the class and each object must know its input data (like every person must know its own properties, name, height, weight, etc).

```
class BMICalculator
{
   private string name = "No Name";
   private double height = 0;   //m, ínch
   private double weight = 0;   //kg, lb
   private UnitTypes unit;
```

But there is a problem. The above data must come from the client of the class (**MainForm** which in turn takes the values from the user of the application), but **MainForm** does not have access to the private fields! We actually don't want them to have direct access!

What to do?

Write a public method and let the caller send each of the value as a parameter. We check the values and if they are good, then save them in the above.

In Assignment 3, we use a standard method (setter).

In Assignment 4, we use Properties instead.

We can choose to have more than one parameter in a method (all the three) or work with each parameter separately. Use the latter (when it is about input):

```
public void SetName(string value)
{
    //validate value before accepting it!
    if (!string.IsNullOrEmpty ( value ))
        name = value;
}

public void SetHeight (double value)
{
    if (value >= 0)    //validate before accepting
        height = value;
}

public void SetUnit(UnitTypes value)
{
    //No validation needed as the caller cannot
    //give any wrong value
    unit = value;
}
```

The main purpose for BMICalculator is to receive the input in some way. This is how it has to be done. Once you understand the above, then you will have a clear understanding of why we can use Properties instead (next module).

The class defines method (so called setter methods) to receive input from a client object (MainForm), but the client may lose the above values because it assumes that **BMICalculator** maintains them. If **MainForm** or any other object wants to know which height, name or unit values stored in an **BMICalculator** object, they should have access to the values (if you, the programmer think it is ok). To give them access to the values, we write getter methods:

```
public double GetHeight()
{
    return height; //no need for any manipulation
}

public string GetName ( )
{
    return name;
}
```

Etc.

The idea behind method parameters (argument) is to pass values to the methods or receive values (in addition to the return value). In the latter case, we use out (or ref) type of variables (as in TryParse methods).

Now the caller (MainForm) can use the above setter methods to furnish the object of **BMICalculator** it uses with input, and whenever it needs to know about the value again, it calls the getter methods.

The **MainForm** class:

```
public partial class MainForm : Form
{
    // create an instance of BMICalculator
    private BMICalculator bmiCalc = new BMICalculator ( );
```

MainForm gets its instance of the **BMICalculator** and use it, just as you buy a hammer and use its capabilities.

MainForm will pass input to the bmiCalc (assume for a while that we send the value without any processing and that the value is a good number)

```
private bool ReadHeight ( )
{
    double outValue = 0;

    if (double.TryParse ( txtHeight.Text, out outValue ); //user’s value
       bmiCalc.SetHeight ( outValue);  //setter method to save the value in the
                                       //object
```

The method **SetHeight** is the setter method in the **BMICalculator**.
You can go on with the other input in the same manner.

We are not using the getter methods but they are good to have for the future.

Sometimes you will pass values between methods of the same class, the same argumentation is to be used to decide if parameters are needed. In fact in all cases, decide the need for method parameters based on whether the methods need extra values in addition to what they already have access to. Sometimes you may need to have method parameters for both input to and output from the same method. A good example is the TryPars (which .NET programmers have designed).In the C# language specification, the method has the following structure:

```
public static bool TryParse(
        string s,
        out int result
)
```

s is input to the method result is output from the method. It has a bool return type too! In C# you can only have one return value. If you need more values from a method, you can use out-parameters as in above.

```
bool ok = double.TryParse ( txtHeight.Text, out outValue );
```

We send the text (ex “123.5” five chars) and get its corresponding double value (123.5, a double). In addition, we also receive a true value (if outValue = 123.5) or a false value (if something goes wrong) as the return value of the method.

Vad är en setter- och en gettermetod och varför skall vi lära oss dessa?

Inkapsling är det första OOP-koncept man måste lära sig. Man kapslar in relaterade data (instansvariabler) och operationer (metoder) i en enhet som vi nu vet är en klass. Man ska även skydda sina data och gömma manipulering av data och hur man löser olika saker i metoderna (implementationer). Hur skyddas data och implementationer? De skyddas med att deklarera dem som privata medlemmar.

Tolkning av OOP-reglerna:

- Deklarera alla instansvariabler (kallas också för fält eller attribut) som **private** (ibland **protected,** diskuteras i senare moduler).
- Deklarera alla de metoder som används internt i klassen som **private**.

Kommunikationen med andra objekt görs då via ett väl-definierat gränssnitt. Gränssnittet består av ett antal **public**-definierade metoder som antingen erbjuder tjänster till andra objekt eller begär tjänster från andra objekt.

Setter- och getter metoder:

Objekt behöver självklart kommunicera med varandra och beställa tjänster av varandra för att gemensamt lösa ett problem. Därför måste vissa metoder deklareras **public**. Ett bra exempel på sådana metoder är s.k. setter- och getter-metoder som i C# är standardiserade som **Properties**. Dessa ger tillgång till privata instansvariabler under kontrollerade former. Många andra språk såsom Java och C++ saknar Properties , istället använder de sig av vanliga metoder fast de kategoriserar dessa under konceptet getter- och setter-metoder.

Getter:

En getter-metod returnerar värdet av en instansvariabel. Här finns det möjlighet att bestämma hur värdet skall lämnas till den anropande metoden. Man kan manipulera värdet, skicka en kopia av ett objekt, eller helt enkelt returnera instansvariabeln utan någon förbehandling.

- Metoden skall ha samma retur-typ som variabeln.
- Metoden behöver ingen metodparameter.

Här är ett exempel: en getter-metod ger andra objekt tillgång till en instansvariabel som heter **maxAttempts**.

```
/// <summary>
/// A getter method giving access to the value of the instance
/// variable maxAttempts
/// </summary>
/// <returns>Returns the value saved in the variable maxAttempts.</returns>
public int GetMaxAttempts()
{
   return maxAttempts;
}
```

Setter:
En setter-metod tar emot ett nytt värde via dess argument och, efter en kontroll av något slag, tilldelas värdet till instansvariabeln.

- Metoden behöver en metodparameter of samma typ som instansvariabeln.
- Metoden behöver ingen returtyp.

Metoden nedan är en motsvarande setter-metod.

```
/// <summary>
/// Setter method giving WRITE-ONLY access to the value of the instance
/// viable maxAttempts.
/// </summary>
/// <param name="newValue">The variable maxAttempts is assigned a new
value.</param>
public void SetMaxAttempts(int newValue)
{
    if (newValue >= 0)
        maxAttempts = newValue;
}
```

Notera, att in-parametern **newValue** valideras så att den har ett giltigt värde. I det fallet att parametern är av en objekttyp, kan valideringen bestå av att kontroller så att referensvariabeln inte är **null**.

Varför använda getter- och setter-metoder när vi har Properties in C#?

Properties är självklart vad som skall användas i C# (nästa modul), men det är nog bara en fördel att bekanta sig med getter- och setter-metoder också.

1. Om ni stöttar på dessa begrepp i litteraturen, och i fall ni i framtiden vill lära er ett annat språk som inte stödjer Properties, har ni redan kunskapen. Dessa metodtyper, fast de är vanliga metoder, är kända begrepp bland programmerare.
2. Ni är förberedda och kommer utan tvekan att förstå och lära er Properties bättre och också förstå syftet med Properties.
3. Metoderna är vanliga metoder och det är absolut ingen nackdel att kunna dessa, trots att vi fr.o.m. nästa modul bara kommer att använda Properties för samma ändamål.