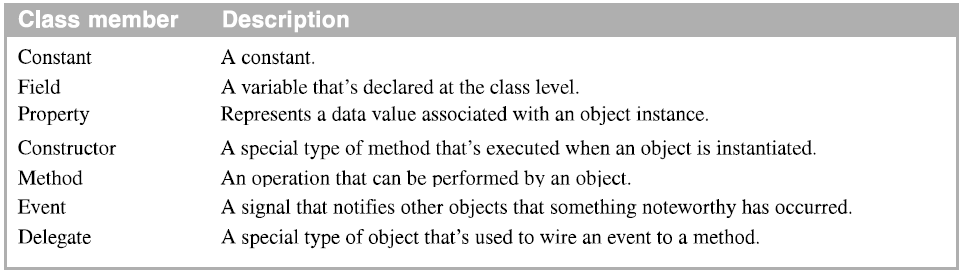
**Classes**

## **Class Members**

A class consists of several members, in our basic class we have touched on a few fields, constructors and methods.

Now we will add a couple and look at them in some detail.



|  |
| --- |
| public class Converter  {  // ----- Constants -----    // ----- Fields -----    // ----- Properties -----    // ----- Constructors -----    // ----- Methods -----    // ----- Events -----    // ----- Delegates -----  } |

Our class now has the following elements:

* The class is declared **public,** this means that objects of this class are visible to and can be constructed by other objects. **class** is used to denote that we are declaring a class. **Converter** is the name we give to the class. Note the **curly brackets { }** these are used to define the start and finish of the code within our class.
* **Constants**, a value that will be used whenever the constant is used in the code.
* **Fields**, store data for each object.
* **Properties**, provide a flexible mechanism to read, write, or compute the value of a private field.
* **Constructors**, allow each object to be set up properly when it is first created.
* **Methods,** implement the behaviour of the objects.
* **Events**, code that executes when an event is fired.
* **Delegate**, allows us to link an event to a method.

**Encapsulation**

One of the principles of an Object Oriented language is the concept of data encapsulation. When we create a class it is useful to be able to define who has access to the data and members inside our class. Encapsulation lets a programmer hide (encapsulate) the data and operations of a class while exposing others.

For now we use the following access modifiers to achieve this:

|  |  |
| --- | --- |
| **Access Modifiers** | |
| public | The type or member can be accessed by any other code in the same assembly or another assembly that references it. |
| private | The type or member can be accessed only by code in the same class. |
| internal | The type or member can be accessed by any code in the same assembly (program), but not from another assembly. Acts the same as public when in the same project. |

Constants

Constants allow us to set up a fixed value that will be used whenever the constant is used in the code.

Variables of type **const** aregiven an initial value which may not be change by your program.

You can also use the ***readonly*** keyword as a modifier on a field to produce the same effect as a constant.

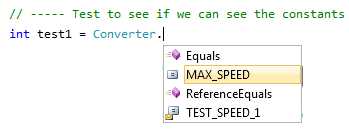
The following code shows how to use constants:

|  |  |  |
| --- | --- | --- |
| // Define the object  Converter testConvert;  // Constructor  public MainPage()  {  this.InitializeComponent();  // Initialise (instantiate) object  testConvert = new Converter();  // Test to see if we can see the constants  // A public constant is accessed by using  // the class name as the reference  int test1 = Converter.MAX\_SPEED;  // An internal constant is accessed by  // using the class name as the reference  int test2 = Converter.TEST\_SPEED\_1;    // A read-only field is accessed using the  // object as a reference  int test3 = testConvert.TEST\_SPEED\_2;  } |  | // Constants in Converter class    // Constant used by all the application  public const int MAX\_SPEED = 100;    // Constant only used by in the class private const int MIN\_SPEED = 20;    // Constant used by all the application  internal const int TEST\_SPEED\_1 = 20;  // Constant used by all the application, field set to read only  public readonly int TEST\_SPEED\_2 = 20;  public Converter()  {  // Test to see the constants  int test1 = MAX\_SPEED;  int test2 = MIN\_SPEED;  int test3 = TEST\_SPEED\_1;  int test4 = TEST\_SPEED\_2;  } |

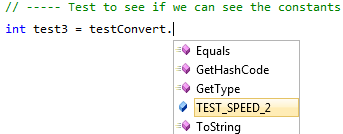
In the right pannel there are four constants defined all of which can be used in the Converter class, in this example in the constructor.

Note: The convention is using Upper case and underscored for the constant names.

In the left pannel the constants are used in the main page.



The constants need to be defind as ***public*** or ***internal*** to be used and they are referenced by using the ***class name*** (converter).

The constant defind as a field using the ***readonly*** modifier can also be used but it is accessed using the ***object name*** (testConvert).

**Fields**

## Fields are used to store data for an object to use.

The convention is to declare fields as ***private*** members of the class and use a ***lowercase character for the first letter of the name***.

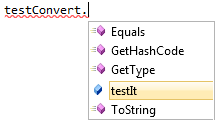
The following code shows how to use fields:

|  |  |  |
| --- | --- | --- |
| // Define the object  Converter testConvert;  // Constructor  public MainPage()  {  this.InitializeComponent();  // Initialise (instantiate) object  testConvert = new Converter();  // A public field is accessed by  // using the object as a reference  int test4 = testConvert.testIt;  } |  | // Fields in Converter class    // Field only used by in the class private double miles;  // Field only used by in the class private double kilometres;  // Field used by all the application public int testIt;    public Converter()  {  // Test to see the fields  miles = 20;  kilometres = 20;  testIt = 20;  } |

In the right pannel there are three fields defined all of which can be used in the Converter class, in this example in the constructor.

Note: The convention is using lower case for the first letter of the field name.

In the left pannel the fields are used in the main page.

****

The field defind as public can also be used in the page class, but it must be accessed using the ***object name*** (testConvert).

**Properties**

Properties combine characteristics of both fields and methods (remember our ***set*** and ***get*** methods).

When we use a property it appears to be a field and accessing the property requires the same syntax. When we implement a property one or / and two code blocks are required that represent a get accessor and / or a set mutator.

The code block for the get accessor is executed when the property is read and the code block for the set mutator is executed when the property is assigned a new value.

A property without a set accessor is considered read-only, a property without a get mutator is considered write-only and a property that has both is read-write.

Properties tend to be used in combination with field the following code shows how to use properties to set and get the data in the associated field:

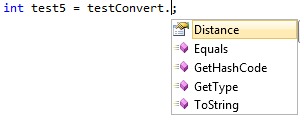
|  |  |  |
| --- | --- | --- |
| // Define the object  Converter testConvert;  // Constructor  public MainPage()  {  this.InitializeComponent();  // Initialise (instantiate) object  testConvert = new Converter();  // A public property can be set in  // this form  testConvert.Distance = (int)21;    // A public property returns the  // value in this form  int test5 = testConvert.Distance;  } |  | // Properties in Converter class    private int distance; // Private field  public int Distance // Public property  {  get  {  return distance;  }  set  {  distance = value;  }  } |

In the right pannel there is a ***private field*** defined called ***distance***. The ***Distance*** property is then used to set and get the value instead of using conventional methods.

The field is defined in the normal way as ***private.***

The property is called the same name as the filed but a capital letter is used for the first character.

The get block returns the value of the feild and the set block set the field.



The property defind as public can also be used in the page class, but it must be accessed using the ***object name*** (testConvert).

The following code shows how to use properties to ***get*** the data in the associated field:

|  |  |  |
| --- | --- | --- |
| // Define the object  Converter testConvert;  // Constructor  public MainPage()  {  this.InitializeComponent();  // Initialise (instantiate) object  testConvert = new Converter();  // A public property returns the  // value in this form  int test6 = testConvert.Speed;  } |  | // Properties in Converter class    private int speed; // Private field  public int Speed // Public property  { // read only  get  {  return speed;  }  } |

In the right pannel there is a ***private field*** defined called ***speed***. The ***Speed*** property is then used to get the value instead of using a conventional method.

The following code shows how to use properties to ***set*** the data in the associated field:

|  |  |  |
| --- | --- | --- |
| // Define the object  Converter testConvert;  // Constructor  public MainPage()  {  this.InitializeComponent();  // Initialise (instantiate) object  testConvert = new Converter();  // A public property can be set in  // this form  testConvert.Length = (int)33;  } |  | // Properties in Converter class    private int length; // Private field  public int Length // Public property  { // write only  set  {  length = value;  }  } |

In the right pannel there is a ***private field*** defined called ***length***. The ***Length*** property is then used to set the value instead of using a conventional method.

The following code shows several variations on the Properties format:

|  |  |  |
| --- | --- | --- |
| // Define the object  Converter testConvert;  // Constructor  public MainPage()  {  this.InitializeComponent();  // Initialise (instantiate) object  testConvert = new Converter();  // Standard format test  testConvert.Width = (int)22;  int test7 = testConvert.Width;  // Property with extended code  testConvert.Age = (int)18;  int test8 = testConvert.Age;    // Auto implemented property  // Note no field is set using this  // variant we are just using a  // property here to store our data.  testConvert.Height = (int)33;  int test9 = testConvert.Height;  } |  | // Properties in Converter class    // Standard format  private int width;  public int Width  {  get { return width; }  set { width = value; }  }  // Property with extended code  private int age;  public int Age  {  get  {  if (age < 0) return 0;  else return age;  }  set  {  if ((value > 0) && (value < 21))  {  age = value;  }  }  }  // Auto implemented property  // Note no field is set using this  // variant we are just using a  // property here to store our data.  public int Height { get; set; } |

In the right panel there are three variations on the Properties format:

* Standard format is used for the read / write.
* Extra code is added to set and gets in the final example.
* The auto implement format is used for the read / write. Note no field is set using this variant we are just using a property here to store our data.

### Constructors

Constructors allow each object to be set up properly when it is first created.

Using a constructor allows you to set default values, limit instantiation, and write code that is flexible and easy to read.

If you do not provide a constructor for your class, C# will create one by default (but it is good form to initialise our fields in the constructor) that instantiates the object and sets member variables to the default values as listed below.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Value type** | **Default value** | **Value type** | **Default value** | **Value type** | **Default value** |
| bool | false | float | 0.0F | short | 0 |
| byte | 0 | int | 0 | uint | 0 |
| char | '\0' | long | 0L | ulong | 0 |
| decimal | 0.0M | sbyte | 0 | ushort | 0 |
| double | 0.0D |  |  |  |  |

A class may have multiple constructors that take different arguments. It is the responsibility of the constructor to put the Object of the class in the correct state to be used when it is created.

# *Important Note:* A constructor does not have a return value and has the same

# name as the class.

The following code shows several variations on the Constructor format:

|  |
| --- |
| public sealed partial class MainPage : Page  {  // Define the objects  Converter convert1;  Converter convert2;  Converter convert3;  // Constructor  public MainPage()  {  this.InitializeComponent();  // Initialise (instantiate) objects  convert1 = new Converter();  convert2 = new Converter(44);  convert3 = new Converter(123,456);  }  } |

|  |
| --- |
| // Constructors in Converter class    // Standard format property  private int width;  public int Width  {  get { return width; }  set { width = value; }  }  // Auto implemented property  // Note no field is set using this variant we are just using a property here  // to store our data.  public int Height { get; set; }  /// <summary>  /// Constructor for the Converter class. Takes no parameters and sets the  /// default values for the fields  /// </summary>  public Converter()  {  width = 10; // Set the default value directly to the field  Width = 11; // Set the default value via a property to the field  this.Width = 12; // Set the default value via a property to the field  Height = 21; // Set the default value of a property  this.Height = 22; // Set the default value of a property  }  /// <summary>  /// Constructor for the Converter class. Takes one parameter and sets the  /// width field.  /// </summary>  /// <param name="inputWidth">User defined width size</param>  public Converter(int inputWidth)  {    width = inputWidth; // Set via the input parameter directly to the field  Width = inputWidth; // Set via the input parameter using the property  this.Width = inputWidth;// Set via the input parameter using the property    }  /// <summary>  /// Constructor for the Converter class. Takes two parameters and sets width /// and height fields. The this keyword is used to remove the ambiguity of the  /// input parameters names.  /// </summary>  /// <param name="width">User defined width size</param>  /// <param name="Height">User defined height size</param>  public Converter(int width, int Height)  {  this.width = width; // Set via the input parameter directly to the field  this.Width = width; // Set via the input parameter directly to the field    this.Height = Height; // Set the property via the input parameter  } |

The following process is used in the code example to implement constructors:

* A field (width) along with its associated property and a property (height) are declared.
* The first constructor sets default values.
* The second constructor uses an input parameter to set the width field.
* The third constructor uses input parameters to set the width field and height property. The ***this*** keyword is used to remove the ambiguity between the input parameter and field / property names.

**The *this* Keyword**

The ***this*** keyword refers to the current instance of the class and is also used as a modifier to identify class members inside a class code.

The following code shows an example use of the ***this*** keyword:

|  |
| --- |
| public Employee(string name, string alias)  {  // Use this to qualify the fields, name and alias.  this.name = name;  this.alias = alias;  } |

# Methods

## Methods are the building blocks of our class, they allow us to do things (process data, run algorithms etc.).

## The basic methods in a class tend to be ***get*** / ***set*** methods which are useful for processing several fields at once (use Properties where possible for single field processing) and methods that perform a specific coding task.

## **The Form of a Method**

The general form of a method is:

|  |
| --- |
| **Accessibility\_specifier type\_specifier method\_name(parameter\_list)**  **{**  **body of the method**  **}** |

* **Accessibility\_specifier:** specifies who see the method ***private*** only within the class ***public*** outside the class.
* **parameter\_list:** is a comma-separated list of variable names and their associated types that receive the values of the arguments when the method is called. A method may be without parameters, in which case the parameter list is empty. However, even if there are no parameters, the parentheses (brackets) are still required.
* **type\_specifier:** specifies the type of data the method returns. A method may return any type of data except an array.

#### The general form of calling a method is

|  |
| --- |
| **Class\_Object.method\_name(parameter\_list);** |

In code a method to add two numbers may look like the following:

|  |  |  |
| --- | --- | --- |
| // Define the object  Converter testConvert;  // Constructor  public MainPage()  {  this.InitializeComponent();  // Initialise (instantiate) object  testConvert = new Converter();  // Calling a method  int testMethod = testConvert.add(2, 2);  } |  | // Methods in Converter class    /// <summary>  /// Method to add two numbers.  /// </summary>  /// <param name="a">First number</param>  /// <param name="b">Second number</param>  /// <returns></returns>  public int add(int a, int b)  {  return a + b;  } |

The left panel contains the method and the right panel show the syntax for calling the method.

# Method Arguments (parameter declaration list)

If a method is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the formal parameters of the method. They are like local variables inside the method, in that; they are created on entry to the method and destroyed on exit.

The parameter declaration list for a method takes this general form:

|  |
| --- |
| **Accessibility\_specifier type\_specifier method\_name(type varname1,………, type varnameN)**  **{**  **body of the method**  **}** |

As shown in the following add method, the parameter declarations occur after the method name.

|  |
| --- |
| public int add**(int a, int b)**  {  return a + b;  } |

The method has two interger parameters **a** and **b**.

### Methods with various data types

## As with normal variables all data types can be declared as the input parameters (formal parameters) to methods.

The following show a variety of input data types.

|  |
| --- |
| public void processInt(**int** a, **int** b);  public void processFloat(**float** a, **float** b);  public void processDouble(**double** a, **double** b);  public void processLong(**long** a, **long** b); |

**Overloading Methods**

The parameters of a method can be overloaded to allow the method to be used in different ways. For this process to work the parameter lists need to be unique for each of the overloaded methods and the compiler uses a ‘best match technique’ to fit the calling code to the method in the class.

The following code explains this:

|  |  |  |
| --- | --- | --- |
| // Define the object  Converter testConvert;  // Constructor  public MainPage()  {  this.InitializeComponent();  // Initialise (instantiate) object  testConvert = new Converter();  // ----- Methods -----  // Test for overloading methods  // Integer passed to method  testConvert.testOverload(1);  // Two integers passed to method  testConvert.testOverload(1, 2);  // Integer and double passed to  // method  testConvert.testOverload(1, 3.5);    // Two doubles passed to method  testConvert.testOverload(1.5, 4.7);  } |  | // Methods in Converter class    // Test for overloading methods  // Integer passed to method  public void testOverload(int a)  {  int x = a;  }  // Two integers passed to method  public void testOverload(int a, int b)  {  int x = a + b;  }  // Integer and double passed to method  public void testOverload(int a, double b)  {  int x = a + (int)b;  }  // Two doubles passed to method  public void testOverload(double a, double b) {  double x = a + b;  } |

In the right panel there are four overloaded methods. The compiler recognises the difference between them depending on the configuration of the input parameters and then selects the best match.

# Returning from a method

There are two ways that a method terminates execution and returns to the caller, via the ‘***}***’ or via a ***return*** statement.

## The flow of control in the example diagram (program) starts at the top of the code:



## When method ***someMethod()*** is called the flow of control goes into the method and the associated code in the method is executed.

## Then because there are no more statements and the ‘}’ is reached the flow of control returns to the point in code where the call to method ***someMethod()*** was made.

## The next statement is then executed.

The second way that a method terminates execution and returns to the caller is via the return statement.

##### The return Statement

The ***return*** statement has two important uses.

* Firstly, it causes an immediate exit from the method that it is in. That is, it causes the program execution to return to the calling code.
* Secondly, it may be used to return a value.

## In the example here the method testVal() is used to test which value is larger a or b.

## The flow of control in the program starts at the top of the code.



## Then when the method testVal() is called the flow of control goes into the method testVal().

## The first test is made and if the result is true the method exits via the return statement and passes a value back to main where the variable val is given the value.

## If the first test is false then second test is made. If the result of this test is true then the method exits via the return statement and passes a value back to main where the variable val is given the value.

## If the second test is false then the method exits via the final return statement and passes a value back to main where the variable val is given the value.

## The next statement in main is then executed.

# The Return Type of a method

The type\_specifier specifies the type of data the method returns.

|  |
| --- |
| **accessibility\_specifier type\_specifier method\_name(parameter\_list)**  **{**  **}** |

A method may return any type of data except an array.

# Returning Values

### All method except those of type *void*, return a value.

### This value is explicitly specified by the return value and the type is defined by the return type.

## If a method is not declared as type void, it can be used as an operand in any valid expression.

|  |
| --- |
| **val = multiply(2, 2);**    **if (multiply(4, 2) < 50)**  **{**  **// Some Code**  **}**  **public int multiply(int a, int b)**  **{**  **return(a \* b);**  **}** |

# Methods that Return Non-integer Values

## Methods may be declared to return any data type except arrays.

The following are all valid method types.

|  |
| --- |
| **public int addInteger(int a, int b)**  **public float addFloat(float a, float b)**  **public long addLong(long a, long b)**  **public char testChar(char ch);** |

As stated previously the **type\_specifier** specifies the type of data the method returns and the value is returned via a **return** statement inside the method (the return value must be the same type as the return type).

**Methods of Type *void***

The ***void*** type is used to declare methods that do not return a value.

# Scope of Methods



## The scope rules of a language are the rules that govern if a piece of code knows about or has access to another piece of code or data.

## Each method is a discrete block of code, the code inside the method cannot be accessed except through a call to the method.

## Variables that are defined inside a method are called local variables.

Methods Used to Get the Value from a Field (Accessor methods)

These methods return information about the state of an object.

The following code explains this:

|  |  |  |
| --- | --- | --- |
| // Define the object  Converter testConvert;  // Constructor  public MainPage()  {  this.InitializeComponent();  // Initialise (instantiate) object  testConvert = new Converter();  // Test Accessor (get) methods  int test = testConvert.getWidth();  } |  | // Methods in Converter class    // Accessor (get) methods  public int getWidth()  {  return width;    // Could also use this format  // return this.width;  } |

The above method accesses the data in the **width** field, by returning the value in the field.

Note: The return **type** must match the **type** of the value returned.

**Methods Used to Change the Data in Fields (Mutator methods)**

These methods are used to change the state of an object, by changing the data in a field.

The following code explains this:

|  |  |  |
| --- | --- | --- |
| // Define the object  Converter testConvert;  // Constructor  public MainPage()  {  this.InitializeComponent();  // Initialise (instantiate) object  testConvert = new Converter();  // Test Mutator (set) methods  testConvert.setWidth(33);  } |  | // Methods in Converter class    // Mutator (set) methods  public void setWidth(int inputWidth)  {  width = inputWidth;  }  // The same code could also be written in  // this format  public void setWidth(int width)  {  this.width = width;  } |

The above method change the state (update the values) of the width field.

**The *static* Keyword**

***Static*** members allow you to call the member of a class without creating an object of the class. It is the same process when you use the functionality of say the string class. The ***static*** keyword allows us to create this type of class member for ourselves.

To create a static member you simple include the ***static*** keyword into the declaration of the member.

The following code explains this by calling static members of the ***Utils*** class :

|  |  |  |
| --- | --- | --- |
| // Constructor  public MainPage()  {  this.InitializeComponent();  // Test a static method  int testSt = Converter.getNum();  // Process the field directly  Utils.testMe = 20;  int test11 = Utils.getTestMe();  // Process the field via the Property  Utils.TestMe = 30;  int test12 = Utils.TestMe;  // Process the Property  Utils.TestProperty = 40;  int test13 = Utils.TestProperty;    // Process the field via methods  Utils.setTestMe(40);  int test14= Utils.getTestMe();  } |  | // Basic test for static keyword  // Makes this a static (global) field  public static int testMe;  // Makes this a static (global) property  // linked to a field  public static int TestMe  {  get { return testMe; }  set { testMe = value; }  }  // Makes this a static (global) property not // linked to a field  public static int TestProperty { get; set; }  // Set the field  public static void setTestMe(int a)  {  testMe = a;  }  // Get the field  public static int getTestMe() {  return testMe;  }  // Basic test for static keyword  public static int getNum()  {  return 123456789;  } |

A typical use for static members is to develop a utilities class of handy functionality that can be called directly as the following code shows:

|  |
| --- |
| // Typical use of a static keyword  // Call to the method  double cost = Utils.calcRate(20,0.8);    // Method from utils class  public static double calcRate(double total, double rate)  {  return total \* rate;  } |

The class can also be declared static if all the members are static in the class.

|  |
| --- |
| public static class Utils  {  } |