VOLUME 06 | ISSUE 02

digit FastTrack

YOUR HANDY GUIDE TO EVERYDAY TECHNOLOGY

TO .NET

A 9.9 Media Publication

www.thinkdigit/forum

Join the forum to express your views and resolve your differences in a more civilised way.

thinkdigit FORUM

Post your queries and get instant answers to all your technology related questions

one of the most active online technology forum not only in India but world-wide

JOIN NOW

www.thinkdigit.com

digit

thinkdigit.com

Fast Track to .NET

thinkdigit.com

Credits

The People Behind This Book

Editorial

Editor	Robert Sovereign-Smith
Head-Copy Desk	Nash David
Writer	Rahil Banthia

Design and Layout

Senior Designers	Baiju NV, Chander Dange
Cover Design	Anil V K

Contents

Introduction

This Fast Track to .NET will get you started with Microsoft's programming model, engine and tools for quickly building Windows applications targeted towards the .net framework. The release of .NET Framework 4 changes several development paradigms from previous versions: everything is now easier to create, execute, and maintain, and you can implement a host of new functionality.

In the first few chapters, you will get to know the basics of creating, hosting and running a .net application. The next few chapters on C# are intended to be an introduction to the programming language and its various constructs in the .NET Framework 4 and Visual Studio 2010, including the new GUI designer, expressions, variables and arguments. Additionally, you will explore the use of some basic built-in activities.

Next, you will be creating a GUI application that mimics a Calculator. The workflow of the application will be constructed both using the designer and XAML, and also using C# code. You will also get to learn event handling and adding conditional logic to the application by using various commands and functions. Finally, you will learn about thread handling and parallelism in .net applications.

Across this book, we will be using the "write the code first" approach, which primarily consists on first writing a code for the new feature being added and then understanding the necessary code to learn the finer nuances of .NET programming.

1 Basics of .NET development

The .NET framework was introduced by Microsoft in June 2000 with a vision for embracing the internet. At the heart of its strategy was language and platform independence which can be arguably denied because of its close tie-up with the Windows operating system. But what it did give you was the ability to use various languages like Visual Basic .NET, Visual C++ .NET, C#, IronPython and more, and executes your program irrespective of the language you used with the Common Language Runtime (CLR). .NET programs compiled to Microsoft Intermediate Language (MSIL), which is then translated into machine code and executed. There is a Linux port developed by the Mono project, but .NET is still primarily used for Windows development. As the saying goes “If you are not programming for Windows, .NET is not your cup of tea”.

1.1 C#

It is an event driven, object oriented, visual programming language with roots in C, C++ and Java which was developed at Microsoft by Anders Hejlsberg *et al.* before being integrated into the .NET platform. What this gives is the ability to access your programs through any device which supports the CLR, distribute web-based applications through the ASP. Net platform and communicate with other .NET compatible computer languages. Another advantage of C# over other contemporary languages is the strong integration into one of the best Integrated Design Environments (IDE) ever made, i.e., Visual Studio. Although the role of an IDE in language preference can be debated, you cannot disagree with the fact that Visual Studio makes programming and debugging fast and easy and helps you with Rapid Application Development (RAD).

Some of the advantages of using C# over other languages are:

Its supports so-called safe internet programming or managed code which restricts your programs by sand-boxing them but makes sure that your computer/OS is not affected adversely by poorly written code - no core dump or dead console.

- It is simpler than other object-oriented languages (C++).
- It has a very good graphics package and an excellent designer in the Form of Visual Studio and Expression Studio.
- It is related to C and C++, and therefore easy to pick up and start

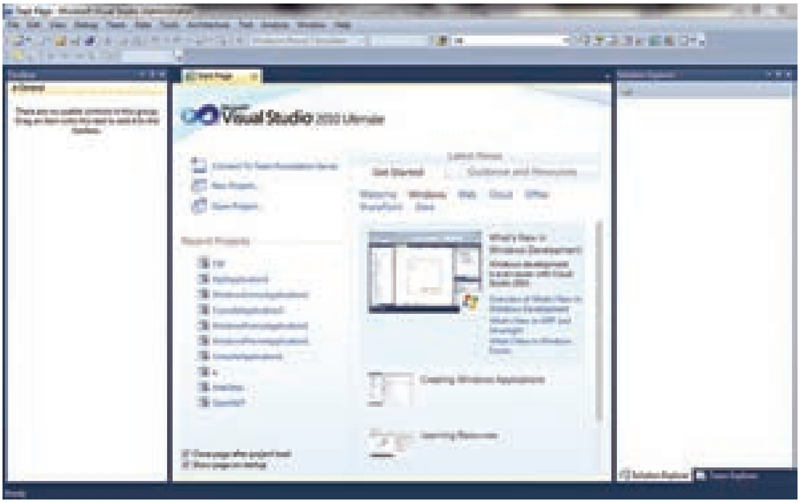
development instead of wasting countless hours trying to get the intricacies of the language.
- It has a very good client-server and network support.

The C# compiler translates C# source code (.cs files) into a special representation called Microsoft Intermediate Language (MSIL), which is not the machine language for any traditional CPU, but a virtual machine. The Common Language Runtime (CLR) then interprets the MSIL file by using a just-in-time compiler to translate from MSIL format to machine code on the fly.

1.2 Hello World

Let's start by creating a very simple "Hello World" C# console application. Our app will simply display the text "Hello World" on the screen whenever it is executed. A console application refers to any application that runs in a command shell within the Windows environment. Using the .NET framework, we can also create windows applications, web services, and components.

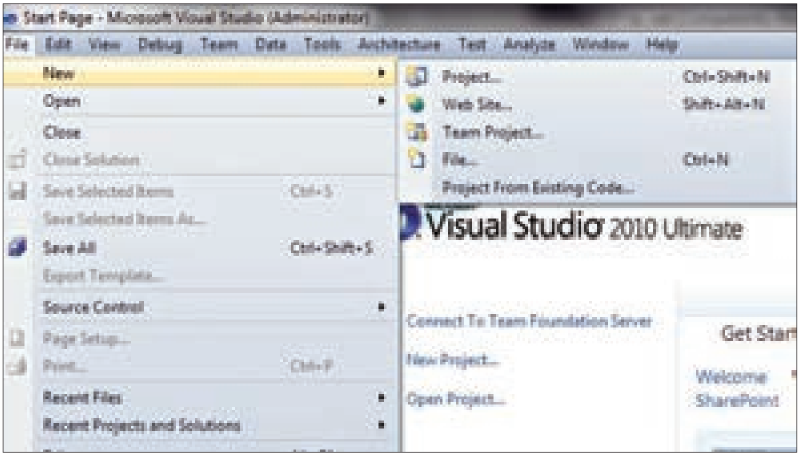
For any programmer who has learned a compiler language, the “Hello World” application is the typical starting point for getting a rough idea of the structure, syntax, look and feel of a C# application. We will also explain how to compile the application once we’ve created it. C# is a language similar to the old school C++ language, except Microsoft has added several namespaces and classes available for web and desktop applications.



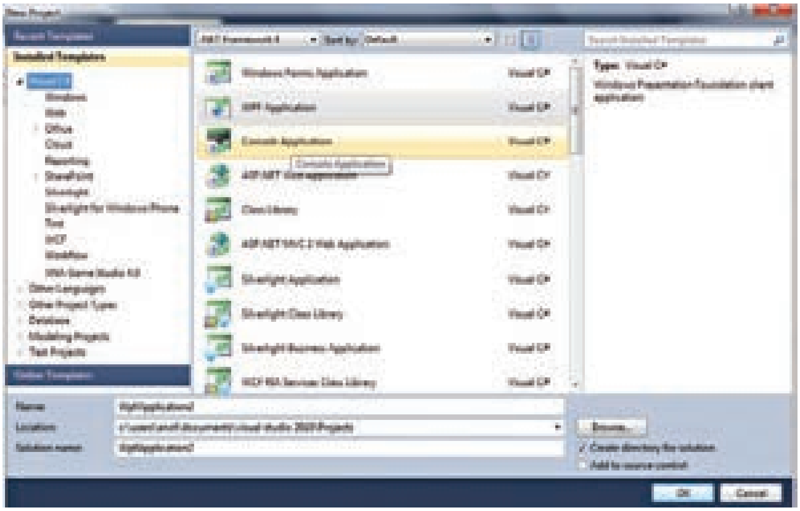
Visual Studio Interface

To get started, open Visual Studio. Once Visual Studio is loaded, you can create a new application. To create a new application, follow the steps mentioned below.

First, click on the "New" menu item and select "Project."


Start Page - Microsoft Visual Studio (Administrator)
File Edit View Debug Team Data Tools Architecture Test Analyze Window Help
New
Open
Close
Close Solution
Save Selected Items Ctrl+S
Save Selected Items As...
Save All Ctrl+Shift+S
Export Template...
Source Control
Page Setup...
Print... Ctrl+P
Recent Files
Recent Projects and Solutions
Project... Ctrl+Shift+N
Web Site... Shift+Alt+N
Team Project...
File... Ctrl+N
Project From Existing Code...
Visual Studio 2010 Ultimate
Connect To Team Foundation Server
New Project...
Open Project...
Get Star
Welcome
SharePoint

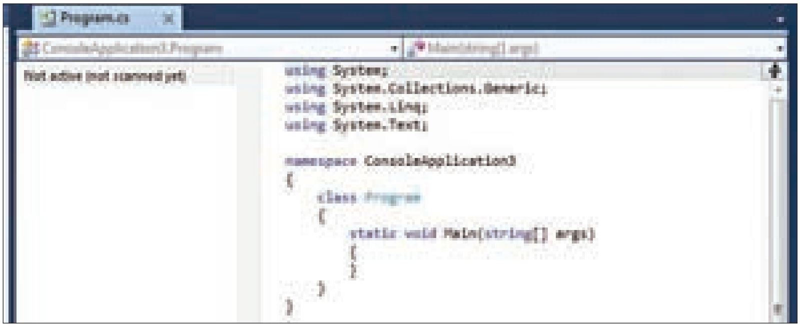
Click on Project to begin



Enter your project name

Select “Visual C#” and then click on “Windows.” Select “Console Application” from the list of project types. Enter a name and location for the project at the bottom of the window.

Once these values are set, the .NET project is created. There are a few important parts of the Visual Studio console. First, the main window is

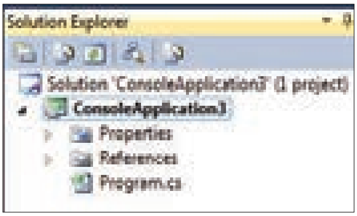


```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication3
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

You can design forms, web pages and create code

where the developer designs forms, web page and creates code.

The Solution Explorer displays the list of files included in the project. Underneath the Solution Explorer is the properties window where the developer can visually set each basic property for the controls created on the form or web page.



You can modify the properties for each control

C programmers recognize the “Main” function shown after the new console app is created. Just like C++, the Main function is the first functioned called. For this console application, variables, and simple integer to string conversions are made. The first step for this C# training is creating a variable. C# variable declarations are similar to C. The following code creates a string and an integer variable:

```
string myString;
int myInt;
```

Next process in this console application is to assign each variable a value. The developer can initialise the variable with a value, but to make the instruction simpler, they are assigned later. The following code is how to assign variables in C#:

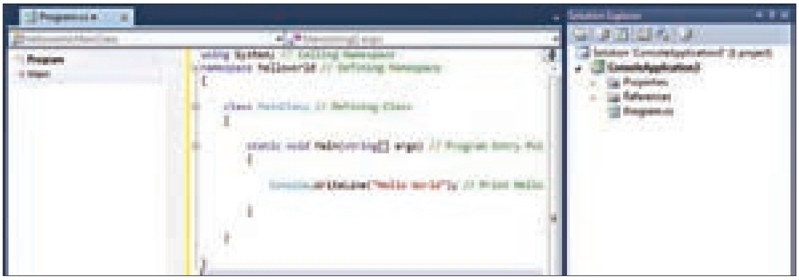
```
myString = “Hello World”;
myInt = 1;
```

Now it’s time to print the variables to the console. C programmers are familiar with the `printf` function. In C#, the information is printed to the console using the `Console.WriteLine()` function. There is also a `Console.Write` function, but the `WriteLine` function automatically adds a carriage return/line feed to the end of the output. The code `Console.WriteLine(myString + “. This is my “ + myInt.ToString() + “st program.”);` writes the output to the console.

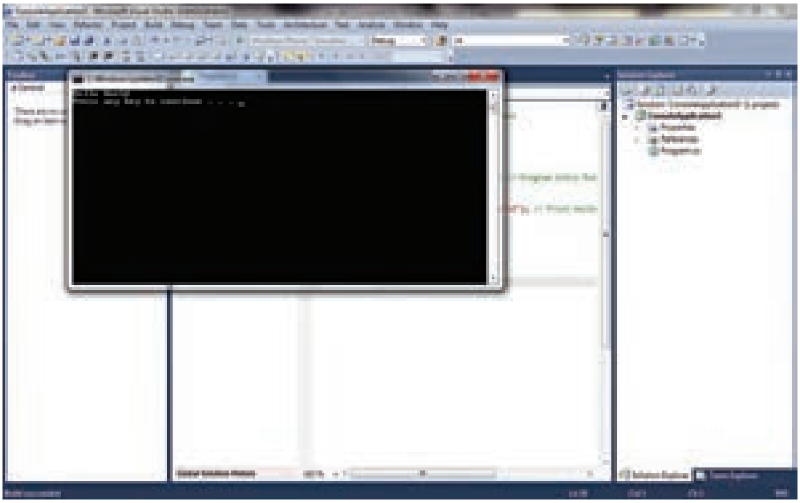
The output for this application is `Hello World. This is my 1st program.` Notice the `ToString()` implementation. This can be used with any object or variable to instantly convert it to a string to print it to output.

The complete code should look similar to:

```
using System; // Calling Namespace
namespace helloworld // Defining Namespace
{
class MainClass // Defining Class
{
static void Main(string[] args) // Program Entry Point
{
Console.WriteLine ("Hello World"); // Print Hello World
}
}
}
```


You can choose to run the program in debug mode

Save the file and press `[Ctrl]+[F5]`. `[F5]` runs the code in debug mode, but running console applications means the Windows command prompt runs the application and then closes it instantly when it's finished. `[Ctrl]+[F5]` stops the application without closing the window when it's finished. The developer should see the results in the command prompt.


If everything goes well, your screen should look like this

Congratulations, you have just created your first working C# console application! Now that we know how to create, compile and execute a C# application, let's take a look at a detailed explanation of our C# source code file.

The HelloWord sample we created is fairly elementary, yet precisely outlines the structure of a simple C# application:

```
using library_name;
namespace namespace_name
{
class class_name
{
static void Main([string[] args])
{
// Application code goes here
}
}
}
```

Let's take a look at what each these lines mean in relation to our C# application as a whole:

```
using library_name;
```

C# is based on the concept of using namespaces. Namespaces separate code into individually named-containers and avoids the chance of having duplicate variable names within the scope of a block. The "`using`" command allows us to specify which libraries namespaces we want access to. "using" commands are in no way similar to C++'s "#include" directive, and do not physically include any other source code into our file, they simply give us access to the namespaces available in the libraries that we have specified.

The "System" library contains the namespaces for all common .NET variables such as System.Int32 (which represents an integer) amongst other things.

Note: For any language to be compatible with .NET, it must provide implementations of the base .NET variable types. These include Int32, Int64, etc, C# provides its own version of these base variables in a form that resembles C++'s variable types: ushort, int, string, etc.

If we reference the System library with a `using` command such as `using System;` and want to create a new integer variable, then we don't need to explicitly declare `Int32` as part of the System namespace, because we already have access to the system libraries namespace:

```
Int32 myNum = 10;
```

On the other hand, if we don't include a `using` command to reference the system library, then we would have to explicitly specify which namespace the Int32 class resides in, like this:

```
System.Int32 myNum = 10;
```

Although .NET contains several pre-defined namespaces, we can create our own using a namespace block, like this:

```
namespace myNS
{
System.Int32 x = 10;
}
```

This would create a new namespace named "myNS" under the global section of our C# application. To reference the integer variable `x` from within our new `myNS` namespace, we would use the following syntax:

```
myNS.x;
```

To access variables, objects, or other namespaces (namespaces can be nested) from within a namespace, you call the namespaces in the order that they are nested and separate each one with a dot operator. If we have three nested namespaces like this:

```
namespace ns1
{
namespace ns2
{
namespace ns3
{
System.Int32 number;
}
}
}
```

... then we can retrieve the value of the "number" integer with the following command:

```
ns1.ns2.ns3.number;
```

Although a namespace isn't required in our sample application, it helps to physically differentiate our code from any other code that might be present in the same source code file.

```
class class_name
{
}
```

Just like in C++, C# uses classes to facilitate its object-orientated design methodology. Classes allow us to create individual self-contained units of code, which can be instantiated throughout our application. All C# applications must contain at least one class. If we wanted to create a class named "myClass", then we would use the following commands:

```
class myClass
{
// Methods and members for this class go here
}
```

All C# applications must have a `Main()` function, which is the entry point for execution of the application:

```
static void Main([string[] args])
{
// Application code goes here
}
```

As you can see, the `string[] args` argument list is optional, and can be used to access the parameters passed to your executable from the command line. The `Main()` function is declared as public by default and can return either an integer variable type or nothing, with "void". To explicitly specify our `Main()` function as public and to return an integer variable, we would use the following `Main()` function:

```
public static int Main()
{
}
```

1.3 C# language features

The first and most confusing point for any Visual Basic developer to remember is that C# is case sensitive. If you typed `console` instead of `Console` for example, then the compiler would throw out an error message as follows:

```
   It's not only commands or function names that are case
sensitive. Variables are also case sensitive, meaning that
the variable "a" is different to the variable "A". Because
of this case sensitivity, certain naming guidelines have
been developed to aid developers when naming functions,
variables, etc, and Microsoft recommends that you adhere
to these guidelines.
```

Firstly, variables should be named to help with intrinsic documentation. For example, if we want to create a variable to deal with how many apples we have, then we would call that variable `numApples` and not something like `a`. Secondly, variables whose names consist of more than one word (such as `databaseName`) should use the camelCase naming convention. The camelCase naming convention dictates that when naming multiple-word variables, every letter should be lower case, accept for the first letter of each new word, which should be capitalised. Some camelCase examples include `myName`, `yourStreetNum`, and `timeLeftToGo`.

Secondly, in the C# source code, white spaces are ignored. C++ and Java developers have enjoyed this for years. All commands are terminated with a `;` instead. This helps us when formatting and stylising our source code, letting us make it as readable as possible.

Thirdly, C# is a block structured language, meaning that all statements are part of a block of code. Each block of code is opened and closed by using curly braces: `{` and `}` respectively.

Lastly, C# supports three different types of comments. These will appear common to current C++ developers. For one line comments, you use the // syntax, like this:

```
// This is a comment
```

One line comments cannot span multiple lines, so the comment shown below would cause the C# compiler to flag an error:

```
// This is the first line of the comment
and this is the second ERROR line
```

If you need to add a comment to your source code which spans more than one line, you can use the "`/* ... */`" syntax, like this:

```
/*
This is a multi-lined comment
and will not raise an error at all
*/
```

The last type of comment that C# supports is the triple-slashed comment. This is a special type of comment that allows us to internally document our code. It is a single line comment that can be used by the .NET compiler to create useful documentation for your application as follows: `/// Important comment`.

1.4 Creating a WPF (Windows Presentation Foundation) application

The WPF designer in Visual Studio 2008 makes it very easy to create a simple, yet fully functional WPF application.

To create a new WPF application, follow these steps:

1. Open Visual Studio 2008.
2. On the File menu, click on New, and then click on Project.
3. In the New Project dialog box, in Templates, click WPF Application or WPF Browser Application.
4. In the Name box, type a name for the application, and then click OK. The WPF designer opens, showing Window1 of the project you created.

When you create an application, Visual Studio 2008 automatically creates a custom Application-derived class for you. In this class, you specify the startup page or window by using the StartupUri property and you also specify any application-level resources. Visual Studio 2008 also generates a XAML application file, for both stand-alone and browser-based applications, that specifies the following information:

- The code-behind class for the application.
- The startup window or page.
- Application-wide resources.

1.5 Defining windows and pages

Stand-alone applications and browser-based applications look very similar; the only obvious difference between them is that the stand-alone application contains a Window element and an XBAP contains a Page element. A Window is a top-level window, whereas a Page is a browser-based page. WPF stand-alone applications can contain windows or pages, but WPF browser applications can only contain pages. WPF browser applications use a different navigation style and are typically limited in their access to system resources.

1.5.1 Window

```
<Window
    xmlns:x="..."
    xmlns"=..."
    x:Class="MyApp.Window1" Title="My Window">
  <Grid>
      ...
  </Grid>
</Window>
```

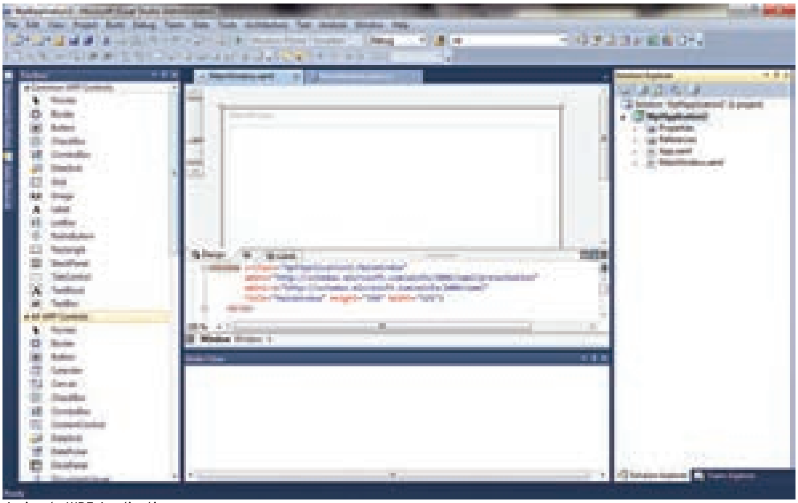
1.5.2 Page

```
<Page
    xmlns:x="..."
    xmlns"=..."
    x:Class="MyApp.Page1" WindowTitle="My Page">
  <Grid>
      ...
  </Grid>
</Page>
```

1.6 Adding controls to an application

In WPF, "control" is an umbrella term that applies to a category of WPF classes that are hosted in either a window or a page, have a user interface, and implement some behaviour. WPF ships with a library of the common UI components that are used in almost every Windows-based application, such as Button, Label, TextBox, Menu, and ListBox.

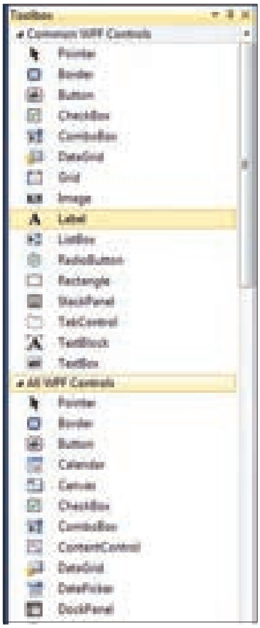
When you use Visual Studio 2010, you can see what WPF controls are available for use by looking in the toolbox. You can add controls by dragging the control from the toolbox and dropping it onto your


A simple WPF Application

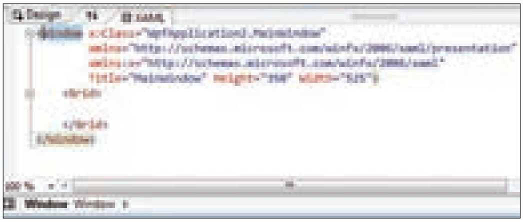
window or page. You can use the same controls in both stand-alone applications and browser-based applications.

The following XAML code example shows how to create a TextBox control and a Button control that are similar to the controls that are illustrated in the image:

```
   <Grid>
      <TextBox Name="TextBox1" />
          <Button   Name="Button1">Click
here</Button>
   </Grid>
```



WPF includes a library of common UI components



```
<Window x:Class="WpfApplication1.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <Grid>

    </Grid>
</Window>
```

You can also define controls explicitly by using XAML or imperative code

1.7 Creating applications

You can create two types of WPF applications in Visual Studio: stand-alone applications and browser-based applications. To create a stand-alone WPF application, you use the WPF application template when you create the project.

Visual Studio uses this template to create a fully functional, stand-alone WPF application that contains one window. Next you see the design view of the window and the XAML representation of the same window. The XAML specifies the class that contains the code for the window, the XML schemas that the XAML markup uses, and the properties of the window, such as the title and size.

Visual Studio also adds a Grid control to your window by default. This control acts as a container for the other controls that you place in the window and determines where they appear. You can add controls to a WPF application by dragging the control from the toolbox onto your window or page.

To create a browser-based WPF application, you use the WPF Browser Application template when you create the project. The design environment is almost identical for both application types. The most notable difference between the two applications is that the browser application contains a Page element, whereas the stand-alone application contains a Window element.

You can add controls to browser-based applications in exactly the same way that you add controls to a Windows-based application. The XAML that is generated when you add a control is almost the same in both applications.

1.8 Handling events and commands

You add user interactivity to your applications by using buttons and other controls, and you respond to interactions with these controls in your applications by implementing handlers and commands.

You generate events in WPF in the same way as you would with any .NET Framework technology, such as Windows Forms or Microsoft ASP.NET, by clicking buttons, entering text, changing list selections, setting control focus, or any other similar activity.

In the WPF event model, there are two ways to specify event handlers:

Imperative approach. You can specify event handlers imperatively in code, as you can by using ASP.NET or Windows Forms.

Declarative approach. You can specify the event handler method declaratively by using XAML. The declarative approach is similar to the approach that is taken by ASP.NET in that it enables you to define event handlers in markup language. However, you must provide all of your event handler implementations imperatively by using managed code, in a file that is known as the code-behind file.

To handle WPF control events, you must first specify the event handler for the control in the XAML file and then implement the event handler in Visual Basic .NET, Visual C#, or any other language that is supported by the .NET Framework, in the code-behind file.

1. Specify the event handler in the XAML file

The following code example specifies an event handler for the Click event of a button in the XAML file.

```
<Button Name="Button1" Click="Button1_Click">
Click here
</Button>
```

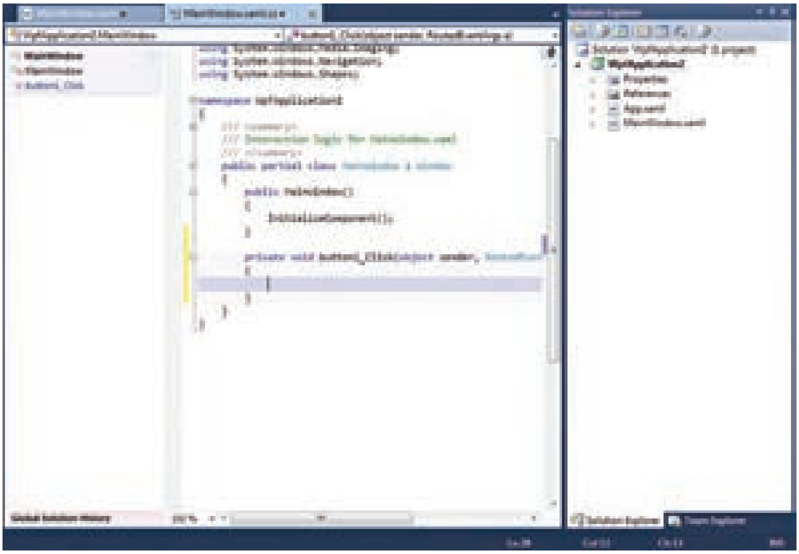
2. Implement the event handler in the code-behind file

The following code examples implement an event handler for the Click event of a button in the code-behind file.

```
public void Button1_Click(object sender, RoutedEventArgs
e)
{
MessageBox.Show("Hello WPF");
}
```

When you create an application that has a user interface, your application uses events to respond to the actions of the user. When the user interacts with an element in the user interface, for example, by clicking a button or selecting a value, that element fires an event. In your application code, you can create methods that are executed every time a particular event is fired. In this way, the actions of the user drive the behavior of your application.

In WPF applications, user interface elements are organized in a hierarchical structure that is known as the element tree. The Window or Page element is at the top of the element tree. The Window element can include one or more container controls, such as a StackPanel, and the StackPanel can include controls such as buttons. In this case, the button controls are at the bottom of the element tree. Routed events can propagate up or down the element tree. There are three types of routed events.



```
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;

namespace WpfApplication2
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, RoutedEvent
        {
            
        }
    }
}
```

Double-click event handler

1. Bubbling events travel up the element tree from the originating element to each parent element in turn.

2. Tunneling events are initially passed to the top element in the element tree, then travel down the tree until they reach the originating element.

3. Direct events are fired only for the originating element and are not passed up or down the tree.

You can take advantage of routed events to write code efficiently. For example, rather than adding an event handler for the Click event to every button within a StackPanel, you can add a single event handler for button Click events to the StackPanel itself. When a user clicks a button within the StackPanel, the Click event is routed up the element tree and handled by the StackPanel. This is an example of a bubbling event. Your event handler method can examine the event arguments to determine which button fired the event, and respond accordingly.

You can use tunneling events to intercept the actions of the user. For example, suppose that your StackPanel is read-only in certain circumstances. When the user tries to click a button on the StackPanel, you can intercept the PreviewMouseLeftButtonDown tunneling event at the StackPanel. If the StackPanel is set to read-only, you can mark the PreviewMouseLeftButtonDown event as handled. As a result, the PreviewMouseLeftButtonDown event is ignored by the originating button, the Click event is never fired, and the user is prevented from clicking the button.

2 The C# object system

2.1 Object-oriented principles

Before we talk about the object system C#, it's good to reflect on the desired properties of object-oriented code (and how they are realised in the language): polymorphism, encapsulation, and extensibility.

Polymorphism deals with code operating over multiple types. Generics embody this principal. For example, the generic `List<T>` class from the .NET library is equivalent to `ArrayList<T>` from the Java class libraries. `List<T>` stores a list of objects of type T (where T is any type) in a (growable) array.

Another example is sorting a list. As long as the contained class defines some sort of comparison function between objects of its type, any reasonable sort function should work on it. In Java, such a class must implement the Comparable interface and in C#, they implement the `IComparable<T>` interface.

```
class IntPair : IComparable<IntPair>
{
    public int X { get; private set; }
    public int Y { get; private set; }
    public IntPair(int x, int y)
    {
        X = x;
        Y = y;
    }

    /// <summary>
    /// We compare pairs element-wise.
    /// </summary>
    public int CompareTo(IntPair other)
    {
        if (X < other.X)
        {
            return -1;
        }
        else if (X > other.X)
        {
           return 1;
```

```
            }
            else if (Y < other.Y)
            {
                return -1;
            }
             else if (Y > other.Y)
            {
                return 1;
            }
            else
            {
                return 0;
            }
        }
         public void Test()
        {
          IntPair[] arr = new IntPair[] { new IntPair(3, 4),
new IntPair(5, 1),
                                         new IntPair(1, 7), new
IntPair(5, 3) };
             Array.Sort(arr);
             foreach (IntPair p in arr)
             {
                 Console.WriteLine("{0}, {1}", p.X, p.Y);
             }
         }
  }
```

Encapsulation deals with exposing a public interface to clients of a class while hiding the details of the implementation of that interface. Properties embody this principle precisely. Finally, extensibility deals with being able to extend code beyond what was originally planned. Inheritence and dynamic dispatch allow us to accomplish this. Consider the entities in a video game. To realise this in an object- oriented language, we might design an abstract Entity object.

```
  public abstract class Entity
  {
      public abstract void Move();
      / ...and other methods/partial implementation /
  }
```

Now to add a new type of Entity object, all we need to do is create a new class that extends Entity and provide an implementation for Move. This is a win for extensibility we do not have to modify the Entity class to add additional classes. However, if we want to add new functionality to Entity (e.g., a Draw method), we need to add that method to Entity and each class that derives from Entity.

Being an object oriented (OO) language steeped in the C-tradition, C# tackles these problems in the same way as Java. However, it does so in a more refined way with particular emphasis on common problems encountered with Java. Keep in mind these properties as you encounter new features in C#'s object system and think about how they enforce these properties (or not!).

2.2 The basics of C#

Syntactically, the basics of C# are derived from the C++ way of doing OO rather than Java. So the syntax may look unfamiliar if you have not programmed in C++, but the basic ideas are the same. Like Java, C# is a single-inheritance language with a unified type system. That is, all types derive from a common class, `System.Object` in both cases. Here is a series of Java class definitions and the equivalent in C#.

Java Implementation:

```
    public abstract class Employee {
        private int yearsEmployed;
        public Employee(int yearsEmployed) {
            this.yearsEmployed = yearsEmployed;
        }
                public  int  getYearsEmployed()  {  return
yearsEmployed; }
        pubiic String getTitle();
    }

    public class Tech extends Employee
        private String speciality;
        public Tech(string speciality, int yearsEmployed)
{
            super(yearsEmployed);
            this.speciality = speciality;
        }
        public String getTitle() { return "Grunt"; }
    }

    public class CEO extends Employee {
        public CEO(int yearsEmployed) {
            super(yearsEmployed);
        }
        public int getYearsEmployed() {
            return super.getYearsEmployed() + 1000;
        }
        public String getTitle() { return "Boss"; }
    }

    public class EmployeeTest {
        public static void DoTest() {
            Employee e1 = new Tech("computers", 5);
            Employee e2 = new CEO(1);
              System.out.println(e1.getTitle() + ", " +
e1.getYearsEmployed());
              System.out.println(e2.getTitle() + ", " +
e2.getYearsEmployed());
        }
    }
```

C# Implementation

```
  public abstract class Employee
  {
      int yearsEmployed;
      public Employee(int yearsEmployed)
      {
          this.yearsEmployed = yearsEmployed;
      }
      public abstract string GetTitle();
      public virtual int GetYearsEmployed()
      {
          return yearsEmployed;
      }
  }
  public class Tech : Employee
  {
      string speciality;
      public Tech(string speciality, int yearsEmployed)
          : base(yearsEmployed)
      {
          this.speciality = speciality;
      }
      public override string GetTitle() { return "Grunt";
}
  }

  public class CEO : Employee
  {
      public CEO(int yearsEmployed) : base(yearsEmployed)
{ }
      public override int GetYearsEmployed()
      {
          return base.GetYearsEmployed() + 1000;
      }
      public override string GetTitle()
      {
          return "Boss";
      }
  }

  public static class EmployeeTest
  {
      public static void DoIt()
      {
          Employee e1 = new Tech("computers", 5);
          Employee e2 = new CEO(1);
            Console.WriteLine("{0}, {1}", e1.GetTitle(),
e1.GetYearsEmployed());
            Console.WriteLine("{0}, {1}", e2.GetTitle(),
e2.GetYearsEmployed());
      }
  }
```

2.3 Differences to note with C#

1. The default access modifier of class members is private.
2. Instead of saying extends/implements, the class name and the list of classes/interfaces it extends/implements is separated with a colon(ala C++).
3. Java allows subclasses to override any of its visible methods. C# is more strict in the interests of encapsulation. Parent classes must mark overridable methods as virtual (ala C++).
4. C# requires that subclasses explicitly mark overriden methods with overrides.
5. Instead of super() as the first call in a constructor, C# uses C++-like initialiser syntax to invoke superclass constructors.
6. Instead of super to invoke methods of the base class, C# uses super to the same effect.
7. C# has a notion of a static class not present in Java that restricts said class

Lots of little differences to note but the fundamental idea is the same (and indeed the two sets of classes perform identically).

2.4 Hiding class members with new

As mentioned previously, we must mark methods in subclasses that override virtual methods in parent classes with overrides. If we wish to instead hide/replace a method (or any inherited members) in a parent class with a subclass version, we use new.

```
   public class NewTest
  {
   public class Parent
     {
      // Note: F is not virtual so it subclasses cannot
override it
        public void F() { Console.WriteLine("Parent"); }
       }
       class Child : Parent
       {
           // Hide Parent's F with our own F
         public new void F() { Console.WriteLine("Child");
}
       }
  }
  #endregion
```

2.5 Assemblies and access modifiers

Assemblies are the fundamental unit of packaging in C# (and the .NET framework in general). An assembly is a collection of C# classes that form a program (if a class contains a Main method) or a library (if no Main method exists). The analog to this in Java is the Jar file which houses collections of .class files.

Access modifiers reflect assemblies as the unit of packaging in C#:

- public - visible to everyone
- protected - visible to self + all subclasses
- internal - visible to self + all classes in the same assembly
- protected internal - like internal but also visible to all subclasses (even if they are in different assemblies)
- private - visible to self

This is slightly more complicated than Java, but the fundamental rule is the same. Expose as little of your members to as little clients as possible.

3 Value types

3.1 Structs

In Java, the type system distinguishes between primitive types and objects. For example, the memory semantics of the types

```
int x = 0;
```

Point p = new Point<int, int>(0, 0)(where Point is a typical tuple of ints) are different in two ways.

1. int has value semantics whereas p has reference semantics. Imagine we have the following two methods that operate over ints and Points.

public static void changeInt(int x) { x = 5; }
public static void changePoint(Point p) { p.X = 5; }

If we call changeInt(x) and then inspect the value of x afterwards, we will find that the value of x is 0. This is because we pass x by-value, that is we pass a copy of x, to changeInt. If we call changePoint(p) and then inspect p.x, we'll see that p.x is now 5.This is because we pass p by-reference, that is we pass a reference of p, to changePoint. Thus we are actually modifying p vs. a copy of it.

2. x is allocated on the stack and contains exactly the bits of the integer whereas p is allocated on the heap and contains a reference to that memory in the heap. Also, the memory allocated on the heap contains other data other than the components of the pair, e.g., a reference to the vtable of Point class.

Furthermore, since x is allocated on the stack, its memory is "released" when the method that allocated x is returns. Since p is allocated on the heap, its memory is not released until the garbage collector cleans up.

The distinction is important in understanding the costs involved with allocating lots of small objects, For example, we may allocate lots of Point objects in doing GUI or graphics that will require many allocations on the heap. Furthermore, these short-lived objects will need to be garbage collected eventually which will incur more cost.

Ideally, we would like to treat Point as a primitive type where we avoid heap allocations, but Java does not provide any way to do this have this facility. C# allows us to create such types, called value types, via the struct construct.

```
public struct Vec2D
{
        public int X { get; private set; }
        public int Y { get; private set; }
        // Note: we must invoke the default constructor of
a struct before
        // assigning to its fields.
        public Vec2D(int x, int y) : this()
        {
            X = x;
            Y = y;
        }
    }
```

Like C++, a struct defines something that looks like a class. However, unlike C++, struct is not a synonym for class. A struct is treated like a primitive type in Java with respect to memory semantics. This comes with some restrictions.

3. structs cannot extend any other classes. They can, however, implement any number of interfaces. They also cannot be extended by any other structs or classes (i.e., they are automatically sealed).

4. structs automatically define their own default (no-parameter) constructor that initialises all fields to their default values. You cannot change this constructor.

5. Because structs are not references, they cannot be assigned null. However, sometimes you want to be able to assign null to a struct as a way of saying "nothing", e.g., an option type. To do so, you can mark a variable with the type of the struct as nullable.

```
    public class NullableTest
    {
        public void Test ()
        {
                // The "?" denotes v should really be
Nullable<Vec2D>
            Vec2D? v;
            v = new Vec2D(0, 0);
            v = null;
        }
    }
```

Thus, structs are very useful for "plain old datatypes" , PODs, that are simple, small collections of fields that should be treated in a lightweight manner and do not require all of the flexibility that classes provide. In fact, all primitive datatypes of C# are actually implemented as structs underneath the covers. For example, the declaration of an integer `int x = 42;` really declares x as value type System.Int32.

3.2 Boxing and unboxing

`int` is a value type. Thus its representation in memory is exactly the bits for the integer and nothing more. However, we can write code like this:

```
public class IntAsObjectTest
{
    public void Test()
    {
        // 3 calls to ToString for ints
        int x = 42;
        Console.WriteLine("{0}", x);
        x.ToString();
        42.ToString();
    }
}
```

That is, we can treat ints as objects even though they're value types. A more striking example is pre-generic containers such as ArrayList. In order to be "generic", non-generic ArrayLists can only hold references to objects. However, we can do add ints to ArrayLists as if they were objects.

```
public class ArrayListIntTest
{
    public void Test()
    {
        ArrayList l = new ArrayList();
        l.Add(42);
    }
}
```

Both of these things are possible are due to boxing. There are many times when you want treat a value type as an object. Boxing wraps a copy of the value type in a wrapper reference class that can call methods or be treated as a reference type. As seen above, this happens automatically.

From the description above, it is clear that boxing can be a costly operation (relative to other primitive operations) if done repeatedly. For example, consider the following code:

```
interface IFoo
{
    void DoIt();
}

struct Foo : IFoo
{
    public void DoIt() { }
}

class FooTest
{
    public static void Test()
    {
        Foo f = new Foo();
        DateTime before = DateTime.Now;
        for (int i = 0; i < 1000000; i++)
        {
            f.DoIt();
        }
        DateTime after = DateTime.Now;
        Console.WriteLine(after - before);

        before = DateTime.Now;
        for (int i = 0; i < 1000000; i++)
        {
            IFoo fi = f;
            fi.DoIt();
        }
        after = DateTime.Now;
        Console.WriteLine(after - before);
    }
}
```

We call an empty method of Foo through Foo in the first loop and through Foo's interface IFoo in the second loop. In the second loop, the assignment off to fi forces a boxing off since fi has type IFoo.

Thus method outputs:

```
00:00:00
        00:00:00.0468915
```

which shows that boxing and calling the method takes significantly more time than just calling the method. In general, we want to avoid boxing when (reasonably) possible to avoid these costs.

The opposite conversion, unboxing is also possible. Although, unboxing only occurs explicitly through a cast. This is because in an unboxing operation, we are casting from a more general type, e.g., object or an interface type, to the more specific struct type. This cast may fail so the programmer must state their intentions appropriately.

```
public class UnboxTest
{
    public static void Test()
    {
        int x = 42;
        object o = x;
        x = ((int)o) + ((int)o);
    }
}
```

3.3 Operator overloading, indexers, and conversions

As you've seen, C# allows operator overloading for user-defined types. This has the benefit of increasing readability as long as the operators are overloaded in a reasonable manner.

```
struct Vec
{
    public int X { get; private set; }
    public int Y { get; private set; }
    public Vec(int x, int y) : this()
    {
        X = x;
        Y = y;
    }

    /// <summary>
    /// Standard vector addition.
    /// </summary>
    public static Vec operator +(Vec left, Vec right)
    {
        return new Vec(left.X + right.X,
                        left.Y + right.Y);
    }

    /// <summary>
    /// Standard vector subtraction.
    /// </summary>
    public static Vec operator -(Vec left, Vec right)
    {
        return new Vec(left.X - right.X,
                        left.Y - right.Y);
    }

    /// <summary>
    /// Standard vector (dot) multiplication.
    /// </summary>
    public static int operator (Vec left, Vec right)
    {
        return left.X   right.X + left.Y   right.Y;
    }

    /// <summary>
    /// Standard scalar multiplication (vec on left)
    /// </summary>
    public static Vec operator (Vec left, int right)
    {
        return new Vec(left.X   right, left.Y   right);
    }

    /// <summary>
    /// Standard scalar multiplication (vec on right)
    /// </summary>
    public static Vec operator (int left, Vec right)
    {
        return right   left;
    }

    /// <summary>
    /// Vector negation.
    /// </summary>
    public static Vec operator -(Vec vec)
    {
        return new Vec(-vec.X, -vec.Y);
    }
}
```

Note the shape of the overloaded operator methods. They are declared public static and take the expected number of arguments.

You can overload a number of unary and binary operators:

```
Unary: +, -, !, ~, ++, --, true, false
Binary: +, -, , /, %, &, |, ^, <<, >>, ==, !=, >, <,
>=, <=
```

The restrictions you might expect apply for certain operators, e.g.,

Overloading ++ on a type T requires that you return a `T`. If you overload != you must also overload ==.

See the MSDN documentation for more information on these restrictions. In addition to operators, you can overload array notation to provide list-like indexing into your classes. These are called indexers.

```
struct Pair<T, U>
{
    public T Fst { get; private set; }
    public U Snd { get; private set; }
    public Pair (T t, U u) : this()
    {
        Fst = t;
        Snd = u;
    }
}

class SimpleMap<T, U>
{
    List<Pair<T, U>> list;
    public SimpleMap()
    {
        list = new List<Pair<T, U>>();
    }

    /// <summary>
    /// Indexer for the map.  Takes Ts and returns Us.
    /// </summary>
    public U this[T t]
    {
        get
        {
            foreach (Pair<T, U> p in list)
            {
                if (p.Fst.Equals(t)) { return p.Snd; }
            }
            return default(U);
        }
        set
        {
            foreach (Pair<T, U> p in list)
            {
                if (p.Fst.Equals(t))
                {
                    list.Remove(p);
                    break;
                }
            }
            list.Add(new Pair<T, U>(t, value));
        }
    }
}
```

Finally, like C++, we can specify how to convert from one type to another via conversion operators. Note conversion operators must be marked implicit or explicit. Explicit conversions must be signified by casts.

```
    public struct MyType
    {
        private int X { get; set; }
        public MyType(int x) : this()
        {
            X = x;
        }
        /// <summary>
        /// Implicitly converts from ints to MyTypes.
        /// </summary>
        public static implicit operator MyType(int x)
        {
            return new MyType(x);
        }
        /// <summary>
        /// Explicitly converts from MyTypes to ints.
        /// </summary>
        public static explicit operator int(MyType x)
        {
            return x.X;
        }
    }
    public class MyTypeTest
    {
        public static void Test()
        {
            int x = 10;
            MyType m = new MyType(5);

            MyType n = x;
            int y = (int) m;
        }
    }
```

3.4 Partial classes and extension methods

Finally, the most recent C# features have given programmers more flexibility in extending classes. The first of these features is partial classes which allow you to split the definition of a class over multiple files.

One use of partial classes is to solve the class design issues. We can get exhausiveness checking of a class hierarchy with the code locality of like-behavior by using partial classes to break up classes into different files.

```
// in Entity.cs
abstract partial class Entity { / ... / }
```

```
  partial class Spaceship : Entity { / ... / }
  partial class Asteriod : Entity { / ... / }
  // in Draw.cs
  abstract partial class Entity { void Draw(); }
  partial class Spaceship : Entity { void Draw() { / ...
/ } }
  partial class Asteriod : Entity { void Draw() { / ...
/ } }
```

Another use of partial classes you'll see is stowing away automatically generated code in a separate partial class definition. This way you can modify an auto-generated class safely in another file. This is how the WinForms designers work. Note that partial methods are also possible. Such methods define their signature in one file and the implementation in another. This is more useful to code generators rather than programmers.

Finally, a powerful extensibility feature of C# are extension methods. Recall that the String class has no Reverse method. The String class itself does not define Reverse; it is instead provided as an extension method (in System.Linq). Here is our version.

```
  public static class MyStringExtensionMethod
  {
      public static string MyRev(this string s)
      {
          string ret = "";
         // Note: I should use StringBuilder here instead
of simple strings
          // to avoid unnecessary allocations.
          foreach (char c in s)
          {
              ret = c + ret;
          }
          return ret;
      }

      public static void Test()
      {
         Console.WriteLine("this is my string".MyRev());
      }
  }
```

Recall that the method call `"this is my string".MyRev()` is like a function call with `"this is my string"` as the first parameter (except privileged in the dynamic dispatch sense). Thus the definition of MyRev (with the "this"-annotated parameter) is just syntatic suger for such a call.

4 Generics

4.1 The basics

Frequently you need to write the code that operates the same way over any type. An example from last chapter was the Pair struct. For example, here are several implementations of Pair over different types

```
public struct IntPair
{
    public int Fst { get; private set; }
    public int Snd { get; private set; }
    public IntPair(int fst, int snd) : this()
    {
        Fst = fst;
        Snd = snd;
    }
}

public struct StringPair
{
    public string Fst { get; private set; }
    public string Snd { get; private set; }
    public StringPair(string fst, string snd) : this()
    {
        Fst = fst;
        Snd = snd;
    }
}

public struct MixedPair
{
    public int Fst { get; private set; }
    public string Snd { get; private set; }
    public MixedPair(int fst, string snd) : this()
    {
        Fst = fst;
        Snd = snd;
    }
}
```

Clearly this approach does not scale well. Generics solve this problem by allowing us to substitute type parameters for these types. Now users of the Pair struct must provide the types that the Pair will hold (i.e.,the user instantiates the generic with particular types).

```
public partial struct Pair<T, U>
{
 public T Fst { get; private set; }
 public U Snd { get; private set; }
 public Pair(T fst, U snd) : this()
 {
     Fst = fst;
     Snd = snd;
 }
  public static void Test()
 {
  Pair<int, string> p = new Pair<int, string>(0, "hi");
}
}
```

Classes, interfaces, methods, and delegates can all be declared generic. In particular, you can declare a method to be generic inside of a generic class if you need to mention additional type parameters in the method.

```
public static class NestedGenerics<T>
{
    public static void GenericMethod<U>(T t, U u) { }
}
```

One particular use of generic methods is to be able to write a method over a generic class, say a collection. You do not wish to instantiate the type variable of the collection because your method should work for all possible collections. Generics methods let us tackle this problem.

```
public static class GenericMethod
{
     public static void ProcessCollection<T>(ICollectio
n<T> c) { }
}
```

4.2 Generic Constraints

Inside of a generic declaration, you don't know what types will be used to instantiate your type parameters. However, all types are subclasses of Object, so we can call methods of Objects on values of generic type. Note, in Java we must use wildcard type parameters to get the same effect.

```
  public partial struct Pair<T, U>
  {
        public override string ToString()
        {
              return "(" + Fst.ToString() + ", " + Snd.
ToString() + ")";
        }
  }
```

If we want to expand on the methods we can call on objects of generic type, we must refine our requirements on how we can instantiate our type parameters by constraints. For example, if we want to implent a sort function over a generic list, we'll need to require that the list contains elements that implement IComparable.

```
  public static class BubbleSort
  {
        public static void Sort<T>(IList<T> l) where T :
IComparable<T>
        {
              for (int i = 0; i < l.Count; i++)
              {
                    for (int j = i+1; j < l.Count; j++)
                    {
                          if (l[j].CompareTo(l[i]) < 0)
                          {
                                T temp = l[i];
                                l[i] = l[j];
                                l[j] = temp;
                          }
                    }
              }
        }
  }
```

We can constraint type parameters to implement a particular interface or extend a particular class. In addition to this, we can also restrict a type parameter to be instantiated only with a class or struct type. For example, Nullable<T> requires that T be a struct (i.e., you can only mark value types as nullable objects).

```
public class MyNullable<T> where T : struct
{
    public T Value { get; private set; }
    public MyNullable(T t)
    {
        Value = t;
    }
}
```

Frequently, we wish to create new instances of our type parameters. However if you think about it, this really isn't possible because classes may specify any number of parameters for their constructors. At best, we can hope that our candidate classes or structs implement a default constructor. Luckily, C# allows us to specify that our type parameter can only be instantiated by a type that has a default constructor. This is useful for implementing a generic factory method.

```
public static class GenericFactory<T> where T : new()
{
    public static T Make() { return new T(); }
}
```

4.3 Type Erasure

Observe that generics actually do not affect the execution of your program as all of their processing occurs during type-checking/compile time. Thus a reasonable question to ask is do we actually need to keep around information about the generic's type parameters at runtime?

Removing this information so that it does not appear at runtime is called type erasure. Java's generics are type erased at compile time and do not appear at runtime. Thus, two instantiations of the same generic, e.g., ArrayList<Integer> and ArrayList<String> appear to be the same object when viewed with reflection or other type-inspection tools. This was due to the desire for Java bytecode using generics to be backwards compatible with older versions of the JVM (this reason was quickly rendered moot as changes to the language broke backwards compatibility, but the implementation of generics remained the same).

How does this work? Here is a simplified generic implementation of an ArrayList and its corresponding type erased variant for a particular instantation.

```
public class GenericArrayList<T>
{
    private T[] Arr { get; set; }
    public GenericArrayList() { Arr = new T[10]; }
    public T Get(int i) { return Arr[i]; }
    public void Set(T t, int i) { Arr[i] = t; }

    public static void Test()
    {
   GenericArrayList<int> l = new GenericArrayList<int>();
        l.Set(10, 0);
        int x = l.Get(0);
    }
}

public class TypeErasedArrayList
{
    private object[] Arr { get; set; }
   public TypeErasedArrayList() { Arr = new object[10];
}
    public object Get(int i) { return Arr[i]; }
    public void Set(object t, int i) { Arr[i] = t; }

    public static void Test()
    {
      TypeErasedArrayList l = new TypeErasedArrayList();
        l.Set(/ Boxing from int32 to object / 10, 0);
        Int32 x = (Int32) l.Get(0);
    }
}
```

Since Object is the root of the class hierarchy, it is safe to substitute Object for all type parameters (constrainted type parameters are substituted for their constraint). To maintain the illusion that the code is generic, casts are inserted automatically. However, the implicit boxing (e.g., with int) and casts incur a performance hit. Furthermore, as evidenced above, type erasure disallows Java generics from being used in tandem with primitive types because of this translation. That is, this translation forces you to write ArrayList<Integer> instead of ArrayList<int>.

C# on the other hand does not erase generic type information. This avoids the need for translating source which leads to the inefficiencies described above. In addition, this allows you to reflect on generic types, something that Java does not allow due to type erasure.

```
  public class MyGenericClass<T>
  {
      public void PrintMyType() { Console.WriteLine(this.
GetType().FullName); }
      public static void Test()
      {
       MyGenericClass<int> c1 = new MyGenericClass<int>();
                    MyGenericClass<string>  c2  =  new
MyGenericClass<string>();
            c1.PrintMyType();
            c2.PrintMyType();
      }
  }
```

5 The .NET framework

Frequently, learning a language is less about syntax and more about getting a handle on the expansive libraries that the language provides. C# is no exception. Of course, having working knowledge of a library doesn't me know every method of every class. Rather, you should know the details of the components that you will use the most and know "the lay of the land" to find everything else quickly.

5.1 The layout of the .NET framework

First thing is where to look all this stuff up. For the MSDN documentation of the .NET framework, go to http://msdn.microsoft.com/en-us/library/ms229335.aspx. The .NET class libraries declare a hierarchy of namespaces that in turn contain the classes that we care about. At the top-level are two namespaces.

1. Microsoft which contains Microsoft-dependent technologies such as the support classes for the various Microsoft compilers, Windows-specific event handling, and tablet PC (ink) libraries.
2. System which contains the platform-agnostic technologies including the basic types, collections, IO, and networking. Our focus will be understanding the layout of System and some of the important classes in there.

5.2 The system namespace

At the root of the System namespace exists many important, general-purpose classes that you should know. These include

- Classes and structs corresponding to fundamental types (e.g., Int32, Char). These classes frequently contain static helper methods that help your process and convert data of those types.
- The Console class contains methods to manipulate the console. This goes beyond simple reading and writing. You can also manipulate the location of the cursor and other things.
- The Random class for random number generation.
- The Exception class and the basic exceptions that inherit from it. Note that unlike Java there is no notion of checked and unchecked exceptions. Checked exceptions must be caught by a callee or the enclosing method must declare that it throws a checked exception by a throws clause. All exceptions are unchecked (i.e., may not be caught) in C#.

```
    public static class NumberConversionExample
```

```
  {
     public static void Test()
     {
       string s1 = "42";
       string s2 = "not 42";
// Int32.Parse throws a System.FormatException if
//the string is not a number. In Java, the equivalent,
//NumberFormatException is a checked exception so we would
//be required to wrap the conversion in a try-catch block.
//In C#, there is no requirement to do so.
     int x1 = Int32.Parse(s1);
     int x2 = Int32.Parse(s2);
   }
  }
```

There are also general interfaces that you should be aware of in System.

- ICloneable provides a Clone method for deep-cloning of objects (vs. shallow or reference copying provided by `Object.MemberwiseClone`).
- IComparable<T> provides ordering between objects.
- IEquatable<T> provides equality checking between objects.
- IDisposable provides deterministic clean-up of resources. We'll revisit this when we talk about IO.

```
public class CloneableCell<T> where T : ICloneable
{
    public T Cell { get; private set; }
    public CloneableCell(T t) { Cell = t; }
    public CloneableCell<T> Clone()
    {
        return new CloneableCell<T>((T) Cell.Clone());
    }
}
```

5.3 The System.Collections namespace

This namespace is broken up into the pre-generic collections found in `System.Collections` and their generic variants in `System.Collections.Generic`. This can be confusing because if you introduce both namespaces into your file (as VS does by default for new code files), you'll see, e.g., `ICollection` and `ICollection<T>`. Just keep in mind which variant you are using, usually the generic variant.

5.3.1 ICollection

ICollection<T> is the parent class of all collections in the .NET framework. This interfaces provides several "essential" collection methods --- Add, Clear, Contains, etc. --- along with implementing IEnumerable and IEnumerable<T> which gives an iterator over a collection. Recall that instead of returning a class that implements IEnumerator we can instead define an iterator block for GetEnumerator.

```
  public class SkeletonCollection<T> : ICollection<T>
  {
      #region ICollection<T> Members

      public void Add(T item){ }
      public void Clear() { }
      public bool Contains(T item) { return false; }
      public void CopyTo(T[] array, int arrayIndex) { }
      public int Count { get { return 0; } }
      public bool IsReadOnly { get { return false; } }
      public bool Remove(T item) { return false; }

      #endregion

      #region IEnumerable<T> Members

     public IEnumerator<T> GetEnumerator() { return null;
}

      #endregion

      #region IEnumerable Members

        IEnumerator  IEnumerable.GetEnumerator()  {  return
null; }

      #endregion
  }

  #endregion
```

5.3.2 IList<T> and IDictionary<K, V>

The immediate sub-interfaces of ICollection are IList and IDictionary which specify the behaviour for list-like and dictionary-like collections. Notably, list collections can be accessed via indexers such as arrays. Dictionaries can also be accessed by indexers, but instead of integers as in the case of lists, dictionaries are indexed by their key type (the K in the generic).

The two concrete implementations of lists and dictionaries that are provided by the .NET framework are List and Dictionary. List is an implemented as an ArrayList, and Dictionary is implemented as a HashMap.

```
public static class ListDictTest
{
  public static void ListTest()
  {
     IList<int> list = new List<int>();
     list[0] = 5;
     list[3] = 8;
     list[25] = -2;
     foreach (int i in list) { Console.WriteLine(i); }
  }
   public static void DictionaryTest()
   {
   IDictionary<string, int> dict = new Dictionary<string,
int>();
     dict["John"] = 10;
     dict["Jane"] = 12;
     foreach (KeyValuePair<string, int> p in dict)
    {
       Console.WriteLine(p.Key + ", " + p.Value);
    }
 }
}
```

5.3.3 Stack, Queue, LinkedList, and others

In addition to lists and dictionaries, .NET provides more specialised collections. Stack and queue are obvious implementation of a stack and queue respectively. LinkedList and LinkedListNode are implementations of linked lists that deserves mention. LinkedLists are special in that they allow you to have a (read-only) handle on individual nodes of the list for O(1) insertion into the list at particular points.

```
public static class LinkedListTest
{
    public static void Test()
    {
        LinkedList<int> list = new LinkedList<int>();
        list.AddFirst(2);
        list.AddFirst(1);
        list.AddFirst(0);

        LinkedListNode<int> node = list.Find(1);
        list.AddAfter(node, 3);
        list.AddAfter(node, 4);

        foreach (int i in list)
        {
            Console.WriteLine(i);
        }
    }
}
```

Finally there are specialised collections in System.Collections.Specialized that contain...specialised types of collections for particular element types of containers or styles, e.g.,

- BitVector32 - stores compact representations of lists of bools and ints
- HybridDictionary - dictionary that uses an efficient ListDictionary implementation for small dictionaries and switches to a hash table for large dictionaries.

5.4 The System.IO namespace

5.4.1 Streams, Files, and Text

Like Java, C# maintains a hierarchy of stream objects that model different IO operations you'd like to perform. Most common of these operations is reading and writing a file to disk.

```
public static class FileTest
{
    public static void Test()
    {
            string path = @"C:\Users\posera\myfile.txt";
            if (File.Exists(path))
            {
            using (StreamWriter sw = File.CreateText(path))
                {
                    sw.WriteLine("Hello!");
                }
              using (StreamReader sw = File.OpenText(path))
                {
                    string line = "";
                   while ((line = sw.ReadLine()) != null)
                    {
                        Console.WriteLine(line);
                    }
                }
                string text = File.ReadAllText(path);
                Console.WriteLine(text);
            }
        }
    }
```

You can also create StreamReaders and StreamWriters over files directly via their constructors. Note that other operations may produce streams, e.g., network connections that you will use in a similar manner. In cases like a network connection where reducing system calls are important, BufferedStream is an available wrapper class to buffer read and write calls.

5.4.2 Deterministic finalisation and the IDisposable interface

The using block above is an example of deterministic finalisation. Recall that Java provides a notion of a finaliser for an object. The finaliser is theoretically responsible for cleaning up resources (e.g., fonts, files, network sockets) that a class utilises. However, the finaliser has no strong guarantees about when it will be called, just that it will be called sometime after the object is eligible to be garbage collected. This may work in some scenarios, but if you need to release that file or network socket immediately, you can't rely on a finaliser.

C# has a notion of finaliser, too, borrowing the syntax of C++ destructors. If that you are familiar with destructors, finalisers have very different semantics.

```
public class ExampleFinalizer
{
  public ExampleFinalizer() { Console.WriteLine("In ctor");
}
  ~ExampleFinalizer() { Console.WriteLine("In dtor"); }
    public static void Test()
    {
      for (int i = 0; i < 50000; i++)
       {
         new ExampleFinalizer();
       }
     }
  }
```

That being said, how do we provide deterministic finalisation? Resources typically provide some operation `close()` that "frees" the resource. Without additional language support, all we can do is make sure we call the method when we are done. Try-finally allows us to do this in a safe way, but it can become incredibly unwieldy.

```
  public static class ManualResourceTest
  {
   public static void Test()
    {
      StreamReader sw = null;
      try
       {
         try
          {
            sw = new StreamReader(@"C:\Users\posera\myfile.
txt");
             }
              catch (FileNotFoundException ex)
              {
                Console.WriteLine("File not found");
                 return;
               }
               try
               {
                Console.WriteLine(sw.ReadToEnd());
               }
            catch (IOException ex)
```

```
                    {
                      Console.WriteLine(
                       "An exception occurred reading from the
file");
                     return;
                   }
               }
                finally
               {
                if (sw != null) { sw.Close(); }
               }
            }
        }
```

This becomes even more complicated if our finally block also has logic that may throw an exception or fail. What we want is a way to ensure that we always close resources when we are done with them and no later.

C/C++ handles this via a design pattern, resource allocation is initialization (RAII). Since local variables live on the stack and are automatically deallocated when the containing method returns, RAII utilizes them (and the deterministic destructors of C++) to automatically allocate resources and then deallocate them when the local variable goes away.

This idiom is essentially what the using declaration above provides.

```
using (StreamWriter sw = File.Create(file)) { }
```

declares a "disposable" variable sw that allocates a resource, in this case a file at the beginning of the using block and then releases that resource when sw goes out of scope at the end of the block.

In order to be the subject of a using block, the class must implement the IDisposable interface. The resulting code is called the Dispose Pattern. See the following MSDN link for more information:

http://msdn.microsoft.com/en-us/library/fs2xkftw%28VS.80%29.aspx

```
public class DisposePatternTest : IDisposable
{
    public DisposePatternTest()
    {
        Console.WriteLine("In DiposePatternTest ctor");
    }

    #region IDisposable Members
```

```
        // Note Dispose is not overridable.  We want this
behavior for all
        // subclasses.
        public void Dispose()
        {
             Console.WriteLine("In Dispose()");
             Dispose(true);
              // We've already disposed this object; don't
call its finalizer
             GC.SuppressFinalize(this);
        }

        protected virtual void Dispose(bool disposing)
        {
             Console.WriteLine("In Dispose(bool)");
             if (disposing)
             {
                         Console.WriteLine("In Dispose(bool):
disposing");

                  // call dispose on managed components that
need it
                  // e.g., myField.Dispose()
             }
             // now free up that socket, file, etc.
        }

        ~DisposePatternTest()
        {
           Console.WriteLine("In DisposePatternTest dtor");
             Dispose(false);
        }

        #endregion

        public static void Test()
        {
                       using  (DisposePatternTest  t  =  new
DisposePatternTest()) { }
        }
    }
```

Now the semantics of the using block are clear. using ensures that when we leave the body of the using block that Dispose() is always called.

5.5 Other basic namespaces

Other namespaces you should keep in mind:

- System.Data contains database manipulation classes. In particular System.Data.Linq contains the support classes for LINQ.
- System.Diagnostics contains tools to help instrument your code for debugging and profiling.
- System.Net provides networking facilities.
- System.Xml provides XML services including serialisation mechanisms in C#.

6 Event-driven programming

In object-oriented programming, we model entities in the world as objects. A natural question is how do we model interaction between these objects. One way of modelling this behaviour is interpreting a method `call x.DoIt()` as passing object x the message `DoIt`. Another way is for objects to register interest in and react to events that other objects raise.

For example, consider a Button object on a Window. One event of interest is when the button is pressed. When this happens, the program may want to execute some logic such as closing the window. To do this, the Button object makes available the event, call it `OnPress`. The program then registers for that event with the Button. Now, when the Button is clicked, the Button notifies all objects that registered for the `OnPress` event.

This is commonly called the observer pattern: an Observable object is one whom has an event of interest to an Observer object. Java implements event-driven programming with these two interfaces.

6.1 Delegates

C#, on the other hand, provides explicit language support for events. The infrastructure begins with delegates. Delegates are a type that defines a method signature. By defining such a type, we can now pass methods around as objects.

```
  public delegate int BinOp(int x, int y);
  public static class BinOpTest
  {
      public static int Add(int x, int y) { return x + y; }
      public static int Sub(int x, int y) { return x - y; }

       public static void Test()
       {
          // Note either method of creating a BinOp works.
The second method
           // is a more recent version called method group
conversion.
           BinOp op1 = new BinOp(Add);
           BinOp op2 = Sub;
           Console.WriteLine(op1(1, 1));
```

```
            Console.WriteLine(op2(1, 1));
        }
    }
```

Delegate objects can reference not just one method, but multiple methods. These are referred to as multicast delegates. Note that if a delegate specifies a return value/out parameters that the result of the call is the result of the last method that is invoked.;

```
  public delegate void PrintStuff();
  public static class PrintStuffTest
  {
  public    static    void    PrintHello()    {    Console.
WriteLine("Hello"); }
  public    static    void    PrintWorld()    {    Console.
WriteLine("World"); }
   public static void PrintBang() { Console.WriteLine("!");
}
      public static void Test()
      {
         PrintStuff p = PrintHello;
         p += PrintWorld;
         p += PrintBang;
         p += PrintBang;
         p += PrintBang;
         p();
     }
  }
```

Delegates that contain only one method derive from `System.Delegate`. Likewise, delegates that contain multiple methods derive from `System.MulticastDelegate`. Delgates can also be made generic just like a concrete method.

```
  public static class GenericDelegateTest
  {
      delegate T Process<T>(T t);
      public static int Inc(int i) { return i+1; }
      public static void Test()
      {
          Process<int> p = Inc;
          Console.WriteLine(p(0));
```

```
        }
    }
```

One use of delegates is to provide predicates for methods. A predicate is a method that takes an argument, runs a test over it, and returns the result of that test. These tests tend to be small enough that defining a separate method in a class for them is too time-consuming. For this purpose, we have anonymous delegates and (as of C# 3.0) lambda expressions to fulfill that role. Note that these delegates have type Func<T1, ..., Tn, R> where Ti is the type of the parameters and R is the type of the return value. In the case of the void delegates, you use Func<T>.

```
  public static class AnonDelegateTest
  {
    public static void Test()
     {
        Func<int, bool> pos1 = delegate(int x) { return x
> 0; };
        Func<int, bool> pos2 = x => x > 0;
        Func<int, bool> pos3 = (x) => { return x > 0; };
        Console.WriteLine(pos1(1));
        Console.WriteLine(pos2(-1));
        Console.WriteLine(pos3(0));
    }
  }
```

Note that Func resides in System.Core.dll rather than mscorlib.dll. The latter is referenced by default so you need to add in System.Core.dll as a new reference to your project if you want to use the Func class.

Delegates are extremely handy as ways to parameterise functions by behaviour (e.g., send a sort function to a method or class) or for our purposes, providing callbacks for objects registering for an event. From our example above, a delegate corresponds to the function that the program wants the Button object to invoke when it is pressed. Thus, the process of registering for the event amounts to sending the Button object a delegate object to invoke.

6.2 Events

From our description above, we could implement events using multicast delegates without much problem. However, C# provides direct support for events as class members.

```
  public class EventBasics
  {
      public delegate void EventDelegate(EventBasics e);
      public event EventDelegate OnVarChange;
      int var;
      public int Var
      {
          get { return var; }
          set { var = value; OnVarChange(this); }
      }
        public void PrintVar(EventBasics e) { Console.
WriteLine(e.Var); }
      public static void Test()
      {
          EventBasics b = new EventBasics();
         // Note PrintVar is not static so we must access
it through an object
          // of type EventBasics; implicitly, "this" will
be this object.
          b.OnVarChange += b.PrintVar;
          b.Var = 15;
          b.Var = 20;
      }
  }
```

This code should look awfully familiar. If you have a good understanding of what delegates provide, it looks like that the event declaration is not adding any extra functionality. That is, we could replace the event member `OnVarChange` with a delegate field `OnVarChange` and get the same effect.

In some sense, events provide a similar abstraction between delegates and the notion of an event that properties provide between instance variables and the notion of field access. The event above automatically provides a backing field of type `EventDelegate` that powers the event. Alternatively, we can provide implementations of the essential functionality of the event abstraction: addding/registering and removing/unregistering.

In the example below, we create an event that double-adds the delegates passed to it. You can imagine writing a custom event that conditionally adds to different delegates or providing locking/synchronization support for an event.

```
  public class SpecializedEvent
  {
      public delegate void EventDelegate(SpecializedEvent
e);
      EventDelegate backingEvent;
      public event EventDelegate MyEvent
      {
           add { backingEvent += value; backingEvent +=
value; }
          remove { backingEvent -= value; }
      }
      protected void OnMyEvent()
      {
          backingEvent(this);
      }
       // Normally we would not expose a way to trigger
the event to clients;
      // this is purely for testing purposes.
      public void TriggerEvent() { OnMyEvent(); }
  }
  public class SpecializedEventTest
  {
      public static void EventDelTest(SpecializedEvent e)
      {
          Console.WriteLine("In Test");
      }
      public static void Test()
      {
          SpecializedEvent e = new SpecializedEvent();
          e.MyEvent += EventDelTest;
          e.TriggerEvent();
      }
  }
```

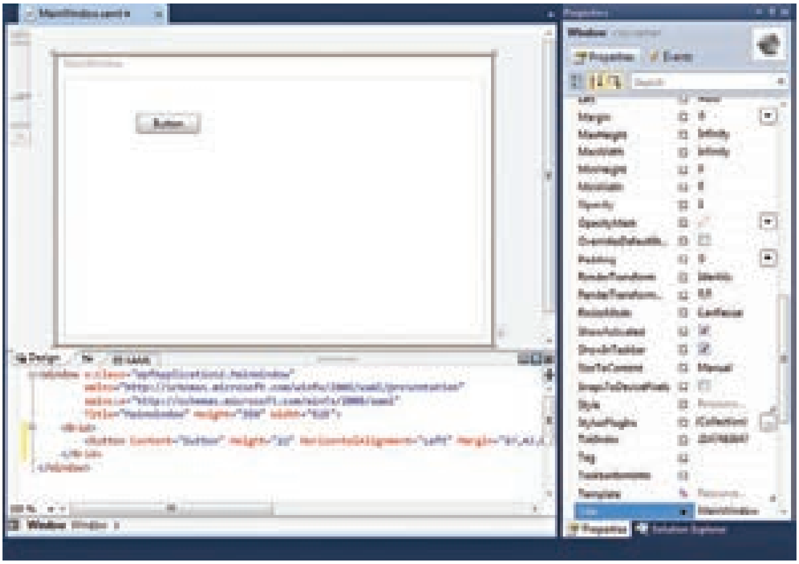
Note that we do not invoke the user-defined event directly when we want to trigger the event like with default events. We invoke the backing delegate instead. Both kinds of events do not expose a way to invoke the event outside of its containing class, even to inheriting subclasses. An idiom to provide this behavior is to declare a protected method OnX that triggers the X event, e.g., OnMyEvent above.

Furthermore, events declared for components that intend to operate with .NET framework such as a library are recommended to use delegates of the form:

`delegate void EventDelegate(object source, EventArgsChild e);` where `EventArgsChild` derives from the `System.EventArgs` class. The `System.EventHandler` delegate is available if your event handler does not need additional parameters. This is the style of events we will see inthe Winforms classes.

7 GUI

Winforms is the original API for implementing GUIs for the .NET framework. In some sense it is primitive because for the most part it merely wraps the Win32 APIs albiet in a cleaner, object-oriented fashion. We'll first look at how to create Winforms applications programmatically and then see how this approach compares to the Visual Studio designers.


Properties of a simple Window

7.1 Creating forms

In Winforms terminology, a form is a window of an application. To change the look of a form, you can set its properties, of which there are an extensive amount.

```
/
class SimpleForm : Form
{
     public SimpleForm()
     {
          Size = new Size(300, 400);
```

```
        Text = "My Simple Form";
        BackColor = Color.Black;
        Opacity = 0.5;
    }
}
class SimpleFormTest1
{
    public static void Test()
    {
      Application.Run(new SimpleForm());
    }
}
```

Note above that we run our Winforms applications via a call to the Application. Run static method that takes the form to show as input.

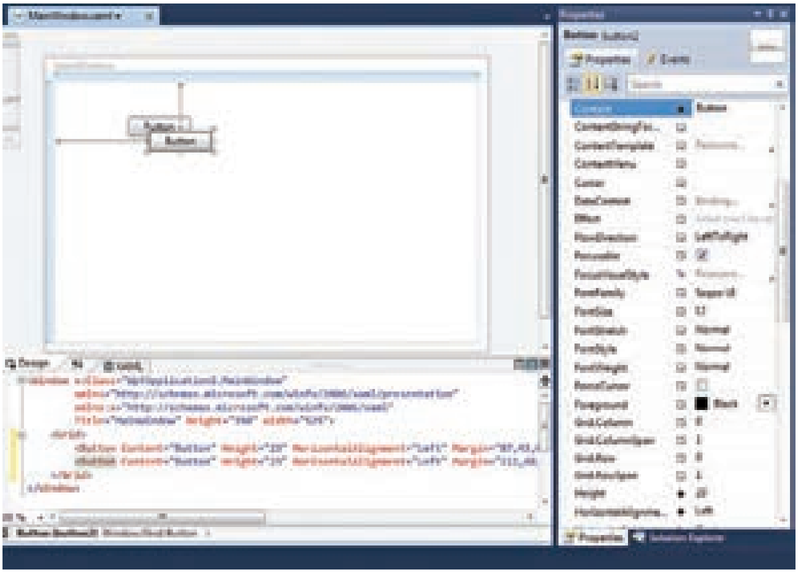
```
class SimpleFormTest2
{
    public static void Test()
    {
      SimpleForm s = new SimpleForm();
      s.Visible = true;
      Console.ReadLine();
    }
}
```

SimpleFormTest2's form seems to hang immediately. It turns out that Application.Run spins up a new thread to handle messages sent to our initial form. Since this thread is distinct from the main thread of execution, the form does not hang. Keep in mind that to start a forms application, you need to start a fresh message loop thread by Application.Run.

7.2 Adding controls and wiring events

Forms are also containers for other components such as buttons and labels.

```
public class AnotherSimpleForm1 : Form
{
    Button button;
    Label label;

    public AnotherSimpleForm1()
```



Adding a button

```
        {
            Size = new Size(300, 400);

            button = new Button();
            button.Size = new Size(200, 50);
            button.Location = new Point(50, 100);
            button.Text = "Press me!";

            label = new Label();
            label.Text = "My label";
            label.Location = new Point(100, 200);

            Controls.Add(button);
            Controls.Add(label);
        }

        public static void Test()
        {
            Application.Run(new AnotherSimpleForm1());
        }
    }
```

AnotherSimpleForm1 contains a button and a label, but right now the button does nothing. To force some behaviour, we must register for the button's click event.

```
  public class AnotherSimpleForm2 : Form
  {
      Button button;
      Label label;
      int timesPressed;

      public AnotherSimpleForm2()
      {
          timesPressed = 0;

          Size = new Size(300, 400);

          button = new Button();
          button.Size = new Size(200, 50);
          button.Location = new Point(50, 100);
          button.Click += button_Click;
          button.Text = "Press me!";

          label = new Label();
          label.Text = "Hello: " + timesPressed;
          label.Location = new Point(100, 200);

          Controls.Add(button);
          Controls.Add(label);
      }

      void button_Click(object sender, EventArgs e)
      {
          timesPressed += 1;
          label.Text = "Hello: " + timesPressed;
      }
```

```
        public static void Test()
        {
            Application.Run(new AnotherSimpleForm2());
        }
    }
```

Surprisingly enough, these are all to the basics of Winforms. Forms contain controls. The look of a component can be changed via properties. The behaviour of a component can be changed via events.

7.3 The Winforms designer

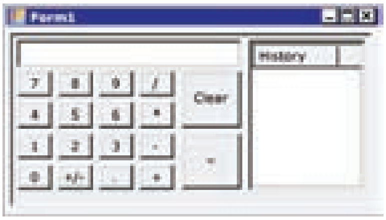
Up to this point, we've created forms programatially, that is, at run-time. However, it is usually the case that most of our UI is static and events define its dynamic behavior. To this end, we'd like to design UIs as a step separate from thec oompilation process, ideally in a designer tailor-made for UI development. The winforms designer built into VS is one such tool.

How does it differ from what we've learned? Here is the designer code verbatim for the simple winforms app we wrote above.

```
    #region Form1.cs (this is the code you can modify)
    public partial class Form1 : Form
    {
        int timesPressed;

        public Form1()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs
e)
        {
            timesPressed += 1;
            label1.Text = "Hello: " + timesPressed;
        }
    }
    #endregion

    #region Form1.Designer.cs (this is the code controlled
by the designer)
```

```
 partial class Form1
 {
     /// <summary>
     /// Required designer variable.
     /// </summary>
     private System.ComponentModel.IContainer components
= null;

     /// <summary>
     /// Clean up any resources being used.
     /// </summary>
     /// <param name="disposing">true if managed resources
should be disposed; otherwise, false.</param>
     protected override void Dispose(bool disposing)
     {
         if (disposing && (components != null))
         {
             components.Dispose();
         }
         base.Dispose(disposing);
     }

     #region Windows Form Designer generated code

     /// <summary>
     /// Required method for Designer support - do not
modify
     /// the contents of this method with the code editor.
     /// </summary>
     private void InitializeComponent()
     {
         this.button1 = new System.Windows.Forms.Button();
         this.label1 = new System.Windows.Forms.Label();
         this.SuspendLayout();
         //
         // button1
         //
           this.button1.Location = new System.Drawing.
Point(197, 227);
         this.button1.Name = "button1";
         this.button1.Size = new System.Drawing.Size(75,
23);
         this.button1.TabIndex = 0;
         this.button1.Text = "button1";
         this.button1.UseVisualStyleBackColor = true;
     this.button1.Click += new System.EventHandler(this.
button1_Click);
         //
         // label1
         //
         this.label1.AutoSize = true;
            this.label1.Location = new System.Drawing.
Point(116, 114);
         this.label1.Name = "label1";
         this.label1.Size = new System.Drawing.Size(35,
13);
         this.label1.TabIndex = 1;
         this.label1.Text = "label1";
         //
         // Form1
         //
         this.AutoScaleDimensions = new System.Drawing.
SizeF(6F, 13F);
            this.AutoScaleMode = System.Windows.Forms.
AutoScaleMode.Font;
         this.ClientSize = new System.Drawing.Size(284,
262);
         this.Controls.Add(this.label1);
         this.Controls.Add(this.button1);
         this.Name = "Form1";
         this.Text = "Form1";
         this.ResumeLayout(false);
         this.PerformLayout();

     }
```

```
        #endregion

        private System.Windows.
Forms.Button button1;
        private System.Windows.
Forms.Label label1;
    }
    #endregion
```

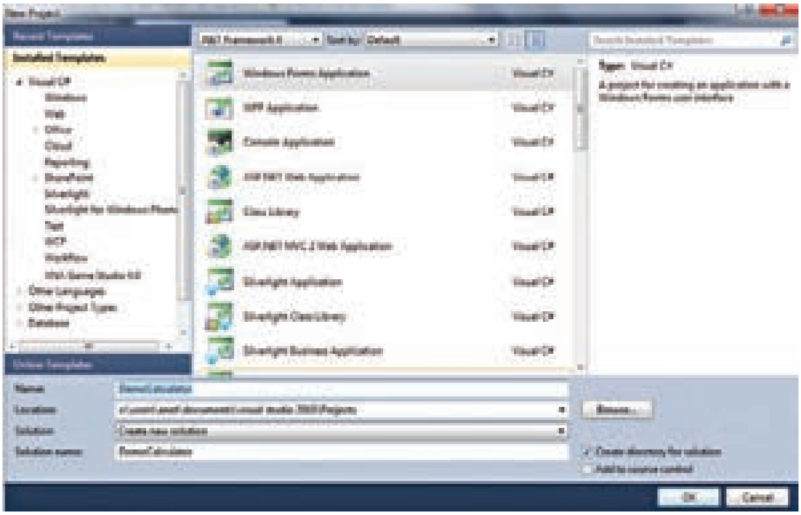


The demo calculator

As you can tell, besides code organisation, the designer creates roughly the same code that we would've produced manually. In other words, the programming model for winforms is the same as if you were doing things by hand or through the designer. The fact that there is no magic happening behind the covers is important when you're debugging designer-created applications. Also, now that you've seen this, you can be confident that by using the designers for most of your daily work that you are not missing out on much additional functionality. Let's see how to create something like this:

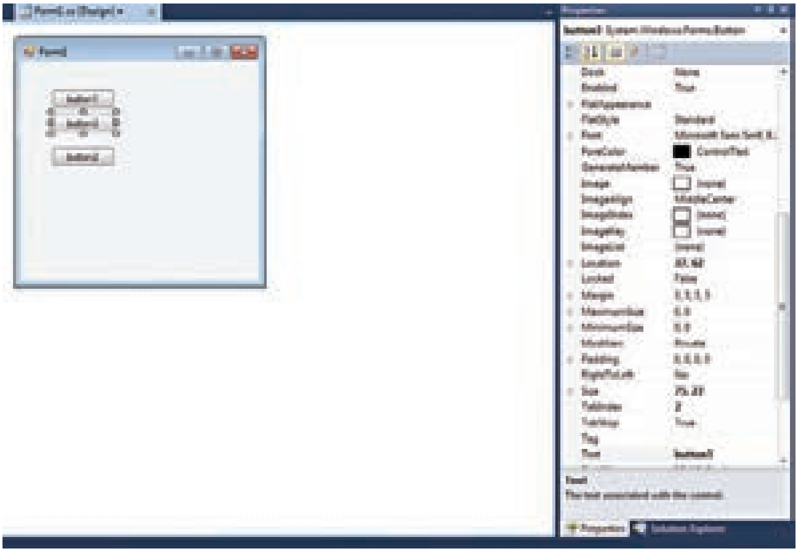
The first step is to create the DemoCalculator control project.

- On the File menu, click on New, and then click on Project to open the New Project dialog box.
- From the list of Visual Basic or Visual C# projects in the Windows



You can graphically design the interface of the Demo Calculator category, select the Windows Forms Control Library project template.

- In the Name box, type DemoCalculatorLib and then click OK.
- In Solution Explorer, right-click `UserControl1.vb` or `UserControl1.cs`, and then click Rename.
- Change the file name to `DemoCalculator.vb` or `DemoCalculator.cs`. Click the Yes button when you are asked if you want to rename all references to the code element "UserControl1".



button3 System.Windows.Forms.Button

Dock	None
Enabled	True
FlatAppearance	
FlatStyle	Standard
Font	Microsoft Sans Serif, 8...
ForeColor	ControlText
GenerateMember	True
Image	(none)
ImageAlign	MiddleCenter
ImageIndex	(none)
ImageKey	(none)
ImageList	(none)
Location	47, 68
Locked	False
Margin	3, 3, 3, 3
MaximumSize	0, 0
MinimumSize	0, 0
Modifiers	Private
Padding	0, 0, 0, 0
RightToLeft	No
Size	75, 23
TabIndex	2
TabStop	True
Tag	
Text	button3

Text
The text associated with the control.

Designing the control layout

The Windows Forms Designer currently shows the designer surface for the DemoCalculator control. In this view, you can graphically design the appearance of your control by selecting controls and components from the Toolbox and placing them on the designer surface.

7.3.1 Designing the Control Layout

The DemoCalculator control contains several Windows Forms controls. In this procedure, you will arrange the controls using some of the rapid application development (RAD) features of the Windows Forms Designer.

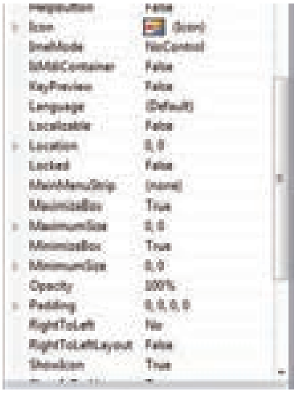
1. In the Windows Forms Designer, change the DemoCalculator control to a larger size by clicking the sizing handle in the lower-right corner and

dragging it down and to the right. In the lower-right corner of Visual Studio, find the size and location information for controls. Set the size of the control to a width of 500 and a height of 400 by watching the size information as you resize the control.

2. In the Toolbox, click the Containers node to open it. Select the SplitContainer control and drag it onto the designer surface.

The SplitContainer is placed on the DemoCalculator control's designer surface.

3. In the Properties window, change the value of the Dock property to None.



HelpButton	False
Icon	(Icon)
ImeMode	NoControl
IsMdiContainer	False
KeyPreview	False
Language	(Default)
Localizable	False
Location	0, 0
Locked	False
MainMenuStrip	(none)
MaximizeBox	True
MaximumSize	0, 0
MinimizeBox	True
MinimumSize	0, 0
Opacity	100%
Padding	0, 0, 0, 0
RightToLeft	No
RightToLeftLayout	False
ShowIcon	True

Set various parameters

The SplitContainer control shrinks to its default size. Its size no longer follows the size of the DemoCalculator control.

4. Click the smart tag glyph ( ) on the upper-right corner of the SplitContainer control. Click Dock in Parent Container to set the Dock property to Fill. The SplitContainer control docks to the DemoCalculator control's boundaries.

5. Click the vertical border between the panels and drag it to the right, so that most of the space in taken by the left panel. The SplitContainer divides the DemoCalculator control into two panels, with a movable border separating them. The panel on the left will hold the calculator buttons and display, and the panel on the right will show a record of the arithmetic operations performed by the user.

6. In the Properties window, change the value of the BorderStyle property to Fixed3D.

7. In the Toolbox, click the Common Controls node to open it. Select the ListView control and drag it into the right panel of the SplitContainer control.

8. Click the ListView control's smart tag glyph. In the smart tag panel, change the View setting to Details.

9. In the smart tag panel, click Edit Columns. The ColumnHeader Collection Editor dialog box opens.

10. In the ColumnHeader Collection Editor dialog box, click the Add button to add a column to the ListView control. Change the value of the column's Text property to History. Click OK to create the column.

11. In the smart tag panel, click Dock in Parent Container, and then click the smart tag glyph to close the smart tag panel.

12. From the Containers node in the Toolbox, drag a TableLayoutPanel control into the left panel of the SplitContainer control. The TableLayoutPanel control appears on the designer surface with its smart tag panel open. The TableLayoutPanel control arranges its child controls in a grid. The TableLayoutPanel control will hold the DemoCalculator control's display and buttons.

13. Click Edit Rows and Columns on the smart tag panel. The Column and Row Styles dialog box opens.

14. Click the Add button until five columns are displayed. Select all five columns, and then click the Percent option button in the Size Type box. Set the Percent value to `20`. This sets each column to the same width.

15. In the Show drop-down box, click Rows.

16. Click the Add button until five rows are displayed. Select all five rows, and the click the Percent option button in the Size Type box. Set the Percent value to `20`. This sets each row to the same height.

17. Click `OK` to accept your changes, and then click the smart tag glyph to close the smart tag panel.

18. In the Properties window, change the value of the Dock property to `Fill`.

7.3.2 Populating the control

Now that the layout of the control is set up, you can populate the DemoCalculator control with buttons and a display.

1. In the Toolbox, double-click the TextBox control icon. A TextBox control is placed in the first cell of the TableLayoutPanel control.

2. In the Properties window, change the value of the TextBox control's ColumnSpan property to `5`. The TextBox control moves to a position that is centered in its row.

3. Change the value of the TextBox control's Anchor property to Left, Right. The TextBox control expands horizontally to span all five columns.

4. Change the value of the TextBox control's TextAlign property to Right.

5. In the Properties window, expand the Font property node. Set Size to 14, and set Bold to true for the TextBox control.

6. Select the TableLayoutPanel control.

7. In the Toolbox, double-click the Button icon. A Button control is placed in the next open cell of the TableLayoutPanel control.

8. In the Toolbox, double-click the Button icon four more times to populate the second row of the TableLayoutPanel control.

9. Select all five Button controls by clicking them while holding down the

Shift key. Press `[Ctrl] + [C]` to copy the Button controls to the clipboard.

10. Press `[Ctrl] + [V]` three times to paste copies of the Button controls into the remaining rows of the TableLayoutPanel control.

11. Select all 20 Button controls by clicking them while holding down the Shift key.

12. In the Properties window, change the value of the Dock property to Fill. All the Button controls dock to fill their containing cells.

13. In the Properties window, expand the Margin property node. Set the value of All to 5. All the Button controls are sized smaller to create a larger margin between them.

14. Select button10 and button20, and then press `[Delete]` to remove them from the layout.

15. Select button5 and button15, and then change the value of their RowSpan property to 2. These will be the Clear and `=` buttons for the DemoCalculator control.

7.3.3 Navigating the Control Using the Document Outline Window

When your control or form is populated with several controls, you may find it easier to navigate your layout with the Document Outline window.

1. On the View menu, point to Other Windows, and then click Document Outline. The Document Outline window shows a tree view of the DemoCalculator control and its constituent controls. Container controls like the SplitContainer show their child controls as subnodes in the tree. You can also rename controls in place using the Document Outline window.

2. In the Document Outline window, right-click button1, and then click Rename. Change its name to sevenButton.

3. Using the Document Outline window, rename the Button controls from the designer-generated name to the production name according to the following list:

- button1 to sevenButton
- button2 to eightButton
- button3 to nineButton
- button4 to divisionButton
- button5 to clearButton
- button6 to fourButton
- button7 to fiveButton
- button8 to sixButton
- button9 to multiplicationButton

- button11 to oneButton
- button12 to twoButton
- button13 to threeButton
- button14 to subtractionButton
- button15 to equalsButton
- button16 to zeroButton
- button17 to changeSignButton
- button18 to decimalButton
- button19 to additionButton

4. Using the Document Outline and Properties windows, change the Text property value for each Button control name according to the following list:

- Change the sevenButton control text property to 7
- Change the eightButton control text property to 8
- Change the nineButton control text property to 9
- Change the divisionButton control text property to /
- Change the clearButton control text property to Clear
- Change the fourButton control text property to 4
- Change the fiveButton control text property to 5
- Change the sixButton control text property to 6
- Change the multiplicationButton control text property to *
- Change the oneButton control text property to 1
- Change the twoButton control text property to 2
- Change the threeButton control text property to 3
- Change the subtractionButton control text property to -
- Change the equalsButton control text property to =
- Change the zeroButton control text property to 0
- Change the changeSignButton control text property to +/-
- Change the decimalButton control text property to .
- Change the additionButton control text property to +

5. On the designer surface, select all the Button controls by clicking them while holding down the SHIFT key.

6. In the Properties window, expand the Font property node. Set Size to 14, and set Bold to true for all the Button controls. This completes the design of the DemoCalculator control. All that remains is to provide the calculator logic.

7.3.4 Implementing event handlers

The buttons on the DemoCalculator control have event handlers that can be used to implement much of the calculator logic. The Windows Forms

Designer enables you to implement the stubs of all the event handlers for all the buttons with one double-click.

1. On the designer surface, select all the Button controls by clicking them while holding `[Shift]`.

2. Double-click one of the Button controls. The Code Editor opens to the event handlers generated by the designer.

7.3.5 Testing the control

Because the DemoCalculator control inherits from the UserControl class, you can test its behavior with the UserControl Test Container.

1. Press `[F5]` to build and run the DemoCalculator control in the UserControl Test Container.

2. Click the border between the SplitContainer panels and drag it left and right. The TableLayoutPanel and all its child controls resize themselves to fit in the available space.

3. When you are done testing the control, click Close.

7.3.6 Using the control on a form

The DemoCalculator control can be used in other composite controls or on a form. The following procedure describes how to use it. The first step is to create the application project. You will use this project to build the application that shows your custom control.

1. On the File menu, point to Add, and then click New Project to open the New Project dialog box.

2. From the list of Visual Basic or Visual C# projects, select the Windows Forms Application project template.

3. In the Name box, type DemoCalculatorTest and then click `OK`.

4. In Solution Explorer, right-click the DemoCalculatorTest project, and then click `Add Reference` to open the Add Reference dialog box.

5. Click the Projects tab, and then double-click your DemoCalculatorLib project to add the reference to the test project.

6. In Solution Explorer, right-click `DemoCalculatorTest`, and then click `Set as StartUp Project`.

7. In the Windows Forms Designer, increase the size of the form to about `700 x 500`.

To use the DemoCalculator control in an application, you need to place it on a form.

1. In the Toolbox, expand the DemoCalculatorLib Components node.

2. Drag the DemoCalculator control from the Toolbox onto your form. Move the control to the upper-left corner of the form. When the control is close to the form's borders, you will see snaplines appear. These indicate the distance of the form's Padding property and the control's Margin property. Position the control at the location indicated by the snaplines.

3. Drag a Button control from the Toolbox and drop it onto the form.

4. Move the Button control around the DemoCalculator control and observe where the snaplines appear. You can align your controls precisely and easily using this feature. Delete the Button control when you are finished.

5. Right-click the DemoCalculator control, and then click Properties.

6. Change the value of the Dock property to Fill.

7. Select the form, and then expand the Padding property node. Change the value of All to `20`. The size of the DemoCalculator control is reduced to accommodate the new Padding value of the form.

8. Resize the form by dragging the various sizing handles to different positions. Observe how the DemoCalculator control is resized to fit.

We showed you how to construct the user interface for a simple calculator. You may want to extend its functionality in the following ways:

- Implement the calculator logic. This may seem straightforward, but there are interesting complexities associated with the calculator's state transitions.
- Package the calculator application for deployment.

8 Concurrency

There are many angles to understanding concurrency in computer programming. We won't explore them all (that's a class all on its own), but we'll look at concurrency in the context of creating responsive gui-based applications.

An RSS reader featuresan underlying engine or model that did the work of parsing RSS feeds. The GUI or view merely reflects the data contained in the model. Whenever the model updates, we update the view.

We assume that the process of updating the model was quick so that the view could simply wait for the model to update before doing its own work. However, what happens when the model takes a long amount of time to update?

```
  public class SlowForm : Form
  {
      Button button;
      TextBox input;
      public SlowForm()
      {
          Size = new Size(200, 200);

          input = new TextBox();
          input.Size = new Size(100, 25);
          input.Location = new Point(35, 50);

          button = new Button();
          button.Size = new Size(100, 25);
          button.Location = new Point(35, 100);
          button.Text = "Do Work";
         button.Click += new EventHandler(delegate(object
sender, EventArgs e)
               {
                   // Simulate some gnarly work to do by
a boring loop
                 for (int i = 0; i < 1000000000; i++) { }
                   MessageBox.Show("Done!");
               });
          this.Controls.Add(button);
```

```
            this.Controls.Add(input);
        }

        public static void Test()
        {
            Application.Run(new SlowForm());
        }
    }
```

While we do the button's work, the UI might become unresponsive. This is because the single thread of execution that processes events on our form is tied up executing the event handler for the button click. In these cases where we know our engine processing will take a long time, we would like to avoid hanging the UI by executing the engine work in a different thread.

The thread class in C# provides this facility. We can instantiate a new thread with a delegate (of type ThreadStart with signature `void()`) that the thread will run and then start it with the `Start()` method.

This thread operates concurrently with the original thread of execution. So now, this original thread of execution is able to continue processing events fired by the UI while the new thread handles the expensive computation.

```
    public class ConcurrentForm : Form
    {
        Button button;
        TextBox input;
        public ConcurrentForm()
        {
            Size = new Size(200, 200);

            input = new TextBox();
            input.Size = new Size(100, 25);
            input.Location = new Point(35, 50);

            button = new Button();
            button.Size = new Size(100, 25);
            button.Location = new Point(35, 100);
            button.Text = "Do Work";
           button.Click += new EventHandler(delegate(object
sender, EventArgs e)
            {
              Thread worker = new Thread(new ThreadStart(
                   delegate()
                   {
                        // Simulate some gnarly work to
do by a boring loop
                        for (int i = 0; i < 1000000000;
i++) { }
                       MessageBox.Show("Done!");
                   }));
               worker.Start();
           });
           this.Controls.Add(button);
           this.Controls.Add(input);
       }

       public static void Test()
       {
           Application.Run(new ConcurrentForm());
       }
   }
```

You can think of a thread as a sequence of instructions that can operate in parallel with other threads. Notice how we did not have to do anything more once we started the thread. This is because the runtime/operating system is responsible for scheduling when threads run. These context switches between threads occur quickly enough that even on a single-processor machine, we maintain the illusion that different threads operate in parallel.

Threads maintain their own copies of local variables, but objects stored on the heap (e.g., reference objects) are shared.

```
  public class BasicThreadExample
  {
       public static void Test()
       {
         Thread t1 = new Thread(new ThreadStart(PrintSeq));
           t1.Name = "t1";
         Thread t2 = new Thread(new ThreadStart(PrintSeq));
           t2.Name = "t2";
           t1.Start();
           t2.Start();
```

```
            // The original thread needs to join the two 
threads we create.  That
             // way this main thread does not end (and 
terminate the program) until
           // t1 and t2 are done.
           t1.Join();
           t2.Join();
           Console.ReadLine();
       }
       // Our static counter, shared by t1 and t2
       static int counter = 0;

       static void PrintSeq()
       {
           // t1 and t2 each have their own copy of i
           for (int i = 0; i < 100; i++)
           {
             Console.WriteLine(Thread.CurrentThread.Name +
                               ": {0}, {1}", i, counter++);
           }
       }
   }
```

There is not a straightforward way for threads to share local data because they each have their own copies. However, threads can both read and write to shared data on the heap as shown above with the counter variable. This should raise a red flag in your head: can conflicts arise between threads reading and writing from the same share variable? Definitely!

Here is the beginning of a sample run of BasicThreadExample:

```
   t1: 0, 0
    t1: 1, 2
    t1: 2, 3
    t1: 3, 4
    t1: 4, 5
    t2: 0, 1
    t2: 1, 7
    t2: 2, 8
    t2: 3, 9
    t1: 5, 10
```

Recall that the second number on each line is the content of our static counter. It seems a little odd that we've skipped over counter == 6. How could have this happened? Keep in mind that counter++" is really the following (conceptual) set of instructions:

```
  int temp = counter;
  counter = counter + 1;
  return temp;
```

Also recall threads are scheduled by the runtime/OS at will. So the following "bad" scheduling can happen:

```
    t2: pulls the value 1 from the counter and stores it
in temp
    t1: pulls the value 5 from counter and stores it in
temp
    t1: increments counter to 6
    t1: prints "t1: 4, 5"
    t2: increments counter to 7
    t2: prints "t2: 0, 1"
```

This is an example of a race condition between threads. We would like the counter++ set of instructions to execute all at once without being pre-empted. However, the runtime/OS is free to schedule threads any way it sees fit, thus leading to the anomaly.

Clearly this is just a toy example, but imagine if counter was not a simple counter but a bank account and the competing threads were the bank account owner and the bank doing operations on the account. In this case, race conditions like above are not acceptable; we need some way to ensure that only one thread "accesses" counter at a time.

C# provides this functionality through it's locking mechanism analogous to Java's synchronized construct.

```
  public class LockingThreadExample
  {
      public static void Test()
      {
        Thread t1 = new Thread(new ThreadStart(PrintSeq));
          t1.Name = "t1";
        Thread t2 = new Thread(new ThreadStart(PrintSeq));
          t2.Name = "t2";
          t1.Start();
          t2.Start();
           // The original thread needs to join the two
threads we create.  That
             // way this main thread does not end (and
terminate the program) until
          // t1 and t2 are done.
          t1.Join();
          t2.Join();

          Console.ReadLine();
      }
      // Value types can't be the subject of locking.  So
we have to
      // explicitly create a wrapper object.
      class IntWrapper
      {
          public int val = 0;
      }
      // Our static counter, shared by t1 and t2
      static IntWrapper counter = new IntWrapper();
      static void PrintSeq()
      {
          // t1 and t2 each have their own copy of i
          for (int i = 0; i < 100; i++)
          {
              lock(counter)
              {
                 Console.WriteLine(Thread.CurrentThread.
Name +
                                          ": {0}, {1}", i,
counter.val++);
              }
          }
      }
  }
```

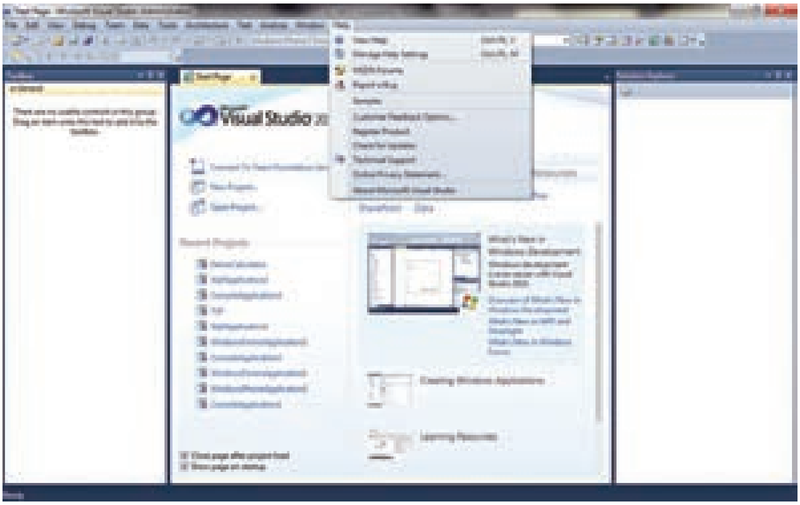
Conceptually a lock is a token that threads can acquire and release. If a thread tries to acquire a lock that another thread holds, then that thread waits until that lock is released before continuing. Every object has a lock that threads can acquire. Threads can acquire this through the lock construct. For the duration of the block, called a critical section, the thread holds the lock of the object releasing it at the end of the block. This allows for mutual exclusion over the counter variable: only one thread modifies or accesses counter at a time.

Note that locking leads to serialization of execution as threads wait for each other to release locks. There is a fine art to ensuring that just the right amount of locking is used: too little leads to race conditions and too much leads to serialized code which defeats the purpose of concurrency in the first place.

There are other issues that we won't go over due to time constraints. Two threads can end up waiting on locks that each other hold. This introduces a deadlock in your program.

Concurrency is a hot research area and developments and techniques are being constantly refined. Consider our discussion to be an introduction to the benefits and complexities of concurrency.

9 Appendix - Shortcut Key Guide



Save time with Keyboard shortcuts

This appendix describes keyboard shortcuts in the following categories:

- General
- Project related
- Window manipulation
- Text navigation
- Text manipulation
- Text selection
- Control editor (designer)
- Search and replace
- Help
- Debugging
- Object browser
- Tool window
- HTML designer
- Macro
- Dialog editor
- Accelerator and string editor

General

Command	**Shortcut**	**Description**
Edit.Copy	CTRL-C CTRL-INSERT	Copies the currently selected item to the system clipboard.
Edit.Cut	CTRL-X SHIFT-DELETE	Deletes the currently selected item and moves it to the system clipboard.
Edit.CycleClipboardRing	CTRL-SHIFT- INS CTRL-SHIFT-V	Pastes an item from the Clipboard Ring tab of the Toolbox at the cursor in the file and automatically selects the pasted item. You can cycle through the items on the clipboard by pressing the shortcut keys repeatedly.
Edit.GoToNextLocation	F8	Moves the cursor to the next item, such as a task in the TaskList window or a search match in the Find Results window.
Edit.GoToPreviousLoca-tion	SHIFT-F8	Moves the cursor to the previous item in the TaskList window or Find Results window.
Edit.GoToReference	SHIFT-F12	Finds a reference to the selected item or the item under the cursor.
Edit.OpenFile	CTRL-SHIFT-G	Opens the file whose name is under the cursor or is currently selected (e.g., if you use this shortcut in a C++ file when the cursor is on a line with a #include statement, it will open the file being included).
Edit.Paste	CTRL-V SHIFT-INSERT	Inserts the item in the clipboard at the cursor.
Edit.Redo	CTRL-SHIFT-Z CTRL-Y SHIFT-ALT-BACK-SPACE	Redoes the previously undone action.
Edit.SelectionCancel	ESC	Closes a menu or dialog, cancels an operation in progress, or places focus in the current document window.
Edit.Undo	ALT-BACKSPACE CTRL-Z	Reverses the last editing action.
File.Print	CTRL-P	Displays the Print dialog.
File.SaveAll	CTRL-SHIFT-S	Saves all documents and projects.
File.SaveSelectedItems	CTRL-S	Saves the selected items in the current project (usually which-ever source file is currently visible).
Tools.GoToCommandLine	CTRL-/	Switches focus to the Find/Command box on the Standard toolbar.
View.NextTask	CTRL-SHIFT-F12	Moves to the next task in the TaskList window.
View.PopBrowseContext	CTRL-SHIFT-8	Moves backward in the browse history. Available in the object browser or Class View window.
View.ViewCode	F7	Switches from a design view to a code view in the editor.
View.ViewDesigner	SHIFT-F7	Switches from a code view to a design view in the editor.
View.WebNavigateBack	ALT-LEFT ARROW	Goes back in the web browser history.
View.WebNavigateFor-ward	ALT-RIGHT ARROW	Goes forward in the web browser history.

Project-related

Command	Shortcut	Description
Build.BuildSolution	CTRL-SHIFT-B	Builds the solution.
Build.Compile	CTRL-F7	Compiles the selected file. C++ projects only–.NET projects do not support compilation of individual files, only whole projects.
File.AddExistingItem	SHIFT-ALT-A	Displays the Add Existing Item dialog.
File.AddNewItem	CTRL-SHIFT-A	Displays the Add New Item dialog.
File.BuildandBrowse	CTRL-F8	Builds the current project and then displays the start page for the project in the browser. Available only for web projects.
File.NewFile	CTRL-N	Displays the New File dialog. Files created in this way are not associated with any project. Use File.AddNewItem (Ctrl-Shift-A) to create a new file in a project.
File.NewProject	CTRL-SHIFT-N	Displays the New Project dialog.
File.OpenFile	CTRL-O	Displays the Open File dialog.
File.OpenProject	CTRL-SHIFT-O	Displays the Open Project dialog.
Project.Override	CTRL-ALT-INSERT	Allows you to override base class methods in a derived class when an overridable method is highlighted in the Class View pane.

Window manipulation

Command	Shortcut	Description
View.FullScreen	SHIFT-ALT-ENTER	Toggles full screen mode.
View.NavigateBackward	CTRL-+	Goes back to the previous location in the navigation history. (For example, if you press Ctrl-Home to go to the start of a document, this shortcut will take the cursor back to wherever it was before you pressed Ctrl-Home.)
View.NavigateForward	CTRL-SHIFT-+	Moves forward in the navigation history. This is effectively an undo for the View.NavigateBackward operation.
Window.ActivateDocumentWindow	ESC	Closes a menu or dialog, cancels an operation in progress, or places focus in the current document window.
Window.CloseDocumentWindow	CTRL-F4	Closes the current MDI child window.
Window.CloseToolWindow	SHIFT-ESC	Closes the current tool window.
Window.MoveToDropDownBar	CTRL-F2	Moves the cursor to the navigation bar at the top of a code view.
Window.NextDocumentWindow	CTRL-TAB CTRL-F6	Cycles through the MDI child windows one window at a time.
Window.PreviousDocumentWindow	CTRL-SHIFT-TAB CTRL-SHIFT-F6	Moves to the previous MDI child window.
Window.NextPane	ALT-F6	Moves to the next tool window.
Window.PreviousPane	SHIFT-ALT-F6	Moves to the previously selected window.
Window.NextSplitPane	F6	Moves to the next pane of a split pane view of a single document.
Window.PreviousSplitPane	SHIFT-F6	Moves to the previous pane of a document in split pane view.
Window.NextTab	CTRL-PAGEDOWN	Moves to the next tab in the document or window (e.g., you can use this to switch the HTML editor from its design view to its HTML view.
Window.PreviousTab	CTRL-PAGE UP	Moves to the previous tab in the document or window.

Text navigation		
Command	**Shortcut**	**Description**
Edit.CharLeft	LEFT ARROW	Moves the cursor one character to the left.
Edit.CharRight	RIGHT ARROW	Moves the cursor one character to the right.
Edit.DocumentEnd	CTRL-END	Moves the cursor to the end of the document.
Edit.DocumentStart	CTRL-HOME	Moves the cursor to the start of the document.
Edit.GoTo	CTRL-G	Displays the Go to Line dialog. If the debugger is running, the dialog also lets you specify addresses or function names to go to.
Edit.GoToBrace	CTRL-]	Moves the cursor to the matching brace in the document. If the cursor is on an opening brace, this will move to the corresponding closing brace and vice versa.
Edit.LineDown	DOWN ARROW	Moves the cursor down one line.
Edit.LineEnd	END	Moves the cursor to the end of the current line.
Edit.LineStart	HOME	Moves the cursor to the beginning of the line. If you press Home when the cursor is already at the start of the line, it will toggle the cursor between the first non-whitespace character and the real start of the line.
Edit.LineUp	UP ARROW	Moves the cursor up one line.
Edit.NextBookmark	CTRL-K, CTRL-N	Moves to the next bookmark in the document.
Edit.PageDown	PAGE DOWN	Scrolls down one screen in the editor window.
Edit.PageUp	PAGE UP	Scrolls up one screen in the editor window.
Edit.PreviousBookmark	CTRL-K, CTRL-P	Moves to the previous bookmark.
Edit.QuickInfo	CTRL-K, CTRL-I	Displays Quick Info, based on the current language.
Edit.ScrollLineDown	CTRL-DOWN ARROW	Scrolls text down one line but does not move the cursor. This is useful for scrolling more text into view without losing your place. Available only in text editors.
Edit.ScrollLineUp	CTRL-UP ARROW	Scrolls text up one line but does not move the cursor. Available only in text editors.
Edit.WordNext	CTRL-RIGHT ARROW	Moves the cursor one word to the right.
Edit.WordPrevious	CTRL-LEFT ARROW	Moves the cursor one word to the left.
View.BrowseNext	CTRL-SHIFT-1	Navigates to the next definition, declaration, or reference of an item. Available in the object browser and Class View window. Also available in source editing windows if you have already used the Edit.GoToReference (Shift-F12) shortcut.
View.BrowsePrevious	CTRL-SHIFT-2	Navigates to the previous definition, declaration, or reference of an item.

Text manipulation		
Command	**Shortcut**	**Description**
Edit.BreakLine	ENTER SHIFT-ENTER	Inserts a new line.
Edit.CharTranspose	CTRL-T	Swaps the characters on either side of the cursor. (For example, AC|BD becomes AB|CD.) Available only in text editors.
Edit.ClearBookmarks	CTRL-K, CTRL-L	Removes all unnamed bookmarks in the current document.

Edit.CollapseToDefinitions	CTRL-M, CTRL-O	Automatically determines logical boundaries for creating regions in code, such as procedures, and then hides them. This collapses all such regions in the current document.
Edit.CommentSelection	CTRL-K, CTRL-C	Marks the current line or selected lines of code as a comment, using the correct comment syntax for the programming language.
Edit.CompleteWord	ALT-RIGHT ARROW CTRL-SPACEBAR	Displays statement completion based on the current language or autocompletes word if existing text unambiguously identifies a single symbol.
Edit.Delete	DELETE	Deletes one character to the right of the cursor.
Edit.DeleteBackwards	BACKSPACE SHIFT-BACKSPACE	Deletes one character to the left of the cursor.
Edit.DeleteHorizontal-Whitespace	CTRL-K, CTRL-\	Removes horizontal whitespace in the selection or deletes whitespace adjacent to the cursor if there is no selection.
Edit.FormatDocument	CTRL-K, CTRL-D	Applies the indenting and space formatting for the language as specified on the Formatting pane of the language in the Text Editor section of the Options dialog to the document. This shortcut is available only in VB.NET—in other languages you must first select the whole document with Ctrl-A and then format the selection with Ctrl-K, Ctrl-F.
Edit.FormatSelection	CTRL-K, CTRL-F	Applies the indenting and space formatting for the language as specified on the Formatting pane of the language in the Text Editor section of the Options dialog to the selected text.
Edit.HideSelection	CTRL-M, CTRL-H	Hides the selected text. A signal icon marks the location of the hidden text in the file. VB.NET only.
Edit.InsertTab	TAB	Indents the currently selected line or lines by one tab stop. If there is no selection, this inserts a tab stop.
Edit.LineCut	CTRL-L	Cuts all selected lines or the current line if nothing has been selected to the clipboard.
Edit.LineDelete	CTRL-SHIFT-L	Deletes all selected lines or the current line if no selection has been made.
Edit.LineOpenAbove	CTRL-ENTER	Inserts a blank line above the cursor.
Edit.LineOpenBelow	CTRL-SHIFT-ENTER	Inserts a blank line below the cursor.
Edit.LineTranspose	SHIFT-ALT-T	Moves the line containing the cursor below the next line.
Edit.ListMembers	CTRL-J	Lists members for statement completion when editing code.
Edit.MakeLowercase	CTRL-U	Changes the selected text to lowercase characters.
Edit.MakeUppercase	CTRL-SHIFT-U	Changes the selected text to uppercase characters.
Edit.OverTypeMode	INSERT	Toggles between insert and overtype insertion modes.
Edit.ParameterInfo	CTRL-SHIFT-SPACEBAR	Displays a tooltip that contains information for the current parameter, based on the current language.
Edit.StopHidingCurrent	CTRL-M, CTRL-U	Removes the outlining information for the currently selected region.
Edit.StopOutlining	CTRL-M, CTRL-P	Removes all outlining information from the entire document.
Edit.SwapAnchor	CTRL-R, CTRL-P	Swaps the anchor and endpoint of the current selection.
Edit.TabLeft	SHIFT-TAB	Moves current line or selected lines one tab stop to the left.
Edit.ToggleAllOutlining	CTRL-M, CTRL-L	Toggles all previously marked hidden text sections between hidden and display states.

Edit.ToggleBookmark	CTRL-K, CTRL-K	Sets or removes a bookmark at the current line.
Edit.ToggleOutliningEx-pansion	CTRL-M, CTRL-M	Toggles the currently selected hidden text section or the section containing the cursor if there is no selection between the hidden and display states.
Edit.ToggleTaskList-Shortcut	CTRL-K, CTRL-H	Sets or removes a shortcut in the tasklist to the current line.
Edit.ToggleWordWrap	CTRL-R, CTRL-R	Enables or disables word wrap in an editor.
Edit.UncommentSelection	CTRL-K, CTRL-U	Removes the comment syntax from the current line or currently selected lines of code.
Edit.ViewWhiteSpace	CTRL-R, CTRL-W	Shows or hides spaces and tab marks.
Edit.WordDeleteToEnd	CTRL-DELETE	Deletes the word to the right of the cursor.
Edit.WordDeleteToStart	CTRL-BACKSPACE	Deletes the word to the left of the cursor.
Edit.WordTranspose	CTRL-SHIFT-T	Transposes the two words that follow the cursor. (For example, |End Sub would be changed to read Sub End|.)

Text selection

Command	**Shortcut**	**Description**
Edit.CharLeftExtend	SHIFT-LEFT ARROW	Moves the cursor to the left one character, extending the selection.
Edit.CharLeftExtend-Column	SHIFT-ALT-LEFT ARROW	Moves the cursor to the left one character, extending the column selection.
Edit.CharRightExtend	SHIFT-RIGHT ARROW	Moves the cursor to the right one character, extending the selection.
Edit.CharRightExtend-Column	SHIFT-ALT-RIGHT ARROW	Moves the cursor to the right one character, extending the column selection.
Edit.DocumentEndExtend	CTRL-SHIFT-END	Moves the cursor to the end of the document, extending the selection.
Edit.DocumentStart-Extend	CTRL-SHIFT-HOME	Moves the cursor to the start of the document, extending the selection.
Edit.GoToBraceExtend	CTRL-SHIFT-]	Moves the cursor to the next brace, extending the selection.
Edit.LineDownExtend	SHIFT-DOWN ARROW	Moves the cursor down one line, extending the selection.
Edit.LineDownExtend-Column	SHIFT-ALT-DOWN ARROW	Moves the cursor down one line, extending the column selection.
Edit.LineEndExtend	SHIFT-END	Moves the cursor to the end of the current line, extending the selection.
Edit.LineEndExtend-Column	SHIFT-ALT-END	Moves the cursor to the end of the line, extending the column selection.
Edit.LineStartExtend	SHIFT-HOME	Moves the cursor to the start of the line, extending the selection.
Edit.LineStartExtend-Column	SHIFT-ALT-HOME	Moves the cursor to the start of the line, extending the column selection.
Edit.LineUpExtend	SHIFT-UP ARROW	Moves the cursor up one line, extending the selection.
Edit.LineUpExtendColumn	SHIFT-ALT-UP ARROW	Moves the cursor up one line, extending the column selection.
Edit.PageDownExtend	SHIFT-PAGE DOWN	Extends selection down one page.
Edit.PageUpExtend	SHIFT-PAGE UP	Extends selection up one page.

Edit.SelectAll	CTRL-A	Selects everything in the current document.
Edit.SelectCurrentWord	CTRL-W	Selects the word containing the cursor or the word to the right of the cursor.
Edit.SelectToLastGoBack	CTRL-=	Selects from the current location in the editor back to the previous location in the navigation history.
Edit.ViewBottomExtend	CTRL-SHIFT-PAGE DOWN	Moves the cursor to the last line in view, extending the selection.
Edit.ViewTopExtend	CTRL-SHIFT-PAGE UP	Moves the cursor to the top of the current window, extending the selection.
Edit.WordNextExtend	CTRL-SHIFT-RIGHT ARROW	Moves the cursor one word to the right, extending the selection.
Edit.WordNextExtend-Column	CTRL-SHIFT-ALT-RIGHT ARROW	Moves the cursor to the right one word, extending the column selection.
Edit.WordPreviousExtend	CTRL-SHIFT-LEFT ARROW	Moves the cursor one word to the left, extending the selection.
Edit.WordPreviousExtend-Column	CTRL-SHIFT-ALT-LEFT ARROW	Moves the cursor to the left one word, extending the column selection.

Control editor (designer)

Command	**Shortcut**	**Description**
Edit.MoveControlDown	CTRL-DOWN ARROW	Moves the selected control down in increments of one on the design surface.
Edit.MoveControlDown-Grid	DOWN ARROW	Moves the selected control down to the next grid position on the design surface.
Edit.MoveControlLeft	CTRL-LEFT ARROW	Moves the control to the left in increments of one on the design surface.
Edit.MoveControlLeftGrid	LEFT ARROW	Moves the control to the left to the next grid position on the design surface.
Edit.MoveControlRight	CTRL-RIGHT ARROW	Moves the control to the right in increments of one on the design surface.
Edit.MoveControlRight-Grid	RIGHT ARROW	Moves the control to the right into the next grid position on the design surface.
Edit.MoveControlUp	CTRL-UP ARROW	Moves the control up in increments of one on the design surface.
Edit.MoveControlUpGrid	UP ARROW	Moves the control up into the next grid position on the design surface.
Edit.SelectNextControl	TAB	Moves to the next control in the tab order.
Edit.SelectPrevious-Control	SHIFT-TAB	Moves to the previous control in the tab order.
Edit.SizeControlDown	CTRL-SHIFT-DOWN ARROW	Increases the height of the control in increments of one on the design surface.
Edit.SizeControlDownGrid	SHIFT-DOWN ARROW	Increases the height of the control to the next grid position on the design surface.
Edit.SizeControlLeft	CTRL-SHIFT-LEFT ARROW	Reduces the width of the control in increments of one on the design surface.
Edit.SizeControlLeftGrid	SHIFT-LEFT ARROW	Reduces the width of the control to the next grid position on the design surface.

Edit.SizeControlRight	CTRL-SHIFT-RIGHT ARROW	Increases the width of the control in increments of one on the design surface.
Edit.SizeControlRightGrid	SHIFT-LEFT ARROW	Increases the width of the control to the next grid position on the design surface.
Edit.SizeControlUp	CTRL-SHIFT-UP ARROW	Decreases the height of the control in increments of one on the design surface.
Edit.SizeControlUpGrid	SHIFT-UP ARROW	Decreases the height of the control to the next grid position on the design surface.

Search and replace

Command	**Shortcut**	**Description**
Edit.Find	CTRL-F	Displays the Find dialog.
Edit.FindInFiles	CTRL-SHIFT-F	Displays the Find in Files dialog.
Edit.FindNext	F3	Finds the next occurrence of the previous search text.
Edit.FindNextSelected	CTRL-F3	Finds the next occurrence of the currently selected text or the word under the cursor if there is no selection.
Edit.FindPrevious	SHIFT-F3	Finds the previous occurrence of the search text.
Edit.FindPreviousSelected	CTRL-SHIFT-F3	Finds the previous occurrence of the currently selected text or the word under the cursor.
Edit.GoToFindCombo	CTRL-D	Places the cursor in the Find/Command line on the Standard toolbar.
Edit.HiddenText	ALT-F3, H	Selects or clears the Search Hidden Text option for the Find dialog.
Edit.IncrementalSearch	CTRL-I	Starts an incremental search—after pressing Ctrl-I, you can type in text, and for each letter you type, VS.NET will find the first occurrence of the sequence of letters you have typed so far. This is a very convenient facility, as it lets you find text by typing in exactly as many characters as are required to locate the text and no more. If you press Ctrl-I a second time without typing any characters, it recalls the previous pattern. If you press it a third time or you press it when an incremental search has already found a match, VS.NET searches for the next occurrence.
Edit.MatchCase	ALT-F3, C.	Selects or clears the Match Case option for Find and Replace operations.
Edit.RegularExpression	ALT-F3, R	Selects or clears the Regular Expression option so that special characters can be used in Find and Replace operations.
Edit.Replace	CTRL-H	Displays the Replace dialog.
Edit.ReplaceInFiles	CTRL-SHIFT-H	Displays the Replace in Files dialog.
Edit.ReverseIncremen-talSearch	CTRL-SHIFT-I	Performs an incremental search in reverse direction.
Edit.StopSearch	ALT-F3, S	Halts the current Find in Files operation.
Edit.Up	ALT-F3, B	Selects or clears the Search Up option for Find and Replace operations.
Edit.WholeWord	ALT-F3, W	Selects or clears the Match Whole Word option for Find and Replace operations.
Edit.Wildcard	ALT-F3, P	Selects or clears the Wildcard option for Find and Replace operations.

Help

Command	**Shortcut**	**Description**
Help.Contents	CTRL-ALT-F1	Displays the Contents window for the documentation.
Help.DynamicHelp	CTRL-F1	Displays the Dynamic Help window, which displays different topics depending on what items currently have focus. If the focus is in a source window, the Dynamic Help window will display help topics that are relevant to the text under the cursor.
Help.F1Help	F1	Displays a topic from Help that corresponds to the part of the user interface that currently has the focus. If the focus is in a source window, Help will try to display a topic relevant to the text under the cursor.
Help.Index	CTRL-ALT-F2	Displays the Help Index window.
Help.Indexresults	SHIFT-ALT-F2	Displays the Index Results window, which lists the topics that contain the keyword selected in the Index window.
Help.NextTopic	ALT-DOWN ARROW	Displays the next topic in the table of contents. Available only in the Help browser window.
Help.PreviousTopic	ALT-UP ARROW	Displays the previous topic in the table of contents. Available only in the Help browser window.
Help.Search	CTRL-ALT-F3	Displays the Search window, which allows you to search for words or phrases in the documentation.
Help.Searchresults	SHIFT-ALT-F3	Displays the Search Results window, which displays a list of topics that contain the string searched for from the Search window.
Help.WindowHelp	SHIFT-F1	Displays a topic from Help that corresponds to the user interface item that has the focus.

Debugging

Command	**Shortcut**	**Description**
Debug.ApplyCodeChanges	ALT-F10	Starts an edit and continue build to apply changes to code being debugged. Edit and continue is available only in C++ projects.
Debug.Autos	CTRL-ALT-V, A	Displays the Auto window to view the values of variables currently in the scope of the current line of execution within the current procedure.
Debug.BreakAll	CTRL-ALT-Break	Temporarily stops execution of all processes in a debugging session. Available only in run mode.
Debug.Breakpoints	CTRL-ALT-B	Displays the Breakpoints dialog, where you can add and modify breakpoints.
Debug.CallStack	CTRL-ALT-C	Displays the Call Stack window to display a list of all active procedures or stack frames for the current thread of execution. Available only in break mode.
Debug.ClearAllBreakpoints	CTRL-SHIFT-F9	Clears all of the breakpoints in the project.
Debug.Disassembly	CTRL-ALT-D	Displays the Disassembly window.
Debug.EnableBreakpoint	CTRL-F9	Enables or disables the breakpoint on the current line of code. The line must already have a breakpoint for this to work.
Debug.Exceptions	CTRL-ALT-E	Displays the Exceptions dialog.
Debug.Immediate	CTRL-ALT-I	Displays the Immediate window, where you can evaluate expressions and execute individual commands.

Debug.Locals	CTRL-ALT-V, L	Displays the Locals window to view the variables and their values for the currently selected procedure in the stack frame.
Debug.Memory1	CTRL-ALT-M, 1	Displays the Memory 1 window to view memory in the process being debugged. This is particularly useful when you do not have debugging symbols available for the code you are looking at. It is also helpful for looking at large buffers, strings, and other data that does not display clearly in the Watch or Variables window.
Debug.Memory2	CTRL-ALT-M, 2	Displays the Memory 2 window.
Debug.Memory3	CTRL-ALT-M, 3	Displays the Memory 3 window.
Debug.Memory4	CTRL-ALT-M, 4	Displays the Memory 4 window.
Debug.Modules	CTRL-ALT-U	Displays the Modules window, which allows you to view the .dll or .exe files loaded by the program. In multiprocess debugging, you can right-click and select Show Modules for all programs.
Debug.NewBreakpoint	CTRL-B	Opens the New Breakpoint dialog.
Debug.QuickWatch	CTRL-ALT-Q	Displays the Quick Watch dialog with the current value of the selected expression. Available only in break mode. Use this command to check the current value of a variable, property, or other expression for which you have not defined a watch expression.
Debug.Registers	CTRL-ALT-G	Displays the Registers window, which displays CPU register contents.
Debug.Restart	CTRL-SHIFT-F5	Terminates the current debugging session, rebuilds if necessary, and then starts a new debugging session. Available in break and run modes.
Debug.RunningDocuments	CTRL-ALT-N	Displays the Running Documents window that displays the set of HTML documents that you are in the process of debugging. Available in break and run modes.
Debug.RunToCursor	CTRL-F10	Starts or resumes execution of your code and then halts execution when it reaches the selected statement. This starts the debugger if it is not already running.
Debug.SetNextStatement	CTRL-SHIFT-F10	Sets the execution point to the line of code you choose.
Debug.ShowNextStatement	ALT-NUM *	Highlights the next statement to be executed.
Debug.Start	F5	If not currently debugging, this runs the startup project or projects and attaches the debugger. If in break mode, this allows execution to continue (i.e., it returns to run mode).
Debug.StartWithoutDebugging	CTRL-F5	Runs the code without invoking the debugger. For console applications, this also arranges for the console window to stay open with a "Press any key to continue" prompt when the program finishes.
Debug.StepInto	F11	Executes code one statement at a time, tracing execution into function calls.
Debug.StepOut	SHIFT-F11	Executes the remaining lines of a function in which the current execution point lies.
Debug.StepOver	F10	Executes the next line of code but does not step into any function calls.
Debug.StopDebugging	SHIFT-F5	Available in break and run modes, this terminates the debugging session.

Debug.This	CTRL-ALT-V, T	Displays the This window, which allows you to view the data members of the object associated with the current method.
Debug.Threads	CTRL-ALT-H	Displays the Threads window to view all of the threads for the current process.
Debug.ToggleBreakpoint	F9	Sets or removes a breakpoint at the current line.
Debug.ToggleDisassembly	CTRL-F11	Displays the disassembly information for the current source file. Available only in break mode.
Debug.Watch1	CTRL-ALT-W, 1	Displays the Watch 1 window to view the values of variables or watch expressions.
Debug.Watch2	CTRL-ALT-W, 2	Displays the Watch 2 window.
Debug.Watch3	CTRL-ALT-W, 3	Displays the Watch 3 window.
Debug.Watch4	CTRL-ALT-W, 4	Displays the Watch 4 window.
Tools.DebugProcesses	CTRL-ALT-P	Displays the Processes dialog, which allows you to attach or detach the debugger to one or more running processes.

Object browser

Command	**Shortcut**	**Description**
Edit.FindSymbol	ALT-F12	Displays the Find Symbol dialog.
Edit.GoToDeclaration	CTRL-F12	Displays the declaration of the selected symbol in the code.
Edit.GoToDefinition	F12	Displays the definition for the selected symbol in code.
View.FindSymbolResults	CTRL-ALT-F12	Displays the Find Symbol Results window.
View.ObjectBrowser	CTRL-ALT-J	Displays the Object Browser to view the classes, properties, methods, events, and constants defined either in your project or by components and type libraries referenced by your project.
View.ObjectBrowserBack	ALT-+	Moves back to the previously selected object in the selection history of the object browser.
View.ObjectBrowserFor-ward	SHIFT-ALT-+	Moves forward to the next object in the selection history of the object browser.

Tool window

Command	**Shortcut**	**Description**
Tools.CommandWindow-MarkMode	CTRL-SHIFT-M	Toggles the Command window into or out of a mode allowing text within the window to be selected.
View.ClassView	CTRL-SHIFT-C	Displays the Class View window.
View.CommandWindow	CTRL-ALT-A	Displays the Command window, which allows you to type com-mands that manipulate the IDE.
View.DocumentOutline	CTRL-ALT-T	Displays the Document Outline window to view the flat or hierar-chical outline of the current document.
View.Favorites	CTRL-ALT-F	Displays the Favorites window, which lists shortcuts to web pages.
View.Output	CTRL-ALT-O	Displays the Output window to view status messages at runtime.
View.PropertiesWindow	F4	Displays the Properties window, which lists the design-time properties and events for the currently selected item.
View.PropertyPages	SHIFT-F4	Displays the property pages for the item currently selected. (For example, use this to show a project's settings.)

View.ResourceView	CTRL-SHIFT-E	Displays the Resource View window.
View.ServerExplorer	CTRL-ALT-S	Displays the Server Explorer window, which allows you to view and manipulate database servers, event logs, message queues, web services, and many other operating system services.
View.ShowWebBrowser	CTRL-ALT-R	Displays the web browser window, which allows you to view pages on the Internet.
View.SolutionExplorer	CTRL-ALT-L	Displays the Solution Explorer, which lists the projects and files in the current solution.
View.TaskList	CTRL-ALT-K	Displays the TaskList window, which displays tasks, comments, shortcuts, warnings, and error messages.
View.Toolbox	CTRL-ALT-X	Displays the Toolbox, which contains controls and other items that can be dragged into editor and designer windows.

HTML Design view

Command	**Shortcut**	**Description**
Format.Bold	CTRL-B	Toggles the selected text between bold and normal.
Format.DecreaseIndent	CTRL-SHIFT-T	Decreases the selected paragraph by one indent unit.
Format.IncreaseIndent	CTRL-T	Indents the selected paragraph by one indent unit.
Format.Italic	CTRL-I	Toggles the selected text between italic and normal.
Format.LockElement	CTRL-SHIFT-K	Prevents an absolutely positioned element from being inadvert-ently moved. If the element is already locked, this unlocks it.
Format.ShowGrid	CTRL-G	Toggles the grid.
Format.SnapToGrid	CTRL-SHIFT-G	Specifies that elements be aligned using an invisible grid. You can set grid spacing on the Design pane of HTML designer op-tions in the Options dialog, and the grid will be changed the next time you open a document.
Format.Underline	CTRL-U	Toggles the selected text between underlined and normal.
Insert.Bookmark	CTRL-SHIFT-L	Displays the Bookmark dialog.
Insert.DIV	CTRL-J	Inserts <div></div> in the current HTML document.
Insert.Hyperlink	CTRL-L	When text is selected, displays the Hyperlink dialog.
Insert.Image	CTRL-SHIFT-W	Displays the Insert Image dialog.
Table.InsertRowAbove	CTRL-ALT-UP ARROW	Adds one row above the current row in the table.
Table.InsertRowBelow	CTRL-ALT-DOWN ARROW	Adds one row below the current row in the table.
Table.InsertColumn-stotheLeft	CTRL-ALT-LEFT ARROW	Adds one column to the left of the current column in the table.
Table.InsertColumnstoth-eRight	CTRL-ALT-RIGHT ARROW	Adds one column to the right of the current column in the table.
View.Details	CTRL-SHIFT-Q	Toggles display of marker icons for HTML elements that do not have a visual representation, such as comments, scripts, and anchors for absolutely positioned elements.
View.NextView	CTRL-PAGE DOWN	Switches from design view to HTML view and vice versa.
View.VisibleBorders	CTRL-Q	Displays a 1-pixel border around HTML elements that support a BORDER attribute and have it set to zero, such as tables, table cells, and divisions.

Macro

Command	Shortcut	Description
View.MacroExplorer	ALT-F8	Displays the Macro Explorer window, which lists all available macros.
Tools.MacrosIDE	ALT-F11	Launches the macros IDE.
Tools.RecordTemporary-Macro	CTRL-SHIFT-R	Places the environment in macro record mode or completes recording if already in record mode.
Tools.RunTemporary-Macro	CTRL-SHIFT-P	Plays back a recorded macro.

Dialog resource editor (but not the Windows Forms dialog Editor)

Command	Shortcut	Description
Format.AlignBottoms	CTRL-SHIFT-DOWN ARROW	Aligns the bottom edges of the selected controls with the dominant control. The dominant control is the last one to be selected.
Format.AlignCenters	SHIFT-F9	Aligns the vertical centers of the selected controls with the dominant control.
Format.AlignLefts	CTRL-SHIFT-LEFT ARROW	Aligns the left edges of the selected controls with the dominant control.
Format.AlignMiddles	F9	Aligns the horizontal centers of the selected controls with the dominant control.
Format.AlignRights	CTRL-SHIFT-RIGHT ARROW	Aligns the right edges of the selected controls with the dominant control.
Format.AlignTops	CTRL-SHIFT-UP ARROW	Aligns the top edges of the selected controls with the dominant control.
Format.ButtonBottom	CTRL-B	Places the selected buttons along the bottom center of the dialog.
Format.ButtonRight	CTRL-R	Places the selected buttons in the top-right corner of the dialog.
Format.CenterHorizontal	CTRL-SHIFT-F9	Centers the controls horizontally within the dialog.
Format.CenterVertical	CTRL-F9	Centers the controls vertically within the dialog.
Format.CheckMnemonics	CTRL-M	Checks uniqueness of accelerator mnemonics. If you have the same accelerator key assigned to two different controls, this will warn you of the problem.
Format.SizeToContent	SHIFT-F7	Resizes the selected control(s) to fit the caption text.
Format.SpaceAcross	ALT-LEFT ARROW	Evenly spaces the selected controls horizontally.
Format.SpaceDown	ALT-DOWN ARROW	Evenly spaces the selected controls vertically.
Format.TabOrder	CTRL-D	Sets the order of controls within the dialog.
Format.TestDialog	CTRL-T	Displays the dialog to allow you to check its appearance and behavior.
Format.ToggleGuides	CTRL-G	Cycles between no grid, guidelines, and grid for dialog editing.

Accelerator and string resource editor

Command	Shortcut	Description
Edit.NewAccelerator	INSERT	Adds a new entry for an accelerator key. Available only in the accelerator editor.
Edit.NewString	INSERT	Adds a new entry in the string table. Available only in the string editor.

Control Panel ▸ System and Security ▸ System
thinkdigit.com/review.php
EENIE MEENIE
MYNIE MO

NEWS AND REVIEWS

Comprehensive news, unbiased reviews and ratings to help you make the right purchase

Control Panel ▸ System and Security ▸ System
thinkdigit.com/download.php

DOWNLOADS

Check out the latest software downloads, games for your Windows, Linux, Mac and PDA/mobiles

All this and more in the world of Technology

VISIT NOW

www.thinkdigit.com

RNI REGN. No. MAH ENG/2001/3791

www.facebook.com

Join 60000+ members of the Digit community

http//www.facebook.com/thinkdigit

Your favourite magazine on your social network. Interact with thousands of fellow Digit readers.

http://www.facebook.com/iThinkGadgets

An active community for those of you who love mobiles, laptops, cameras and other gadgets. Learn and share more on technology.

http://www.facebook.com/GladtobeaProgrammer

If you enjoy writing code, this community is for you. Be a part and find your way through development.

http://www.facebook.com/ithinkgreen

Make a difference to your world and your planet. Join our community and show that you care.