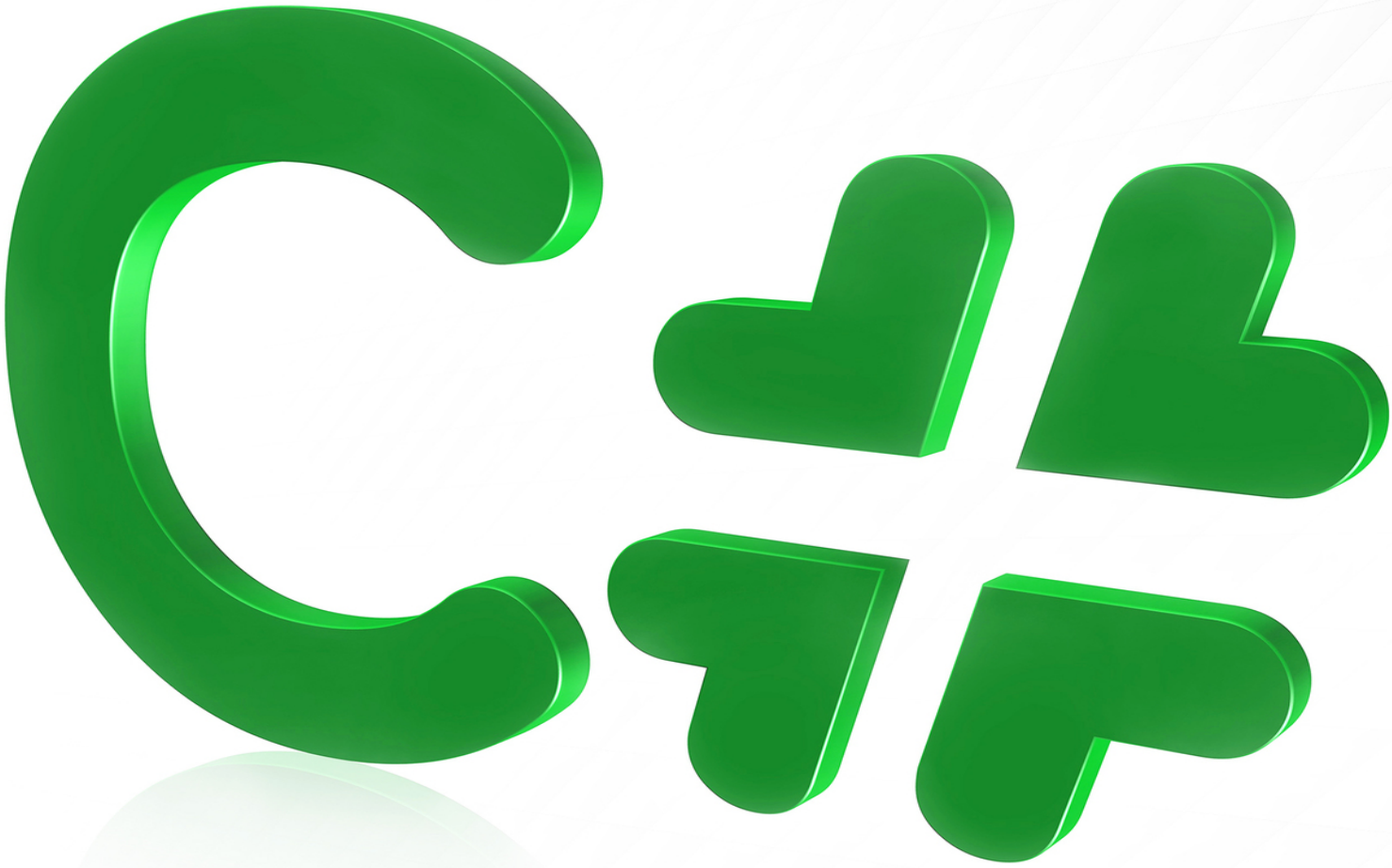


ADVANCED GUIDE TO LEARN C#

PROGRAMMING EFFECTIVELY

BENJAMIN SMITH

C#

Advanced Guide to Learn C# Programming Effectively

Table Of Contents

Introduction

Object-oriented programming is like a rabbit hole; once you start to explore it, the tunnel only seems to get deeper and deeper. This is especially true because of the family of programming languages whose cores are based on object-oriented techniques and principles. C++, C, Java, Javascript, PHP, Visual Basic, Microsoft .NET, Ruby, Scala, and a bunch of other programming languages are all classified as 'OOPL' (Object-Oriented Programming Languages). In this language family, C# is similar to that sibling who is difficult to get along with but incredibly resourceful. C# is alluring and fascinating to some people, while to others, it is a real hassle to learn and work with this language, but none of these two people at the opposite ends of the spectrum are to blame.

People familiar with mainstream OOPL, such as C++, will find that C# has all those bells and whistles that you would find in other object-oriented languages, ultimately making it easier to learn C#, but even with these similarities, the implementation of the same techniques, tools, and concepts are radically different, and on top of that, C# does not have one defined IDE, anyway. At the same time, you can argue that no programming language has one IDE that's better than the rest. Still, in the case of C#, the result is an inconsistency in the programming experience, not the case for other languages. Hence, trying to teach the reader a universal set of techniques and practices is quite literally impossible. Even the fundamental concepts may be different, even though you may be familiar with them from other object-oriented languages.

For this very reason, the first chapter of the book is essentially a review of the most important programming concepts in C#. In this way, since all the readers will likely read this chapter before moving on, everyone will have the same idea of the concepts that are being explained and used in the upcoming chapters. This is very important because literally, every discussion being made after chapter 1 builds upon the concepts that are highlighted at the very beginning. Even if the reader already knows about all the concepts discussed in the first chapter, the chapter itself does not become useless because it will still serve as an opportunity to review the knowledge that they already possess. If there's something that is not clear or is ambiguously explained, then do a brief web-search. There's a high

chance that you might just need a small detail to understand what's being discussed in the topic.

Chapter 1: Getting Familiar with the Foundations of C#

This chapter will provide the reader with an overall review of the most important concepts in C sharp. This is necessary because the upcoming chapters will build upon the things we review here. This allows the readers to become familiar with the idioms commonly used in advanced C sharp programming (which will be quite frequently used in the upcoming chapters). In this way, this chapter hopes to get the readers on the same page before we move on with advanced programming techniques and projects in the latter half of this book.

However, if you already have experience with object-oriented programming languages like Java, Javascript, C++, PHP, Visual Basic .NET, etc., you might think this chapter will probably not be worth your time, but this chapter will prove to be a really useful checkpoint where you can review the concepts you are already familiar with. On the other hand, most readers picking up this book will find this chapter to disclose certain aspects of not only C sharp but also .'NET' programming that they were previously unaware of.

In short, the content here will provide the readers with foundations necessary for understanding the discussion of C sharp programming to create advanced applications throughout the course of this book.

The Essentials of an Object-Oriented Programming Language

There is a set of fundamental elements that form the core of the language itself in any object-oriented programming language. These elements are generally known as the object-oriented basics, but this title is quite misleading as these "basics" are not to be underestimated. To be more precise, these basics are wrapped around with an external shell that gives the object-oriented language its unique identity. This is why C sharp, even though it is an object-oriented language like C++, Visual Basic, and Java, has its own unique way of distinguishing itself from its family members.

In any object-oriented programming language, the underlying functioning principles remain the same. These principles are essentially the "basics" of an object-oriented programming language, and they are a total of four. These basic principles are the following:

1. Encapsulation
2. Inheritance
3. Aggregation
4. Polymorphism

As such, any programming language that incorporates these principles is categorized as an object-oriented programming language. In order to implement each of these principal elements properly within their cores, programming languages use tools known as '**idioms** ' and '**constructors** .' For instance, these include

- Templates
- Operator overloading
- Interfaces
- Multithreading
- Multiple Inheritance
- Exception handling
- Pointers
- Garbage collection, etc.

When exploring C sharp, we find out that the core of this object-oriented programming language implements the principal elements with operator overloading, inheritance, interfaces, exception handling, garbage collection, multiple interface inheritance, reflection, and multi-threading, but you might notice that C sharp shuns a few functionalities such as templates, raw pointers, and even multiple class inheritance. The reason why these functionalities were not included within the C sharp programming language is because of the argument, which, on one side, supports the claim that these features introduce more problems than they can potentially solve. On the other side, the belief is that these are key components of any object-oriented programming language, and the features they bring to the table are priceless, and they need to be included, but keep things simple and straightforward, further complications were avoided by simply not adding these features to C sharp.

However, this does not give much confidence to anybody who has any experience in programming. This is because the feature set of templates, multiple inheritances, and pointers greatly facilitates the user in Web application and Web Services projects. Hence, this might be a monumental

deal-breaker for some people, but C sharp does not disappoint their end-users. In order to make up for the lack of such tools, C sharp includes its own feature set that can help users to build Web Applications and Web Services. For instance, C sharp features a tool that allows for hosting applications on the Windows platform, namely, '**COM interop** .' Not only that, but C sharp also boasts multilanguage programming as well as rapid application development.

But through all this ramble, there is still good news. As we discussed before, all the object-oriented programming languages have the same core principles with a difference in the features implementing them. You will find that some features from other OOPLs carry on to C sharp. For instance, if you are comfortable with the syntax of C++ or Java, then you'll find that the syntax of C sharp is strikingly familiar, thus allowing you to settle in considerably faster. This is just one of the many similarities you'll find in C sharp when comparing it to other object-oriented programming languages, so it's a good idea to make a quick comparison if you're coming from such languages.

In the upcoming sections, we will discuss the object-oriented basics and discuss the relative features available in C sharp.

Defining Classes

Creating and defining classes is done the same way as you would in C++, albeit the underlying construct of this action is slightly different in C sharp. The following syntax demonstrates the construct required to define a class in C sharp.

```
Access-modifier class name

{

}
```

One thing to be mindful of is that just like C++, C sharp is a case-sensitive programming language. Thus, you need to be vigilant of the case in which your classes have been defined when referencing them. Otherwise, the compiler will not recognize which class you are referring to.

The syntax demonstrated above has the potential of being confusing, so let's clarify its elements. The syntax has two elements - the '**access modifier** ' and the '**name** .' The second element is self-explanatory, i.e., it is the set of characters by which the class is recognized, but the first element's purpose is to define the scope of the class (you should already be familiar with 'scopes'). Generally, classes are specified as '**public** ' (which is the same as '**global** ' in other programming languages) so that they can be accessed by any function in any program, as long as the header file is included (where the classes have been already defined).

So, if we replace the appropriate arguments in the syntax shown above, we will end up with something like this:

```
public class MyFirstClass

{

}
```

In the curly brackets, you fit in your class's contents (the functions, methods, and objects it includes). If you look a bit closer, you will see that the class's name has all the first letters of each word capitalized. This is not random or a mistake. Instead, we follow a convention here when naming classes. The purpose of a convention is to maintain consistency when programming, which can be really impactful in the long run, but it is not mandatory to follow this naming convention (known as 'Pascal Casing'). It's all up to you whether you choose to use it or not (the same goes for other numerous conventions in programming).

Using a Class in a Console Application

Assuming that you have some knowledge of programming or any amount of programming experience, then you must be familiar with the term 'console.' When you are using IDE software to code, you will find a small window in the bottom that looks similar to that of a command prompt, giving the user feedback information when they execute certain commands (such as compile, debug, etc.). Moreover, a console will also display syntax errors in the code when asked to compile the project you have opened currently.

Similarly, suppose we want to work with C sharp to code applications, etc. In that case, we will do so by using an Integrated Development Environment (such as Microsoft Visual Studio), but we are not to confuse a 'console' with a 'console application.' While a console is natively open in an IDE at all times, a 'console application' on the other hand, is simply an executable file that does not have a fancy GUI. The interface through which the user interacts with the application is like a simple command prompt.

In such tasks, the most common action that you will likely perform during C sharp programming is implementing the classes you have defined in a certain console application. We can easily create a fresh console application by opening a 'New Project' within the IDE. The underlying code for the newly created console application is as follows:

```
using System;

namespace HelloWorld

{

/// <summary>

/// Summary description for Class1.

/// </summary>

class Class1

{

/// <summary>

/// The main entry point for the application.

/// </summary>

[STAThread]

static void Main(string[] args)

{
```

```
//
// TODO: Add code to start the application here
//
}
}
}
```

Now let’s break this code down a bit. At the very start of this code block, we notice a statement being used, which is **‘using** .’ The argument passed to this statement is ‘**System** .’ In this block of code, ‘System’ actually refers to a .’**dll** ’ assembly file available through the .’**NET Framework** ’ installed on our computer (using Windows, obviously). We do not need to go into the details as to ‘why’ a fresh console application is using the ‘**System.dll** ’ assembly file. A simple answer to this question of ‘why’ is that this file contains the necessary elements required for ‘Common Language Runtime.’ It wouldn't be wrong if we even said that the .’dll’ file is the console application itself. The ‘**using** ’ statement we use in C sharp is synonymous with the ‘**include** ’ statement in C++ and the ‘**uses** ’ statement in Delphi.

In the next step, we specify the ‘**name** ’ of the console application through the ‘**namespace** ’ statement. The argument passed to this statement will be the name we assign the project, which in this case, would be the conventional name of the very first program created by a user, i.e., ‘Hello World.’ However, in the argument, we do not add spaces in between a single namespace.

The ‘slash’ symbols that you see in the middle of this block of code represent comments. Anything that is written after three/two slashes (\\\ or \\) is considered as additional commentary made by the programmer. Comments can be really helpful either when you need to go through the program's code later on its life cycle or if somebody else is exploring it.

After the initial three comments, we will see the very first class is defined. Since this is a code generated when we create a default console application, the classes' names and constructs are standard. It is recommended that you

name a class you create something that is self-explanatory, short, and easy to recall (although not every name can have all three of these elements).

Up next, we have the core element of the entire application, the '**Main()** ' member. If you have experience with C++, then you will be quite familiar with this member. Main is not a standard member. In fact, it has a special property that turns it into what we call a '**static member** .' The property associated with a 'static' member is pretty useful. It allows us to use the method(s) associated with such a member without having an 'enclosing class' to exist. In addition, an argument known as '**string[] args** ' tells us that a bunch of strings (in the form of an array) are passed to the Main static member. These strings are basically the 'Command Line Arguments' that you will generally pass during a coding session.

Printing a Statement in a Console Application

Now that we understand the underlying elements of a console application let's build one to display a simple phrase, 'Hello World!.' To do this, we will simply be filling in the necessary arguments to the statements shown in the code of a console application.

```
using System;

namespace HelloWorld

{

/// <summary>

/// Summary description for Class1.

/// </summary>

class Class1

{

/// <summary>

/// The main entry point for the application.

/// </summary>
```

```
[STAThread]

static void Main(string[] args)

{

Console.WriteLine("Hello World!");

Console.ReadLine();

}

}

}
```

To print out the phrase, we use two basic functions, namely, '**Console.WriteLine()** ' and '**Console.ReadLine()** .' We specify the phrase or sentence we want the console application to display and pass it as an argument to the '**Console.WriteLine()** ' function. Once this is done, the '**Console.ReadLine()** ' function then stands by until it receives the carriage return it expects before it continues with its job. The purpose of the '**Console.WriteLine()** ' function is the same as the 'cout>>' statement in the C++ programming language.

When analyzing the usage of the methods to print out the phrase "Hello World!", you might notice that these methods were used without including an instance of the corresponding 'Console object.' If this were a normal scenario, then you would be absolutely right, but the truth is that these methods are actually static members present in a class named '**Console** .' So, if we want to call upon and use these methods, then we don't need to create an instance of the corresponding class's object.

Performing a 'Read' on the Command Prompt

We will now go over the approach through which we can effectively intercept the arguments specified to the console application. To do this, we simply pass a special argument to the '**Console.WriteLine()** ' method. This argument is '**args[0],** ' and you can see it being implemented in the following block of code:

```
using System;

namespace HelloWorld

{

/// <summary>

/// Summary description for Class1.

/// </summary>

class Class1

{

/// <summary>

/// The main entry point for the application.

/// </summary>

[STAThread]

static void Main(string[] args)

{

Console.WriteLine(args[0]);

Console.ReadLine();

}

}

}
```

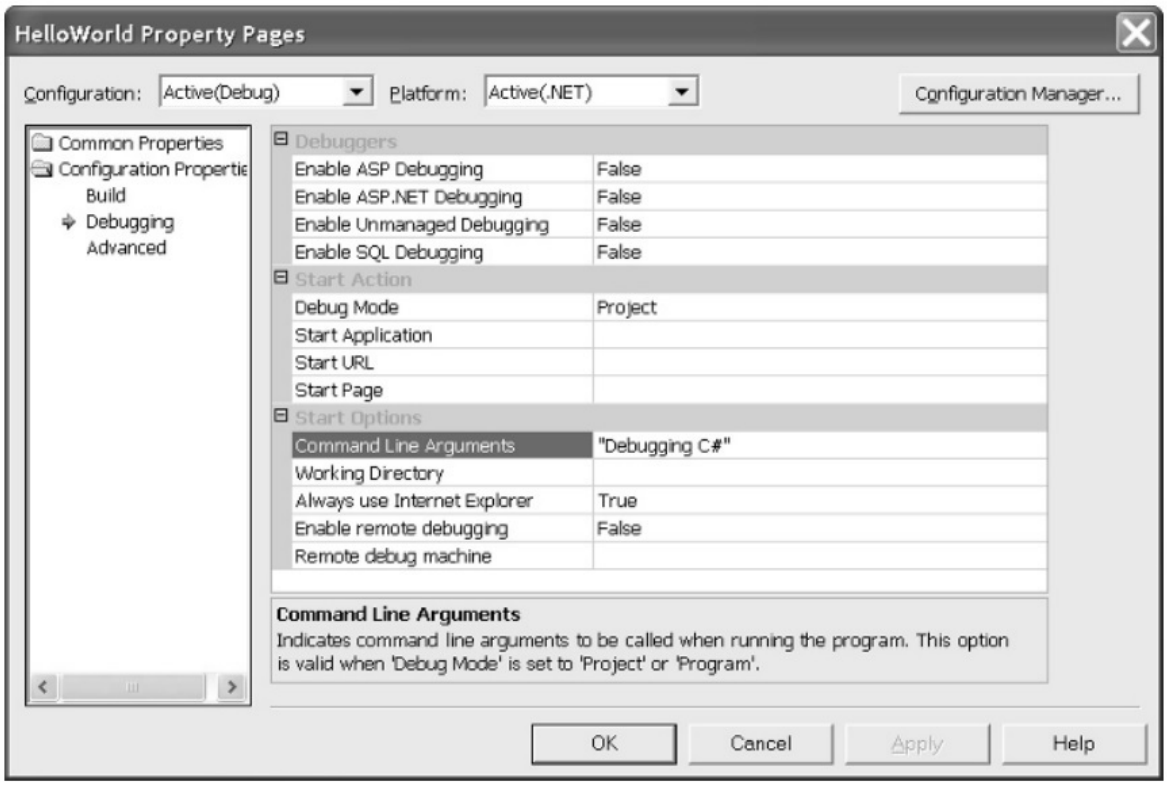
When we execute this block of code, whenever we pass an argument to the command prompt, it is intercepted by the console application displayed on the console.

Passing an argument to the command prompt can be done in two ways (there are other ways as well, but for the sake of simplicity, we will just stick with two).

- Execute an assembly through the command prompt and pass a corresponding argument.
- Using the Microsoft Visual Studio IDE to pass arguments to the command prompt.

Let's get into some details regarding the second method. Using the .'NET Visual Studio' Integrated Development Environment is not that complex since you must have experience with other IDEs at this point. In order to use the IDE to pass an argument to the command line, we need to perform the following actions:

1. Open the Visual Studio IDE and look for the 'Solution Explorer' menu inside the 'View' tab.
2. Once you are inside the 'Solution Explorer' menu, you will find a list of all the projects that you have been working on or placed in the IDE's browsing directory. Over here, you need to find the project you want to work on for this task and then just right-click it. This will open a drop-down menu, and from here, you select the 'Properties' option.
3. This will open a new window as shown below:



On the left-hand side, you will see a small file navigation interface. From here, double click 'Configuration Properties,' and inside, select 'Debugging.'

4. In the 3rd category named 'Start options,' the first section will be 'Command Line Arguments.' Over here, you need to specify an appropriate value. In this case, we pass an argument 'Debugging C#.'
5. Once we are done with passing a command-line argument, we can reboot the application we have opened in the IDE manually or by pressing the 'F5 key' as the shortcut key. Once the application runs using the IDE after a reboot, we will be able to see the argument passed to the command prompt.

Using Fields and Properties

'Fields' and 'Properties' are elements that are interestingly related to each other, but before we can discuss them, let's first understand the underlying concept of 'consumers' and 'producers' in programming. '**Consumers** ' are those people that are the end-users of things like classes. Similarly,

‘**Producers** ’ are those people that build these classes. An interesting thing to note here that both a consumer and a producer are programmers.

Now with that clarified, let’s talk about ‘**Fields** .’ A ‘Field’ is simply a bunch of data that is contained within a class. Conventionally, the fields of a class are never public. Instead, they are set to ‘private’ by the producer, but there are times when consumers need access to the fields of a class they are using, but, since the fields are private, they are hindered. This is where ‘**Properties** ’ come in. Properties have the nature of duality. To elaborate, if a consumer is using this ‘Property Method,’ it will be indistinguishable as if the consumer were dealing with data. On the other hand, if the producer uses the same property method, it will function as if it were a method.

You might ask why you would even need access to the data fields of a class in the first place. Well, the answer to that is very simple and straightforward, having access to the data of a class allows the programmer to have the freedom to make changes to the class’s methods and even perform some tweaks to the class's objects. Similarly, using ‘**Properties** ’ allows us to control how the data in the class is accessed.

Defining a Field

The nature of a field, i.e., the type of data it contains, can generally be anything (it just needs to conform to the purpose of the class in which it will be used). Since this data is important to the class, it is generally kept private through the use of access modifiers (i.e., setting its scope to ‘private’). In this way, only the programmer who created the field (the producer) will be able to make changes to it. If anyone other than the producer wants to access the fields in a class and make changes to it, they will have to do so through ‘Property Methods.’

In the following block of code, we are creating a field that stores string data.

```
using System;

namespace HelloWorld

{

/// <summary>
```

```
/// Summary description for Class1.
/// </summary>
class Class1
{
private static string arg0 = null;
/// <summary>
/// The main entry point for the application.
/// </summary>
[STAThread]
static void Main(string[] args)
{
arg0 = args[0];
Console.WriteLine(arg0);
Console.ReadLine();
}
}
}
```

In this demonstration, you can find that we are not using an instance of '**Class1** ' in this program. Now let's modify this block of code a bit such that we actually use an instance of the corresponding class while it essentially performs the same job.

```
using System;
namespace HelloWorld
```

```
{
/// <summary>
/// Summary description for Class1.
/// </summary>
class Class1
{
private string arg0 = null;
public void WriteCommandLine( string arg )
{
Console.WriteLine(arg);
Console.ReadLine();
}
/// <summary>
/// The main entry point for the application.
/// </summary>
[STAThread]
static void Main(string[] args)
{
Class1 = new Class1();
class1.arg0 = args[0];
class1.WriteCommandLine(class1.arg0);
}
```

```
}

}
```

Basically, the revision involves how we use the 'Main()' method. Originally, this method handled the majority of the work to perform the corresponding job, but we changed the purpose of the 'Main()' method in the revision such that it's the only job would be to act as a 'startup method' only. To account for the changes in the way the code works, we implement an instance of '**Class1** .'

Defining Properties

Due to their nature and function, 'Properties' are basically considered as 'methods' (you might have guessed it when we talked about 'Property Methods' in the previous sections). Generally, properties can be used in one of two ways, either as a '**right-hand side value** ' or as a '**left-hand-side value** .' If the property is being used as an RHS value, we call a specific property method known as '**getter** .' On the other hand, if the property is being used as an LHS value, then the corresponding property method we call is known as a '**setter** .'

Properties also have access to modifiers defining their scope. Conventionally, properties are set to public. Here's how a property would be defined:

```
Access-Modifier Type Property-Name

{

get

{

return field;

}

set

{
```

```
field = value;

}

}
```

In essence, the property has three basic elements - an access modifier (which is generally set to public), a ‘type,’ and last but not least, a name.

‘Properties’ can be of three types based on their use and implementation, i.e., ‘Read-only Properties,’ ‘Indexed Properties’ and ‘Write-only Properties.’ Before moving on, let’s discuss these different types of properties.

Read-Only Properties

The main element through which we can distinguish a read-only property is from observing if it possesses a ‘getter’ or a ‘setter.’ If the property only has a ‘getter’ then it is a ‘Read-only Property.’ Just like the name suggests, such properties only allow the users to view the property elements, such as its values, and the user has no authorization to make any changes.

Here’s an example where we access the value of a ‘Read-only Property’ for the temperature defined in a class. This temperature class features readings in both Celsius and Fahrenheit scale. Here’s how we can use the Read-only Property to fetch the scale in which the corresponding class's temperature will be displayed.

```
public enum TemperatureMode

{

Fahrenheit, celsius

}

class Temperature

{

private TemperatureMode mode = TemperatureMode.fahrenheit;

private double celsius = 0;
```

```
private double fahrenheit = 0;

public Temperature( double aTemperature, TemperatureMode aMode )

{

mode = aMode;

SetTemperature(aTemperature, aMode);

}

private void SetTemperature(double aTemperature, TemperatureMode
aMode)

{

if( aMode == TemperatureMode.fahrenheit )

{

fahrenheit = aTemperature;

celsius = FahrenheitToCelsius(aTemperature);

}

else

{

celsius = aTemperature;

fahrenheit = CelsiusToFahrenheit(aTemperature);

}

}

public static double CelsiusToFahrenheit(double celsius)

{
```

```
return celsius * (9.0/5) + 32;
}
public static double FahrenheitToCelsius(double fahrenheit)
{
return (fahrenheit - 32) * 5.0/9;
}
public TemperatureMode Mode
{
get
{
return mode;
}
set
{
mode = value;
}
}
public double Value
{
get
{
return mode == TemperatureMode.fahrenheit ?
```

```
fahrenheit : celsius;
}
}
public bool Test()
{
return fahrenheit == CelsiusToFahrenheit(celsius)
&& celsius == FahrenheitToCelsius(fahrenheit);
}
}
```

To determine the scale on which the temperature will be converted to (in simpler terms, determining whether to use the Celsius scale or the Fahrenheit scale), the block of code shown above executes a corresponding enumeration.

Write-Only Properties

Just as a property with only 'getters' is the defining feature of a 'Read-only Property,' a property that only has '**setters** ' is primarily known as a '**Write-only Property** .' In such a type of property, a user accessing the class can make changes to it and even to the property itself. Usually, you won't come across write-only properties as much because they are not often used, as it can cause unnecessary complications due to the modification privileges given to the end-user.

However, this does not mean that write-only properties are entirely useless. For example, you can use the functionality of write-only properties for tasks such as requesting authentication from the user in the form of a passcode. Instead of using a read-only property to display the authentication code to the end-user, it would be better to use a write-only property and allow the user to type the correct passcode in by themselves.

Indexed Properties

This type of property is also simple, like the other two. Just as the name 'Index' suggests, this type of property is primarily used to communicate the items which have been placed in the corresponding class in the form of a 'list.' Since the main user will be a consumer, the form in which the requested data is displayed is in the form of '**object[index]** .'

Here's a demonstration of the use of 'Indexed Properties.'

```
class IndexedProperty

{

private string[] args = null;

public IndexedProperty(string[] args)

{

this.args = new string[args.Length];

args.CopyTo(this.args, 0);

}

[System.Runtime.CompilerServices.IndexerName("Command")]

public string this [int index]

{

get

{

return args[index];

}

set

{

args[index] = value;
```

```
}

}

}
```

Using an Instance of a Class

In order to create an instance of an object and then have it run in the program where it was created, we need to call upon a 'Constructor' with the help of an operator. When we create an instance of an object, we can take a bunch of variables and link them to the instantiated object, but there's a condition to perform this action: whenever we try to assign a variable to an object, both elements need to be compatible with one another. The instantiated object's type and the variable's type need to be similar, to be more precise. This linking process will fail if the types of the object and the variable do not match. The type compatibility even extends to the object's predecessors. By predecessors, we mean those objects whose features this object instance inherited. So, if the variable's type matches any of the object's ancestors, then it can be assigned to the object. In other words, if we have a variable that was previously declared to be of the same type as an object's ancestor, then we can use this variable with the very same object's child instance as well.

If we want to create an instance of an object, we will need to use a 'Constructor.' A constructor is basically a method that is primarily used during the process of creating an object (this includes creating an instance as well). A constructor has three elements, a name, an access modifier, and parameters. The name of the constructor is usually the name of the class to which it belongs to. The access modifier can be either public or private, and it can have numerous parameters or none.

The following example shows the process of creating an object instance with the help of a constructor.

```
// simple object creation

FileStream fs = new FileStream("c:\\temp\\killme.txt",
FileMode.CreateNew);
```

```
fs.Write(new Byte[]{65, 66, 67, 68, 69}, 0, 5);

fs.Close();


// array examples

string[] strings = new string[10];

strings[0] = "Some Text!";

int[] integers = new int[]{0,1,2,3,4,5};
```

Defining an Interface

In the context of object-oriented programming languages, an interface isn't any different from what you might actually be familiar with. The main use of the interface is to create a method of interaction for a specific class. To be more precise, an interface is an actualization of tapping into the functionality of a class. Consider a sound system installed in a vehicle. This sound system has two circular controllers that can be rotated to navigate through the menu, increase or decrease the volume, change the radio station, or even shuffle through songs being played through external media. Now, let's narrow things down a bit. Let's say that these hardware controllers allow you to tune the radio station, and you can also do this without having to use these knobs as well. For instance, you can have a small infrared controller that can do this task for you. In this scenario, the tuning of the radio stations is done through an underlying interface, and we can access the functionality of this interface through external tools. If you think about it, the sound system's volume is controlled through a specific 'volume interface.' We can connect to this interface to leverage its volume changing functionality through an infrared remote, hardware buttons, or even voice control. As such, the hardware buttons controlling the volume are seen as the 'physical controls.' The aspect we are controlling is 'volume,' which is known as the 'attribute.' We manipulate this 'attribute' by implementing either 'Methods' or 'Property Methods', and these methods act as the supporting framework of an interface.

Here's how you can easily define an interface.

```
public interface IAudio
```

```
{
void AdjustVolume( int value );
}
```

Implementing an Interface

Once we have created an interface, we still need to implement it properly in a program or application. Generally, interfaces are implemented within classes. When implementing an interface, it should always be remembered that all the interface methods will be implemented as well.

To implement an interface, simply specify the class you want it to be in. The following block of code shows us how we can implement an interface within a class:

```
public interface IAudio
{
void AdjustVolume(int value );
}
public class Radio : IAudio
{
private int volume = 0;
void IAudio.AdjustVolume(int value)
{
volume += value;
}
}
```

In this code, the interface we are implementing is '**IAudio,** ' and the class in which we are implementing it is '**Radio** .' The process is simple. First,

we write the access modifier of the concerned class, then type its name, then place a colon after the class's name (:) and finally, specify the interface's name. This has been demonstrated in the 3rd line of the code block.

Inheritance

The concept of inheritance is almost universal, be it in biology or computers. Inheritance is basically when a class passes on its features and certain characteristics to another class, but the result is not going to be a completely identical copy of the original class. The inheritance relationship usually involves two or more than two classes. The class whose features are being inherited is known as the '**superclass,** ' and the class which is inheriting those features is known as the '**subclass** .' Think of it as the relationship of a parent and a child, in this case, the first one (superclass) would be the parent class and the second one (subclass) would be the child class.

Like we discussed before, we are not simply making an identical copy of the original class. Instead, the subclass has its own unique features, and it simply inherits the corresponding **fields, properties, and methods** belonging to the original superclass.

However, to make the inheritance work, we must also know how to denote it in our code properly. Let's say that we are dealing with two classes, 'Class X' and 'Class Y.' If we say that class **Y** inherits the members of class **X** , then we would denote this inheritance relationship in the following syntax.

```
Public class B : A
```

Here is a list of some important inheritance concepts that you always need to remember.

- The class from which the features are inherited is the **parent** class, and the class which inherits those said features is the **child** class. For example, If '**Class Y** ' inherits the members of **'Class X,'** then '**X** ' is the Parent and '**Y** ' is the Child.
- If class **Y** inherits the members of class **X** , then class **X** is referred to as the '**superclass,** ' and class **Y** is referred to as the

‘**subclass** .’

- The process of ‘**Inheritance** ’ is sometimes also referred to as ‘**generalizing** .’
- If the inheritance relationship has multiple children, then the relationship of each child with each other would be termed as ‘**siblings** .’ So, multiple classes that inherit their members from the same superclass are termed as ‘**siblings** .’

Here’s an example of using the Inheritance feature, where multiple classes are inheriting the members of the ‘**Radio Class** ’, which, in this case, would be known as the parent class.

```
public interface IAudio

{

void AdjustVolume(int value );

}

public class Radio : IAudio

{

private int volume = 0;

void IAudio.AdjustVolume(int value)

{

volume += value;

}

public int Volume

{

get

{
```

```
return volume;
}
set
{
volume = value;
}
}
private double station = 94.1;
public double Station
{
get
{
return station;
}
set
{
station = value;
}
}
public void Receive()
{
}
```

```
}
public enum RadioBand
{
AM, FM
}
public class AMFMRadio : Radio
{
private int volume = 0;
void IAudio.AdjustVolume(int value)
{
volume += value;
}
public int Volume
{
get
{
return volume;
}
set
{
volume = value;
```

```
}
}
private double station = 94.1;
public double Station
{
get
{
return station;
}
set
{
station = value;
}
}
public void Receive()
{
}
}
public enum RadioBand
{
AM, FM
}
```

```
public class AMFMRadio : Radio

{
```

This might be a little confusing for you because there are multiple instances of inheritance going on, but the core concept remains the same. In this example, we have a parent class (superclass), '**Radio** ,' and two child classes (subclasses), '**CommunicationsRadio** ' and '**AMFMRadio** .' These sibling classes inherit the important members of their parent class in order to perform their own unique functions. For the '**CommunicationsRadio** ' class, it would be to enable the functionality of transmitting and receiving radio signals, and for the '**AMFMRadio** ' class, it would be to support different Radio Bands, i.e., Amplitude Modular Bands (AM) and Frequency Modular Bands (FM).

Encapsulation and Aggregation

Both 'Encapsulation' and 'Aggregation' indicate the affinity of a class (or any data structure) to be stuffed with members, but both these terms are not the same. While encapsulation refers to the affinity of a class to withhold members within itself, **aggregation** includes those members that are classes themselves (or data structures). If the concept of encapsulation is still confusing, then consider an analogy. Think of the small leaflet that has small compartments holding medicine tablets. The medicine is 'encapsulated' in the leaflet. In this analogy, the 'medicine' is actually the 'class,' which has the methods, functions, and objects, and the leaflet represents the 'encapsulation' method. Through the use of encapsulation and aggregation, we can essentially put data in a class while controlling the authorization to access it. With encapsulation, we can add simple data types and structures. With aggregation, we can add more complex data types and structures inside a class.

However, we cannot just throw data randomly inside classes. We need to think about which class is the best container for our data. One way of going about this is to look through the list of classes available to you and determine which is the most suitable contender, i.e., which class is inherently responsible for holding a specific type of data. For example, if we consider the human lungs as 'data,' then it would make more sense to put them inside a 'human body (class)' rather than in a plant (another class).

Let's use the '**Radio** ' class to explore how we can implement 'Encapsulation' and 'Aggregation.'

```
public class BoomBox

{

private AMFMRadio radio;

public AMFMRadio Radio

{

get

{

return radio;

}

}

BoomBox()

{

radio = new AMFMRadio();

}

}
```

In this code block, we are working with a new class that we haven't previously used called '**BoomBox** .' The interesting part is that this class actually contains the '**AMFMRadio** ' class. As such, the 'AMFMRadio' class is representing the process of encapsulation as well as aggregation.

Polymorphism

The concept of 'Polymorphism' in C sharp is actually not that complicated. Polymorphism is basically involved when a programmer encounters a problem that can be handled by any general 'provider.' However, the catch

here is that the provider must be part of a bigger set of providers. This means that our problem can be solved regardless of whichever provider we choose from this collection. This is known as ‘Polymorphism, but just understanding polymorphism from this perspective might not do you any good so let’s take a more technical approach.

First, let’s take a class and declare it as a general type. When we generalize a class, it basically inherits the type of its parent class (the superclass). If we want to take this class and create several instances and want each instance of the class to have a type, we specify and not a general type, this is our ‘problem.’ The notion of this being a ‘problem’ might seem silly, but it's to help you understand the concept of ‘polymorphism.’ We can solve this problem by choosing a type from a selection of different ‘types.’ Choosing any one type will solve the problem, but we still have a wide selection we can choose from. This is ‘polymorphism’ where we first declare a class of a ‘general type’ and then declare instances with specific ‘types.’ This phenomenon can be easily observed in the working of the ‘**EventHandler delegate** .’ The delegate is instructed in a way such that its initial parameter will be an object. Since this is the first parameter, the class harboring the object will be considered as a ‘root’ for all the upcoming classes that will be used. So, this means that the delegate can virtually take any object as the first parameter to satisfy the need to specify a root class.

Access, Class, and Member Modifiers

Let’s first talk about ‘**Access Modifiers** .’ To put it simply, access modifiers define what the users of a class can see and access. If the access modifier of a class is set to ‘**private** ,’ then the consumer will not be able even to see the class, much less try to access it. If the access modifier is set to **‘public** ,’ then the consumer will be able to see the class as well as access its contents and make changes to it.

The practical use of access modifiers is allowing for guiding a user during their programming session. If someone is using your class in their code, then keeping those classes ‘private’ that do not require any attention from the consumer will focus the concentration primarily on the classes that have been purposefully kept ‘public’ by the producer. In this way, the producer ensures that the consumer only tinkers with those classes that need interaction in order to use them.

In C sharp, there are a total of five access modifiers that you can use. These access modifiers have been listed below, accompanied by brief explanations.

1. **Public Access Modifier** : this modifier provides the end-user with the freedom to access the members of the corresponding class (this also includes the 'types' in the class as well).
2. **Protected Access Modifier** : this access modifier is generally used with classes that are nested (a class within a class). As such, the subclasses (child classes) will be able to access the members of the superclass (parent class) and other classes within the nest, but the same level of access is not given to consumers. Instead, they will be restricted from accessing the members of the nested classes.
3. **Private Access Modifier** : any members of a class that have a private access modifier cannot be accessed by any subclass (child class) or any other user except for the one who created the class.
4. **Internal Access Modifier** : a class whose members and types have their access modifier specified to '**internal** ' have their access restricted only to the assembly in which they are present. In this way, any external will not be able to access these classes, or members and the code will be able to communicate with the members and types of classes.
5. **Protected Internal Modifier** : this access modifier is basically the result of slapping the protected modifier on top of the internal modifier. Just as how a protected access modifier would generally be used in nested classes, the protected internal modifier is mostly used with nested types as well. Only the assembly has unrestricted access to the members of the nested classes in a protected internal modifier.

However, there are some rules you need to follow when using access modifiers in order to avoid generating errors.

- Whenever you are defining a high-tier type, its access modifier must always be specified to 'Public.' Otherwise, it will surely cause complications when other types need to communicate with the high-tier type. If you do not want to use the 'Public' access

modifier, then the only other option you have is to use the 'Internal' access modifier.

- Whenever you are dealing with nested classes or types (in a nested class, a class contains another class. Similarly, a nested type is a type that holds the definition of another type), you have to choose between either using the 'Private' access modifier or the 'Protected' access modifier.
- Even though 'Protected Internal' is a combination of two access modifiers (protected and internal), it is highly advised to avoid combining any other access modifiers in this way.
- If a class or a type is not specified with any access modifier, then a 'default access authorization' will be given to the corresponding members, but it is not recommended to rely on this. Producers should always specify the appropriate access modifier.

The concept of using modifiers with Classes and Members has already been discussed in one way or the other when we talked about access modifiers. Regardless, if you find that the class you have defined is enclosed within a namespace, it is best to set its access modifier to either 'Public' or 'Internal.' When working with nested classes, your options are limited as to which access modifiers you can use to define them. These options include '**protected, protected internal or private'** access modifiers.

Members give the producer the freedom of assigning any access modifier to them. If you do not specify any access modifier, then it will be set to **private** by default.

Reflection

This is basically a functionality that is largely used in the .'**NET Framework** .' By using 'reflection,' a user can explore a program's assembly to extract data in the code, which can include.

- Namespaces
- Interfaces
- Classes
- Methods
- Properties
- Fields

Apart from that, reflection is an extremely useful object-oriented function that allows users to:

- Execute '**IL** ' (Intermediate Language) code during a program's run-time.
- Use special methods that allow for the viewing of metadata values recorded in attributes.
- Do some detective work, for instance, exploring the types defined in the corresponding assembly and uncovering the members.

Chapter 2: Working with the Assembly Viewer and UserControls

In the very last section of the previous chapter, we briefly discussed the object-oriented function 'Reflection.' This chapter extends the discussion and allows the reader to explore the practical use of 'Reflection' to examine the data contained within the assemblies.

This entire chapter's core concept is based on dynamically loading the assemblies and using Reflection to explore this assembly using the assembly viewer. The main take away here is that understanding the process of loading assemblies and using them gives huge potential beneficial uses. For instance, we can wirelessly load assemblies using HTTP connections, and by lightly programming the client, we can automate two tasks:

- Deploying applications that are based on the 'Windows-Forms' framework.
- Roll out updates for these applications based on Windows Forms.

Aside from this, we will also explore a number of other topics in this chapter, such as creating and declaring static methods, using interfaces like IEnumerator, and practically demonstrating the use of inheritance.

In the end, we will also explore the functionality of **UserControls** to create the buttons, icons, and actions that you see designed in many Windows applications.

Analyzing the Assembly Manager's Code Listing

In this section, we will explore one of the most important modules necessary for using the Reflection function to load assemblies and extract information. The necessary functions, methods, classes, and objects can be found within the '**AssemblyManager.cs** ' module. Unfortunately, the website on which this module's source code was hosted has been shut down, but the following list shown below has all the important bits and pieces you would need from the module itself (such as the methods or classes).

```
1: using System;

2: using System.Reflection;
```

```
3: using System.Diagnostics;
4: using Diag = System.Diagnostics;
5: using System.IO;
6: using System.Collections;
7: using System.Text;
8: using System.Windows.Forms;
9:
10: namespace AssemblyViewer
11: {
12: /// <summary>
13: /// Summary description for AssemblyManager.
14: /// </summary>
15: public class AssemblyManager
16: {
17: private string name = "";
18: private Assembly;
19:
20: public AssemblyManager(string assemblyName)
21: {
22: Debug.Assert(File.Exists(assemblyName));
23: Name = assemblyName;
24: }
```

```
25:
26: public Assembly Assembly
27: {
28: get
29: {
30: return assembly;
31: }
32: }
33:
34: public string Name
35: {
36: get
37: {
38: return name;
39: }
40: set
41: {
42: if(name.Equals(value)) return;
43: name = value;
44: Changed();
45: }
```

```
46: }
47:
48: private void Changed()
49: {
50: Debug.Assert(File.Exists(name));
51: assembly = null;
52: assembly = Assembly.LoadFrom(name);
53: }
54:
55: private string Formatted(Type type)
56: {
57: const string mask = "{0} \r\n";
58: string result = string.Format(mask, type.Name);
59: Broadcaster.Broadcast(result);
60: return result;
61: }
62:
63: private string Formatted(MemberInfo info)
64: {
65: const string mask = "\t{0} {1}\r\n";
66: string result = string.Format( mask,
67: info.MemberType.ToString(), info.Name );
```

```
68: Broadcaster.Broadcast(result);
69: return result;
70: }
71:
72: private string Load()
73: {
74: StringBuilder str = new StringBuilder();
75: IEnumerator outer = assembly.GetTypes().GetEnumerator();
76:
77: while(outer.MoveNext())
78: {
79: Type type = (Type)outer.Current;
80: str.Append(Formatted(type));
81:
82: IEnumerator inner = type.GetMembers().GetEnumerator();
83:
84: while(inner.MoveNext())
85: {
86: str.Append(Formatted((MemberInfo)inner.Current));
87: Application.DoEvents();
88: }
```

```
89: }
90:
91: return str.ToString();
92: }
93:
94: public string Text
95: {
96: get
97: {
98: return GetText();
99: }
100: }
101:
102: public string GetText()
103: {
104: return Load();
105: }
106:
107: }
108:}
```

The 'using' Statement

Generally, '**using** ' is implemented in either of the two scenarios (or in both at the same time):

- When importing a namespace
- When handling objects (in this case, it would be used as a 'block statement')

In the extensive list shown above, implementing the '**using** ' statement has already been demonstrated. If you have any experience with C++, then you can think of the '**using** ' statement as the '**include** ' statement. If C# is your first object-oriented programming language exposure, it will help you understand the purpose of the '**using** ' statement as a tool that helps you bring in external references to your assembly, but take note that it will be stored within a file with the extension .'DLL' whenever you are working with an assembly. So, if we use the '**using** ' statement to add an assembly itself, we are essentially bringing in a .'DLL' file. Moreover, this statement can also be used to point to specific classes or functions that belong in a general assembly file. If we want to use 'Reflection,' we will have to refer to the appropriate assembly file.

```
using System``.Reflection;
```

The assembly we are referring to is '**System.dll,** ' and the class we are using '**System.Reflection** ' is part of the assembly file.

If we want to use the controls to access the Windows Forms functionalities, then we will have to include the corresponding assembly file (**System.Windows.Forms**) in the program first.

We can also implement the '**using** ' statement to create a reference for a namespace. Whenever we specify this 'reference,' the code will automatically interpret it as the namespace it is linked to. For example,

```
using System.Diagnostics;

to

using Diag = System.Diagnostics;
```

Whenever we use the term '**Diag** ,' the program will see it as the '**System.Diagnostics** ' namespace.

Before moving on to the next section, we will discuss one final use of this directive. Let's say that you are implementing the '**using** ' statement to create a block using an object you have specified. If the '**using** ' directive is implemented explicitly with the object that the block will be using, then once this block ends, the object will be thrown away by the program. This will continue to happen regardless if there is an exception before the block finishes executing. Here's a demonstration.

```
using (FormAbout form = new FormAbout())

{

form.ShowDialog();

}
```

Once this simple code block finishes executing, the object will be thrown away.

Working with the Assembly Using Reflection

The main element which is arguably the one that does most of the work for the entire '**AssemblyManager** ' class is none other than the function '**Load()** .' This function primarily utilizes specialized objects, known as enumerators, to perform two tasks (using a different enumerator for each one):

- Going through every type which has been specified in the assembly and loads them.
- Going through the corresponding members of each type previously explored and then loading these members.

By creating specialized looping structures in a program (for instance, a nested loop), we can effectively iterate over all of these types and their respective members, thus extracting this information. Once the function has gathered the necessary information, it then converts it into a tangible form, like a string, and then displays it to the user. This gives the end-user a much-needed insight for designing and creating elements such as an interface for a class.

Before we move on, let's briefly discuss enumerators. Consider that you are presented with data stored in the form of a list or an array, and you are asked to design a program that will go through each entry in the data structure and perform a certain task. Traditionally, the first approach that you might consider is to create a loop for such a task. To do this, you would create statements using either '**for** ' or '**while** .' However, we know that the construct of such a directive is essentially a '**statement** .' If we want to use a method to execute a certain task when the program is iterating over the data, we will need to pass certain parameters to the corresponding method, but methods do not accept statements as their parameters. This is where '**enumerators** ' come in. Enumerators are simply objects that can be used for special tasks such as looping through a table of data, an array, a collection of strings, etc. Basically, enumerators are used along with an algorithm. So, you take an enumerator, pass it to a method as its parameter, and specify the algorithm the method will use.

If you want to use the **'Load()** ' function, you will first have to know about its syntax and define it in your program. If you look at the Assembly Manager's code listing at the beginning of this section, you will find the load function's definition beginning from 'line **69** to line **90** .' If we bring our attention to line number 70 on the code listing, we will find that the function definition includes creating an object instance named '**StringBuilder** .' An instance of this object is created seemingly effortlessly because it has already been defined previously (you will find the definition if you look through the namespace '**System.Text** '). The purpose of this object is to boost the logistics of the converted string characters. To be more precise, a lone string buffer available to the '**StringBuilder** ' object has a considerably larger capacity. Thus, a larger volume of string information can flow through the buffer, making string operations even faster.

Now let's proceed with discussing how we can actually use Reflection. The first requirement is to select an appropriate instance of an object from the assembly file we are working with. Once we have the object instance in our sight, we essentially have the assembly's reference. From this point, all we have to do is simply employ Reflection, and the assembly will reveal all the members that have been defined within it, but things are a bit different if we are dealing with a type instance instead of an object instance. As we know,

the members of each type are specific to their unique elements (for instance, a type that only has members referring to 'Methods' and other types referring to 'Field' information, etc.). So, when we use Reflection on a type instance, then we will extract information that it refers to. Hence, if we want data regarding Methods, Fields, Events, Properties, etc. then we will have to use Reflection on the respective type instances.

Although we know what to do when we have an object instance, however, we do not know how we can summon the object instance in the first place. Well, there are largely two methods through which we can extract an object instance.

1. Using a method '**AssemblyLoadFrom()** .' The argument to be passed to this method is the name of the assembly we are working on.
2. Using a static method '**Reflection.Assembly.GetcallingAssembly()** .'

With either of these two methods, we can essentially extract the reference to the assembly we want to work on. Once you obtain the reference, you now need to use appropriate methods to perform certain actions on this assembly file. For instance, if we have an assembly reference and want to extract information that tells us all the types defined in this assembly, we will have to use the appropriate methods. In this case, we will first have to pass an array declaration and set the 'type' of this array to the types we want to extract from the assembly. Once the array is set up, all we have to do is call upon the method '***assembly*** **.GetTypes()** .' However, you need always to remember when using this method that the part '***assembly*** *'* basically is the actual assembly reference you previously obtained.

There's more to the functionality of '**Reflection** ' than just simply unraveling the contents of an assembly. When we use Reflection to extract information regarding the types and members contained within the assembly, at this point, Reflection is capable of using them as well, even though their definitions are in the assembly file. Here's a demonstration of how to load an assembly file and using reflection to execute a static method which is originally from the assembly itself.

```
const string s =
```

```
"c:\\winnt\\Microsoft.NET\\Framework" +
"\\v1.0.3705\\System.Windows.Forms.dll";
Assembly = Assembly.LoadFrom(s);
Type type = assembly.GetType(
"System.Windows.Forms.MessageBox", true);
type.InvokeMember("Show", BindingFlags.Public |
BindingFlags.InvokeMethod |
BindingFlags.Static, null, null,
new object[] {"Invoked by Reflection"});
```

The 'Broadcaster' Class

If you go through the Assembly Manager's code listing, you will come across this somewhat unique statement utilizing a class which most of you would not have seen before.

```
Broadcaster.Broadcast
```

This statement essentially uses the 'Broadcaster' class to literally 'broadcast' information to a designated target. Any other user who intercepts this broadcast can access the information being propagated as well. Such an approach is commonly used to send out information such as an ongoing operation's current progress. For this, the class would essentially need to define a 'main form' as the application that will receive the broadcast and display the information. We can see the '**AssemblyViewer** ' use this same approach to display its current status while an operation is ongoing.

Here are the '**Broadcaster** ' class and the necessary interface for receiving the broadcast.

```
1: using System;
2: using System.Collections;
```

```
3:
4: namespace AssemblyViewer
5: {
6: /// <summary>
7: /// Summary description for Broadcaster.
8: /// </summary>
9: public class Broadcaster
10: {
11: private static Broadcaster instance = null;
12: private ArrayList listeners = null;
13: protected Broadcaster()
14: {
15: listeners = new ArrayList();
16: }
17:
18: static private Broadcaster Instance
19: {
20: get
21: {
22: if( instance == null )
23: instance = new Broadcaster();
24: return instance;
```

```
25: }
26: }
27:
28: public static void Add(IListener listener)
29: {
30: Instance.listeners.Add(listener);
31: }
32:
33: public static void Remove(IListener listener)
34: {
35: Instance.listeners.Remove(listener);
36: }
37:
38: public static void Broadcast(string message)
39: {
40: IEnumerator enumerator = Instance.listeners.GetEnumerator();
41: while(enumerator.MoveNext())
42: {
43: if(((IListener)enumerator.Current).Listening())
44: ((IListener)enumerator.Current).Listen(message);
45: }
```

```
46: }
47: }
48:
49: public interface IListener
50: {
51: bool Listening();
52: void Listen(string message);
53: }
54: }
```

Can essentially link any class with the '**IListener** ' interface and the class will gain the ability to intercept the string information that is being sent to the target location, but to do this, the class needs to implement the interface and register with the class that is '**Broadcasting** .'

Using the 'Broadcaster' Class

Now let's understand how we can effectively implement this class in an application. Before that, the reader must always remember that '**Broadcaster** ' belongs to those groups of classes that would ideally want only one instance of their respective class running. Such a group is known as '**Singleton** .' So, the broadcaster class isn't built like other classes. In order for it to function as the other singleton classes, it has a '**Private Instance Property** .'

In the entire class, you will find that there are numerous public methods available, but we must keep in mind that every single one of these '**public methods** ' is, in fact **static** . Whenever a public member from the broadcaster class is called, the corresponding method first and foremost checks with a property named '**Instance** ,' and this property is set to '**read-only** ,' so the method can access it but not modify it. This property then double-checks whether the broadcaster class has any instances created and if so, then how many are there. So, if the Instance property confirms that

there is no '**singleton** ' instance of the broadcaster class, then it proceeds to create one. This newly created instance of the broadcaster class is then passed on to a field (set to private) whose name is '**instance** .' If the **Instance** property confirms that a singleton instance of the class is already present, it then links the instance to the same private class. Once done, an object from the singleton instance of the class is given by the Instance property. The end product of all this skirmish is that the broadcaster class ends up with an object that refers to the broadcaster class's singleton instance.

Lastly, there are two important elements left,

- The **IListener Interface -** we can add this interface to an array defining the parts of the system that are tapping into the broadcast. Once the interface is added to the listener's '**ArrayList** ,' we can now execute the **broadcast()** method. Whenever we do so, every listener in the array will have the broadcaster information propagated to it.
- The **IEnumerator** - we use an enumerator as opposed to using a '**for, while** ' loop statements to have the method loop through the entire list of listeners defined in the '**ArrayList** .'

Defining the Interface for the 'Broadcaster' Class

As we have already mentioned before, the interface that is primarily being used with the **Broadcaster** class is the '**IListener** ' interface. This interface has two essential methods that help carry out the core function of the class.

1. **Listening** : this is a function whose output is a True or False value. In other words, a '**Boolean value** .' This Boolean value returned by the function actually tells the interface if the object (in which the interface was implemented) is willing to intercept the messages being sent from the broadcaster or not. In this way, the object can mute the broadcast without having to remove its registry with the broadcaster it is linked to.
2. **Listen** : this method is responsible for intercepting the message being broadcasted in strings.

In order to use the broadcaster in your application, all you have to do is implement the '**IListener** ' interface with a class that you want to receive

the broadcasts. The following block of code demonstrates how to use the '**IListener** ' interface in order to use the broadcaster class.

```
public class FormMain : System.Windows.Forms.Form, IListener
{
private void FormMain_Load(object sender, System.EventArgs e)
{
Broadcaster.Add(this);
}
protected override void Dispose( bool disposing )
{
if( disposing )
{
if(components != null)
{
components.Dispose();
}
}
Broadcaster.Remove(this);
base.Dispose( disposing );
}
void IListener.Listen(string message)
{
```

```
ChangeStatus(message.Trim());

}

bool IListener.Listening()

{

return true;

}

}
```

Working with Windows Forms Control

Numerous Form Controls have been defined in the AssemblyViewer that we can use. For instance, we have the:

- **StatusBar, MainMenu, OpenFileDialog, RichTextBox controls, etc.**

Only in the 'FormMain.cs' module of the AssemblyViewer. We can use these controls to create interfaces for fully fledged applications using the .**NET Framework** , but we cannot fully list the module's entire code since it contains over 400 entries. Due to this, we will see portions of code shown from this module.

Let's start things off by simply adding the element of 'Form Controls' in an existing project. Whenever a form is included in a program, the primary namespace used in the project is essentially extended to cover the class responsible for defining the '**form** ' as well. As such, we do not need to create a separate namespace when adding a form control. The syntax for including a Windows Form Control has been shown below:

```
public class FormMain : System.Windows.Forms.Form, IListener
```

Once you begin using Form Controls in your programs, you will come to notice that the underlying syntax of adding a 'form control' is actually based on '**Inheritance** .' This is because for each 'form control' added to the project, the syntax creates a class that inherits from '**System.Windows.Forms.Form** .' If we were to explain just the syntax

shown above, then it would essentially say that the class '**FormMain** ' is supposed to inherit its members and types from the assembly file '**System.Windows.Forms.Form** .' Finally, this simple line of code also implements the '**IListener** ' interface as well.

If you use the '**Visual Studio .NET** ' as an IDE for C# programming, then in this case, you will find great convenience in implementing Windows Forms in a project. This is because with Visual Studio .NET, you have two options when working with Windows Forms Controls:

1. **Code view:** just as the name suggests, the IDE shows you the underlying codes responsible for the shape, size, and contents of the 'form control.' You can change its elements by manipulating the code directly and include additional controls by adding the corresponding code.
2. **Form view:** in this option, Visual Studio .NET provides the programmer with a purely graphical view of form control being added. In this mode, you can manipulate the form control elements directly by interacting with the IDE interface. You can also add other form controls through the options provided within the IDE itself. This provides an intuitive way for people to work with form controls without having to interact with the corresponding code too much.

Implementing the 'StatusBar' Form Control

Anyone with any experience with using computers is familiar with the term '**Status Bar** .' Generally, the purpose of a status bar is to display the progress of anything, and these status bars are usually present in an application. For example, take an installation wizard application. When you click the '**install** ' button (which is also a form control if you hadn't noticed), the application proceeds to copy the application files to the specified destination. The installation progress is shown on the '**status bar** .' We can also use '**StatusBar** ' as a live graphical feed shown the application's current status. To implement the '**StatusBar** ' in an application, we use the present in the module using the application known as '**AssemblyViewer** .' The code that you will have to extract has already been shown below.

```
private void Initialize()
{
openFileDialog1.InitialDirectory = GetInitialDirectory();
Broadcaster.Add(this);
}
private void ChangeStatus(string status)
{
statusBar1.Text = status;
statusBar1.Refresh();
}
void IListener.Listen(string message)
{
ChangeStatus(message.Trim());
}
```

We already talked about how we can achieve this functionality using the **Broadcast** function in the broadcast class and the corresponding interface '**IListener** .'

Implementing the 'MainMenu' Form Control

On any standard windows application, you will see a 'main menu' that features all possible features that a user might want to access. Generally, a menu can be primarily made up of only menu items or a bunch of button controls too. If we look at a menu from a technical viewpoint, it will have two objects - '**MainMenu** ' and '**MenuItem** .' When we are creating a 'MainMenu' form control, we are basically taking a blank form control and placing the respective objects and elements on it, (although the IDE provides templates for this purpose as well).

The interactive process of adding elements on a '**MainMenu** ' form control is very intuitive, easy to understand, and doesn't require any special coding so let's move on to the next form.

Implementing the 'OpenFileDialog' Form Control

The '**OpenFileDialog'** form control is something that you have come across when using Windows. When you are uploading something from your computer's hard drive to cloud storage, you first need to specify the location of the file on your system. This opens a small pop-up window which is like a mini-file explorer, and you navigate to your desired file through this explorer. This is the '**OpenFileDialog** ' form control being practically implemented. To implement this, we simply need to use a method that is defined in the '**FormMain** ' module as shown below.

```
public void OpenAssembly()

{

if( openFileDialog1.ShowDialog() == DialogResult.OK)

OpenAssembly(openFileDialog1.FileName);

}
```

The statement '**OpenAssembly** ' is responsible for loading the file we have just selected on to the AssemblyViewer. Remember that you will have to add corresponding functionality to the 'OpenFileDialog' form control for different purposes. For instance, you might remember that some pop-up file explorers like this have a special category named '**File Type** ' which allows the user to filter through different file types. We can add this functionality to this form control as well by using the following line of code:

```
openFileDialog1.Filter = "Class Library(*.dll)|*.dll|Executables

(*.exe)|*.exe";
```

Multi-Threading using the ThreadPool Class

The very subject of 'multi-threading' is not a simple one but, almost every programmer or a computer enthusiast is familiar with what this term insinuates. A thread basically defines a stream of code execution that is

given to an application by the CPU. A CPU has a finite number of threads that are advertised as part of its specification (for example, 8 core and 16 threads CPU means each core has features 2 threads).

In order to ensure that an application initializes smoothly and then also runs without being bottlenecked by aspects such as waiting for a portion of code to finish executing before proceeding, programmers need to implement multi-threading. We can go on talking about multi-threading for hundreds of pages and still not cover the entire topic, thus let's talk about multi-threading only when we work with **AssemblyViewer** . Let's discuss multi-threading inside the AssemblyViewer application itself. When you run the application, you see a 'splash-screen' as the greeting animation. Well, AssemblyViewer uses a total of 2 threads at this point, one thread is used to execute the code for the splash-screen while the other thread is used to initialize the application itself. The splash-screen itself serves other purposes as well, but that's a discussion for another day.

In this section, we will talk about the class which is primarily designed to help programmers implement and handle multi-threading inside their application. This class is the '**ThreadPool** ' class. Think of this class as a bank manager. Just as how the bank manager 'manages' the employees at the branch, similarly, the threadpool class manages the total number of threads that are available to the application, but when you are implementing multi-threading on Windows Form Controls, it calls for extra care because form controls can readily gobble up and hog a precious resource, threads, for other applications or other processes in the same application.

The following lines of code use the threadpool class to use multi-threading to implement a different animation of the same splash-screen.

```
1: bool done = false;

2: private void Increment()

3: {

4: Opacity += .05;

5: lock(this)

6: {
```

```
7: done = Opacity >= 1;
8: }
9: }
10:
11: private void Show(Object state)
12: {
13: try
14: {
15: while(!done)
16: {
17: this.Invoke(new MethodInvoker(Increment));
18: Thread.Sleep(100);
19: }
20: Thread.Sleep(1500);
21: }
22: finally
23: {
24: Close();
25: }
26: }
27:
```

```
28: public static void Splash()
29: {
30: #if !DEBUG
31: FormSplash form = new FormSplash();
32: form.Opacity = 0;
33: form.Show();
34: ThreadPool.QueueUserWorkItem(
35: new WaitCallback(form.Show));
36: #endif
37: }
```

Implementing a Control Library

If you want to start building UserControls, the easiest way is to use the template in the Windows Control Library available in the New Project dialog. Using a solitary UserControl the project will be made, and an assembly having a DDL extension will be yielded by the Windows Control Library. There is another way, to begin with, Class Library and later you add all the UserControls yourself, but this method is not supported as there is no motivation not to use easy shortcuts if you have easy access to them.

The yield type you get will be a Class Library at the point when a Windows Control Library is made. The Windows Control Library template will add modules like UserControll.cs and AssemblyInfo.cs. It will likewise add namespace references like System, System.Drawing, Systemn.XML, and System.Data. To start working with the first UserControl, a class that is present in the UserControl module can be used. It is inherited from the UserControl. To add the metadata, the AssemblyInfo file can be used.

The Windows Control Library allows you to add unlimited UserControls as per your requirements or demand. In light of the fact that a .resx file is related to the UserControl, it is preferred to restrict every module to a

solitary UserControl, but there is definitely no restriction to have a solitary class for every module.

You can also build the UserControl by using some general solution and later test it with the help of some generic application. Many experts mostly recommend this method. It is easy to include the new UserControl in any application when the requirements like methods, constituent controls, overall properties, and events are present in the control. In this chapter, this method is used for building the UserControls. Considering the main control, it consists of a pile of buttons. These buttons are 4 in total. Based on the UserControl's size these 4 buttons are consequently resized. There is another control with the help of which a UserControl is made having two listboxes. This control also has buttons that allow the component movement between the listboxes. These buttons serve as visual allegory.

Creating a ButtonCluster Control

On a UserControl holder, 4 buttons are normally used by the ButtonCluster. These are present at the center of the form. The components can be moved in a demonstrated direction and this is done by clicking a specific button. The text on these buttons indicates the direction.

The norm is that the product designers must have some information regarding their audience. Furthermore, if the applications go astray from the Windows look and if it feels excessive, at that point even the proficient PC clients will not understand what to do. The methodology used currently requires a custom- made practice for the customers. It will help them in understanding the new applications.

Four buttons combine to form the UserControl. Buttons need to be kept aligned, and for this, some points must be kept in mind. Resize events must be responded to when sent to the UserControl. The button's text property must be adjusted. The Button Click events must be surfaced as it permits the customer or user to react to these events.

Making the Graphical Interface of the UserControl

Four buttons of similar size are arranged in the form of a vertical pile. These buttons play a vital role in the composition of the visual appearance of the main user control. These buttons may be aligned horizontally to make the user control as common as could reasonably be expected. The button

cluster will be kept basic. The genuine buttons may be delivered utilizing some different types of buttons.

Now, it is required that we create a new project “Windows Control Library.” This will allow for the creation of the class “button cluster.” On the UserControl holder make four buttons. By using the .NET format from Visual Studio, create the buttons which precisely have the same size. Keep the shape of these buttons generally square. Now to align the buttons vertically at the center use the Format Align Centers menu. After that, adjust the size of the UserControl. Make it as long as the height of all the buttons and keep its width equal to a solitary button's width. Try not to stress over the exact visual estimating because it will be made perfect as we’ll compose the code for it. The following task performed will be the composition of the code, which will help resize the button according to containing control size.

Implementing the UserControl Resize Event Handler

At the point when the size of UserControl is adjusted, it is required to make sure that the four buttons attached must occupy exactly ¼ of the accessible space. Luckily, the design of the UserControl can be similar to that of a form. In the Properties window, the Event view can be utilized as well as to help compose the Resize event handler.

```
private void UserControlButtons_Resize(object sender,

System.EventArgs e)

{

ResizeButtons();

}
```

The event handler is basically implemented by calling a method that is named accordingly. The method is named accordingly as it will lessen the requirement of any comment. It also makes it easy for the user to understand the specific purpose of this code. There is another way to do so by overriding the method i.e. OnResize method. Keep in mind to call the base class method on the off chance that it is decided to override the OnResize method. We see that the Resize event has a method named

‘OnResize’ which invokes the event. This is because the .NET Format utilizes the prefix ‘On’ for the event methods.

In the case that it is decided that the method OnResize will be overridden, then we can consider the code shown below for its implementation.

```
protected override void OnResize(System.EventArgs e)

{

base.OnResize(e);

ResizeButtons();

}
```

For the implementation of the integrant resize functioning, the OnResize method that is overridden is considered a good alternative. As mentioned above that both techniques work perfectly therefore only one either event handler or overridden event handler is required. Mostly the latter is preferred.

In the event that the user is working on some Format other than .NET, it is essential to comprehend why overridden methods are preferred over the event handlers. Like in this case the OnResize method.

Some programming languages use two things i.e. function pointers and procedurals types, which are both actually almost similar, for their execution. If by any chance a function pointer is utilized for the execution of the resize functioning and the users then want to execute this event, then in this case it would not be possible. The reason behind this is that at the same time the essential function pointer is able to point to just a single function.

Now if we have a look at delegates, they work differently. They keep up an inner list for the purpose of invocation. These delegates work on the principle of multicasting. Multicasting permits the delegates i.e. the .NET Format events to point to various event handlers at the same time. Each time an event is triggered the list informs every one of those handlers. There is no need to stress over that a programmer will tag along and stamp the functioning of Resize event handler as the delegates are able to perform

multicasting. To expand the functioning of the UserControls, multicast delegates are preferred as they permit the user to override methods and utilize the event handlers.

Determining Equal Subdivisions for the Buttons

Whenever the size of UserControl is adjusted, it is informed because now the event handlers are available to look after this task. This helps in executing a code for the similar resizing of the buttons.

It is advised that when writing a code, finding out which answer is general enough to be reasonable in any other situation can be helpful. Expertise says that finding out the decimal part of a shape say rectangle can be commonly helpful as this function can be used again when required. It is a smart thought to take another class and execute the functioning in that class at the point when the code might be commonly helpful. The following code helps to ascertain another base and top for the shape say rectangle. This rectangle is a fragment of an encircled rectangle.

```
using System;

using System.Drawing;

namespace UserControlsExamples

{

public class Rectangles

{

private Rectangles(){}

public static Rectangle GetVerticalRectangle(

Rectangle rectangle, int index, int segments )

{

Rectangle r = rectangle;

r.Size = new Size( r.Width,
```

```
NewBottom(rectangle, index, segments));
r.Location = new Point(0,
NewTop(rectangle, index, segments));
return r;
}
public static int NewTop( Rectangle rectangle,
int index, int segments )
{
return (int)((float)index / segments *
rectangle.Height);
}
public static int NewBottom( Rectangle rectangle,
int index, int segments)
{
return rectangle.Height / segments;
}
}
}
```

The above code has a quite simple class. It is easy to find another top dependent on the quantity of partitions and the real division. Similarly, the base can be computed by making the altitude an equivalent partition. More significant is how it is decided whenever there is another class. As mentioned in the previous topics, the most straightforward way is to name the method according to the function it performs. Here the question arises

that in the above class does the method ‘GetVerticalRectangle’ sound according to the function it performs? Is it a decent method for the button cluster? So the answer to these questions is Yes. Because the thing Rectangle gives a sensible piece of information concerning the sort of thing this method likely ought to have a place with. This can be done if you have the required understanding. Otherwise, it is definitely no ideal science.

With the above code's help, adjusting the size of the cluster buttons can be done without any problem. The following is the code for resizing the buttons:

```
private void Initialize()

{

buttons = new Button[]{buttonAllRight, buttonRight,

buttonLeft, buttonAllLeft};

}

private void ResizeButtons()

{

if( buttons == null ) return;

for( int i=0; i < buttons.Length ; i++)

{

buttons[i].Bounds = Rectangles.GetVerticalRectangle(

this.Bounds, i, buttons.Length);

}

}
```

In the above code, you can see that an array is declared to store every button’s reference. This is all executed in the ButtonCluster class. Storing the references of the button in an array encourages repeating above every

button. Furthermore, this way the method for rectangles i.e. Rectangles.GetVertivalRectangle is invoked. The parameter determines the user control's rectangle passes above i.e. 'this.Bound.' The array's length demonstrates the quantity of partitions and the index 'i' shows a single partition. In this code, variables are used to decide the quantity of buttons and the index and it does not depend on the literal values. So this makes the above code pretty extensible.

A last word on the procedure is altogether. In the case, that a user decides to utilize the Controls array of UserControl, at that point the user will be more disposed to compose a code that examines 'Type" data of every Control. Both the methodologies function admirably. It is not advised to execute a consistent procedure on the controls of a similar kind because the utilization of the 'Controls' array can lead to a complication on the off chance that the controls of UserControl are of the same kind.

Surfacing Constituent Events

To complete the task of visual effects of the cluster of buttons the thing that needed to be done is the adjustment of the Text property. Button.Image can also be utilized for a fine touch by assigning a realistic arrow.

Technically, the task is not yet completed. The buttons are created so they can send a response to the Click events. The current event handler will not allow the buyers to use the buttons on the control to react to the Click events. A solution must be proposed for this marginally tacky issue. To overcome this problem surfacing constituent controls can be helpful. Although it is not the best solution. It may seem troublesome to surface the events of the constituent control but it is not. However, the question arises that what is the need for doing this surfacing?

To foresee what is probably going to come in the extremely not so distant time, the know-how of the importance of surfacing constituent events and a substitute method will be proved quite helpful.

Promoting Events in Constituent Controls

Now, at times there is a need to add new controls in the UserControl. These new controls are added as fields. Two layers of profound UserControl's controls have their properties enclosed. Now, make a few assumptions. First of all, assume an instance i.e. userControl1 for UserControl. Additionally,

make an assumption that the instance has a control Button. Now, a code such as userControl1.Button.Text is required if the user wants to access the button's Text property. The property is two layers profound. It is realized by the quantity of the member of operators which in this case are two. Thus, in the Properties window, the Text property of buttons will not appear.

The user will be able to alter the UserControl's Button properties technically by supposing that the Button is declared as public. Next, surfacing the constituent control's properties and events is necessary if the user wishes to alter them in the Properties window. Furthermore, surfacing the events of Button Click is required if the user is to allow composing code that will react to the clicks of buttons separately.

It is suggested to compose event handlers as they allow the surfacing of the constituent control events. At the point when the constituent control events are proposed these are then passed to the events in the new UserControl. So basically there is a need to execute new UserControl events. The code given below shows how constituent control events are bubbled up to the level of Interface.

```
1: private void Initialize()
2: {
3: buttons = new Button[]{buttonAllRight, buttonRight,
4: buttonLeft, buttonAllLeft};
5:
6: buttonAllRight.Click += new EventHandler(OnAllRightClick);
7: buttonRight.Click += new EventHandler(OnRightClick);
8: buttonAllLeft.Click += new EventHandler(OnAllLeftClick);
9: buttonLeft.Click += new EventHandler(OnLeftClick);
10: }
11:
```

```
12: private bool IsValidIndex(int index)
13: {
14: return (index >= buttons.GetLowerBound(0) &&
15: index <= buttons.GetUpperBound(0));
16: }
17:
18: public Button this[int index]
19: {
20: get
21: {
22: Debug.Assert(IsValidIndex(index));
23: return buttons[index];
24: }
25: }
26:
27: public event EventHandler AllRightClick;
28: public event EventHandler RightClick;
29: public event EventHandler AllLeftClick;
30: public event EventHandler LeftClick;
31:
32: private void OnAllRightClick(object sender, System.EventArgs e)
33: {
```

```
34: if(AllRightClick != null)
35: AllRightClick(sender, e);
36: }
37:
38: private void OnRightClick(object sender, System.EventArgs e)
39: {
40: if(RightClick != null)
41: RightClick(sender, e);
42: }
43:
44: private void OnAllLeftClick(object sender, System.EventArgs e)
45: {
46: if(AllLeftClick != null)
47: AllLeftClick(sender, e);
48: }
49:
50: private void OnLeftClick(object sender, System.EventArgs e)
51: {
52: if(LeftClick != null)
53: LeftClick(sender, e);
54: }
```

In the above code, see line 6 to line 9. This segment of code adds event handlers that are declared private. These event handlers are added for the constituent button controls to the Click events. In the above code, there are 4 events that are declared public i.e. from line 27 to line 30. The names of these events are LeftClick, RightClick, AllLeftClick and AllRightClick. These events can be found in the Properties window in the Event view. Event handlers can be added to these public events by users. There is a time when the events of the Button Click are proposed. If the user has related the public events and the event handlers, then the inner events verify this situation. If the case is true, then the events are raised to the UserControl's surface.

Why surfacing events and constituent control properties is required is the genuine question that pops up in the mind. Because the event handlers can also be composed using Forms Designer. Yet the outcome is similar. It does not matter if the event handlers are composed using Forms Designer or were written.

Other Choice for Member Promotion

The ButtonCluster's buttons are declared public. It can be supposed that just like these buttons the declaration of the constituent control is also possible. As a matter of fact, the .NET IDE of Visual Studio permits the users to do so. When this constituent control is declared public, its events and properties are straightforwardly accessible to the users.

Some properties have nested reference types. These do not appear in the Properties window and their property values can be altered, but their events cannot be changed. The altered properties of these nested reference types have not persevered effectively to the asset document. Basically, these altered properties are neither kept up during the run and compilation of the program nor when the IDE is closed. Programmatically, this is considered an error.

In this case, if a person has some knowledge about C #'s historical background, he will be able to make a few inferences. The foremost is that it can be said that the C# is incompletely the contrivance of a similar person who assembled Borland's Delphi. His name was Anders Hejlsberg. Borland's Delphi is basically based on the Object Pascal Language. It is its ongoing forms assists the nested objects. Secondly, the nested properties

should have the option to be altered at design time as in the case that they can be changed programmatically. It is advised to search for nested objects to perform appropriately in the IDE sooner rather than later. Until further notice, at the design time, the nested objects don't persevere the state effectively.

Have a look at the code written below it tells that the user can also attempt the surfacing of constituent events.

```
public EventHandler AllRightClick

{

get

{

return buttonAllRight.Click;

}

}
```

As shown in the above code the event property can be returned in that way if the constituent event is to be surfaced directly. This way the event property will be permitted to become the value for an assignment statement. This value will be the right-hand side value. Yet, for the events, the assignment is not characterized. If the regular tasks are kept in consideration, then the subtraction and addition assignments are not characterized for them, but they are characterized by events.

Additionally, it may be attempted to over-burden the assignment operator. It may also be attempted to make the delegate class a sub-class. Be that as it may, the declaration of classes that are delegate is done by utilizing a sealed modifier. This implies that it is not possible to make a delegate class a sub-class.

Creating a PickList Control

The ButtonCluster which is present can be utilized in any application that consists of the ButtonCluster. This is done by referring to the DLL assembly. Likewise, in any UserControl, this ButtonCluster can be utilized

as a control. Instead of making a solitary, solid control, the complication is layered and as a result of this, the independent controls are simpler to oversee.

The following control that is constructed is the UserControl. This UserControl has three controls i.e. a ButtonCluster and two ListBox controls. The person using the ButtonCluster acts as a user of it and that user only has to concentrate on the new functioning. This is done when the PickList control is constructed for the ButtonCluster as it is a different UserControl. This is unmistakably less difficult than constructing a solid control that characterizes the ListBox and ButtonCluster functioning in a single class.

Now build another UserControl. This is for the execution of the PickList. On the UserControl color the 2 ListBox. After that, put a panel in the middle. Next, a ButtonCluster is to be placed in the panel. Now, to keep up the overall arrangements change some settings. Change the Dock properties of Right ListBox and Left ListBox to right and left, respectively. Change the middle panel to Fill. At the point when all the adjustments of the UserControl are done, the ButtonClusters and Listbox keep up their positions in the panel. The code written below helps in keeping up the arrangement of ButtonCluster:

```
buttonCluster1.Location = new Point(

panel1.Left + (panel1.Width - buttonCluster1.Width) / 2,

panel1.Top + (panel1.Height - buttonCluster1.Height) / 2);
```

Adding and Removing Elements

The four types of buttons each have their own functionality and behavior which can further be implemented in different ways. The “all right” button represented as >> is set to transfer all the elements contained in the left list to the right list whenever it is pressed. On the other hand, the “all left” button << does the exact opposite, that is moves the elements from the right to the left list. If it is required that only the selected elements are to be transferred, then the “right” button > is used to transfer them from left to the right list while the “left” button < does so from right to left.

The ListBox is composed of items or elements which are defined as the nested type i.e., ObjectCollection. The ListBox controls responsible for moving the elements around are handled by the method shown below.

```
1:  private void SourceToTarget(ListBox.ObjectCollection source,
2:    ListBox.ObjectCollection target)
3:    {
4:      target.AddRange(source);
5:      source.Clear(); 6:    }
7:
8:    private void MoveAllLeft()
9:    {
10:     SourceToTarget(listBoxRight.Items, listBoxLeft.Items);
11:   }
12:
13:   private void MoveAllRight()
14:   {
15:    SourceToTarget(listBoxLeft.Items, listBoxRight.Items);
16:   }
17:
18:   private void SourceToTarget(ListBox.SelectedObjectCollection
19:     source, ListBox.ObjectCollection target)
20:   {
21:    IEnumerator e = source.GetEnumerator();
```

```
22:     while(e.MoveNext())
23:     {
24:       target.Add(e.Current);
25:     }
26:   }
27:
28:   private void RemoveSelected(ListBox listBox)
29:   {
30:     for( int i=listBox.Items.Count - 1; i>=0; i--)
31:       if(listBox.GetSelected(i))
32:         listBox.Items.RemoveAt(i);
33:   }
34:
35:   private void MoveLeft()
36:   {
37:     SourceToTarget(listBoxRight.SelectedItems,
38:       listBoxLeft.Items);
39:     RemoveSelected(listBoxRight);
40:   }
41:
42:   private void MoveRight()
```

```
43:  {
44:    SourceToTarget(listBoxLeft.SelectedItems,
45:      listBoxRight.Items);
46:      RemoveSelected(listBoxLeft);
47:  }
```

Source ToTarget method is one which takes the two ListBox.ObjectCollection objects when it is called by MoveAllRight and MoveAllLeft. In accordance with the direction in which the items are moved, the lists are relocated as well. All the items present are represented by the ListBox.ObjectCollection. The object AddRange is set to clear up all the previous elements shown in the source list. SourceToTarget moves the selected part of the elements and makes use of two arguments which are:

- ListBox.SelectedObjectCollection which is designated as the first argument in this method.
- ListBox.ObjectCollection chosen as the second argument.

The direction of the moving elements is the one that determines the transposition of the arguments. The enumerator is used to transfer all the selected items from the source to the target and then RemoveSelected is called upon. An integer is used in lines 31 and 32 where GetSelected and RemoveAt are defined as such that they only take an integer.

The feature of ListBox includes storage of objects in the list through a special collection thus adding to the optimization of memory. Even still there is a downside to this, which is observed in managing the movement of items between lists, more specifically faced when writing general-purpose utility.

BeginUpdate and EndUpdate

In case of transferring numerous items in a ListBox, it is important not to allow it to update view every time a new element is added to it or removed from it, doing so slows down the process of loading.

The process goes such that when items to be loaded are in a large number call ListBox.BeginUpdate at the start of the update and when the process is

complete invoke ListBox.EndUpdate. Following this process, the list does not show the new items that have been added to it until the EndUpdate is called, thus preventing longer load time. This can be further explained by the code given below.

```
listBox1.BeginUpdate(); // load the list. For example, add the code on lines 21 to 25 of listing  listBox1.EndUpdate();
```

Most Windows applications have the feature AboutBox, which has its utility due to its one-time development and multiple times use. Further properties of AboutBox can be accessed by using the System.Diagnostics.FileVersionInfo class to interpret the assembly metadata.

Surfacing Constituent Properties

Constituent properties are surfaced comparatively in an easier way than the events. When a control property found in the UserControl is to be surfaced, a public property of the same type is declared. The constituent control's property is returned and set in the new control property contained in the UserControl.

As an example, consider the AboutBoxInfo UserControl is defined and we need to let the users freely alter the property of PictureBox.Image, the Image property is then surfaced. If the PictureBox control were set to use a default name, then the new user property statement added to the user control would surface the constituent image property.

```
public Image Image

{

get

{ return pictureBox1.Image;

}

set

{ pictureBox1.Image = value;
```

```
}

}
```

Observing it will show that the UserControl now contains the Image Property.

It is not important to use a descriptive name in case of the PictureBox contained within the UserControl AboutBoxInfo as a private member and there is only one of it. Hence it is quite okay to use a default name for it because even consumers can't interact with it and alter it directly.

A desired feature regarding the constituent controls would be the ability to make them public and freely manipulate them. Though they can be made public even now, they will not work properly when viewed in the properties window. The values modified while designing the constituent controls in the properties window will not be saved when the Visual Studio .NET is closed or when the application is simply compiled. If such a feature is to be supported, the constituent properties and events will not need to surfaced.

Implementing the Class 'ContactInformation'

This section will discuss how we can use the '**ContactInformation** ' class to store certain information regarding our **UserControl** and then, later on, explore this information. Generally, we can also use this class as a container for a custom class we have created for a project as well.

The following code listing defines two elements of the '**ContactInformation** ' class, i.e., the '**contact** ' class and the '**Contacts** ' object (which will serve as the container).

```
1: public class Contacts

2: {

3: private ArrayList items;

4:

5: public Contacts()

6: {
```

```
7: items = new ArrayList();
8: }
9:
10: public ArrayList Items
11: {
12: get
13: {
14: return items;
15: }
16: }
17: }
18:
19: public class Contact
20: {
21: private string firstName;
22: private string lastName;
23: private string phoneNumber;
24:
25: public Contact( string firstName, string lastName, string phoneNumber)
26: {
27: this.firstName = firstName;
```

```
28: this.lastName = lastName;
29: this.phoneNumber = phoneNumber;
30: }
31:
32: public string FirstName
33: {
34: get
35: {
36: return firstName;
37: }
38: set
39: {
40: firstName = value;
41: }
42: }
43:
44: public string LastName
45: {
46: get
47: {
48: return lastName;
49: }
```

```
50: set
51: {
52: lastName = value;
53: }
54: }
55:
56: public string PhoneNumber
57: {
58: get
59: {
60: return phoneNumber;
61: }
62: set
63: {
64: phoneNumber = value;
65: }
66: }
67: }
```

When we go through this entire code listing, we find that the class that has been defined here is essentially responsible for storing three distinct fields of information (as well as starting these fields up) - **‘firstName,’** ‘**lastName** ,’ and ‘**phoneNumber** .’ The type of information these fields contain are explained by their namespaces.

Chapter 3: Working with Video Kiosk Using C#

Windows has developed tools in the now not-so-recent past to allow users to use its flexible graphics programming structures and techniques, which were, frankly, dense and intimidating to use before then. We shall use the example of a Video Kiosk program to understand how this toolbox, named "GDI+", functions, and in doing so learn how to apply these tools in making unique and creative graphical interfaces of our own, as well as simply using them effectively in more straightforward projects.

We will cover the following sub-topics:

- Introduction to creating Custom Controls.
- Making a Control Library.
- The ControlPaint class and its implementation in customizing controls.
- Virtual Methods.
- Overriding Events via Controls.
- Exception Handling and caching Exception Flags.
- COM Interop and how to use it.
- Designing and Implementing Interfaces.
- TODO lists.
- TimeSpan and MessageBox Classes

An Introduction to GDI+

There has always been an exceptionally robust and capable graphics programming toolkit available in Windows, referred to as "GDI", though as we have mentioned it has often been held back by the technical investment required to learn how to use it effectively. Specifically, GDI consists of a group of unorganized structures and subroutines and uses API methods that do not very simply fall in line with each other. In other words, GDI is implemented onto a "canvas" procedurally.

In contrast, GDI+ is an object-oriented library of graphical tools that encapsulates all of the above into one structure, i.e. it is an ordered, categorized version of GDI. This lets us peruse one cleanly divided section of tools at a time, instead of spending half our time diving into a jumble of structures to find one that we need.

Let's learn about how GDI has been sorted into GDI+, and the resulting types, classes, and namespaces that it is made of.

GDI+ Namespaces

The GDI+ Classes are stored in *System.Drawing.dll* , and are further divided using various namespaces:

- System.Drawing
- System.Drawing.Design
- System.Drawing.Drawing2D
- System.Drawing.Imaging
- System.Drawing.Printing
- System.Drawing.Text

Combined, these broadly cover the three foci of graphics programming: *typography* (drawing text using different glyphs, fonts, sizes, etc.), *imaging* (non-trivial graphical objects, such as images, photographs, bitmaps, etc.), and *2D vector graphics* (comparatively trivial graphical objects, such as basic shapes, lines, and curves).

An Introduction to GDI+ Programming

An important concept in the modern era of internet development is statelessness, i.e. code does not retain information but rather carries flags that are accessed every time it is used. This also applies to using GDI+: code using these classes require a reminder of their properties and states, such as font size, shape, position, screen size, etc. In short, almost every piece of information that relates to the graphics environment has to be explicitly stated every time the code interacts with a DC, or device context (the "canvas" on which you're designing a graphic). You cannot cheat by caching the DC itself, either:

```
public class CacheGraphics : System.Windows.Forms.Form
{
Graphics graphics;
public CacheGraphics()
{
```

```
graphics = new Graphics();
}
}
```

This defines a class *CacheGraphics* that inherits information from *Form* , and creates a new *Graphics* object, storing it in the *graphics* variable. This does not work because GDI+ is stateless. Any changes in the DC (which is contained in the *Graphics* object) will not be transferred over into the *graphics* variable as it is only displaying a graphics according to the information it has been given - it has stored none of said data. Thus, please do not try to create a Graphics object directly using new, nor store an object of this sort in some field.

Instead, use *CreateGraphics* to make a new Graphics object to work on every time you need one (alternatively, get the *Graphics* object argument from an event handler). Creating a Graphics object is much more complex that simply calling for a new one, and using this predefined method ensures that an object is initialized smoothly every time. You will have to provide it all the state data it needs every time a new graphics operation is called for.

Let's now look at an example of how to make a graphical interface using GDI+: The PlayControl:

PlayControl

The PlayControl is a simple example of how to integrate graphics with functionality. A few things are going on, on the surface level: rounded shapes and buttons, gradients and colors, images contained in the button spaces, the VideoKiosk trademark logo, the slider on the top of the interface, etc. As one might expect, the buttons respond to Click events, causing a Click event handler to be called into function that passes its output to run the requisite code block.

Two things contribute to the reusable design of these graphical controls: one, controls are implemented separately by the button, and two, the graphical design does not actually do anything - it only serves to call a function using an IPlayer interface (which will be discussed later). For example, pressing Play plays something by passing the respective command of the PlayControl, on the condition that the IPlayer interface is used. Classes allow repeatability of individual controls and the interface allows us to reuse the entire component. This interface and its subroutines is described below.

```
using System;
namespace VideoKiosk
{
public interface IPlayer
{
void Close();
void Play();
void Pause();
void Stop();
void FastForward();
void FastReverse();
void VolumeUp();
void VolumeDown();
bool Open();
double Duration();
bool Mute();
void Mute(bool state);
double Elapsed();
}
}
```

Owing to this sizeable list of functions, we can conclude that the IPlayer interface can be used to emulate many modern devices on one computer instead.

Form1.cs , the main form in the VideoKiosk, is used to implement the interface, i.e. all functions named in the interface are handled by this form. Therefore, we can call Form1 as a player using PlayControl. To do so, we pass a reference to the PlayControl, contained within its constructor, to the form as follows:

```
private PlayControl control = null;
private void Form1_Load(object sender, System.EventArgs e)
{
control = new PlayControl(this);
control.Show();
}
```

The PlayControl instances themselves associate and keep track of players. Its constructor is as follows:

```
private IPlayer player;
public PlayControl()
{
InitializeComponent();
}
public PlayControl(IPlayer player) : this()
{
this.player = player;
}
```

Since PlayControl is dependent on Form1, the PlayControl closes when Form1 does. The above constructor shows how to call for the default constructor, which is the PlayControl method itself, and takes no arguments.

It must call this constructor since this is the function to initialize the component. Without doing so, the control objects we wanted to create will not be created, and the form will not work correctly.

This default constructor is called by *this()* ,and the subsequent player controls are cached in the *this.player* variable field. This has now created a player that we can control using PlayControl.

Let's look at how, for example, the *play* functionality is handled when we press the play button. To begin with, the function *player.Play()* is called, which runs code in Form1. Form1 contains the following in our example. This can be different for other sets of interfaces:

```
void IPlayer.Play()
{
axMediaPlayer1.Play();
}
```

A more nuanced and practical implementation of a *play* functionality (as shown in this example code for VideoKiosk's *play* command) would involve things such as event invocation and logic checks:

```
private void Play()
{
if(player==null) return;
try
{
player.Play();
PlayerState = PlayerState.playing;
}
catch
{
PlayError();
PlayerState = PlayerState.stopped;
```

```
}
}
private void playButton1_Click(
object sender, System.EventArgs e)
{
Play();
}
```

Generally, it is a good habit not to write code into an event handler itself, instead placing a call to a method, *Play* , and using PlayControl's current context to determine what to do. It uses a sentinel structure (we may also alternate towards using if-else or other conditional block statements) to make sure that PlayerState is not null. For exception handling, this method is used again, ensuring that nothing wrong happens to the player.

If the call (*player.Play*) succeeds, PlayerState is updated and the player performs an action. However, if it does not (for example, in the case of there not being a media file to play) the *Catch* block (or else conditional, etc.) is called which runs a *PlayError* method and stops the PlayerState. PlayError displays a message box, which you can edit using previously discussed procedures.

Creating the Graphics for the PlayControl Interface

Creating graphics for a computer program requires one to be proficient at programming and have a level of artistic capability and thought. Both these qualities are difficult and rare, and good graphical interfaces are challenging, though interesting, to develop.

We can apply a few different types of graphical inputs, such as vector art, photography, drawing/sketching, and others. It would be a good idea to leave the art to the artists and implement their vision into the interface and this means that creating graphics becomes a two-person job: one artist and one programmer, working in tandem.

Returning to our example, the PlayControl graphics interface consists of a background image with buttons, sliders, the tracker, and elapsed time (displayed using text) stacked on top.

Background

The background graphic for a Form is assigned using a variable called *BackgroundImage* . By default, this is a grey background defined by Windows itself, though you can use any image of your choosing and assign it to that variable. This also reduces the need for using complex control to create graphics to use. It is easier to work on a base of a predefined image and edit that instead of using custom control methods to define something, and speeds up code by not making an excessive amount of calls and keeping it lean.

Using the remaining Form controls to design regions on top of the background results in straight lines and boxes, and does not come off as very interesting. We will thus just limit our interaction to it to adding a background for now.

Buttons and Custom Controls

Let us now look at how we can add buttons to the PlayControl interface, for which we will use the GDI+ Toolbox. We would prefer to have buttons with rounded edges, and elliptical shapes as borders, but this is not offered straightforwardly by the toolbox. We'll have to make some ourselves. To do so, we will add a custom control to our project that contains all the classes and methods we might need to implement into the button's functionality. The following code describes a basic button made using custom controls, which serves as a basis to all the other buttons in our interface.

```
1: public class RoundButton :
2: System.Windows.Forms.Control
3: {
4: private bool hasOutline = false;
5:
```

```
6: protected bool down = false;
7:
8: protected virtual Color Getcolor()
9: {
10: Color[] colors = {Color.Silver, Color.Gray};
11: return colors[Convert.ToInt32(down)];
12: }
13:
14: protected virtual Brush GetBrush(bool buttonState)
15: {
16: return new LinearGradientBrush(
17: new Point(2,2), new Point(Width -1 , Height - 1),
18: Color.White, Getcolor());
19: }
20:
21: private int GetPenWidth()
22: {
23: // use button state to adjust pen width
24: return 1 + Convert.ToInt32(down);
25: }
26:
27: private Pen GetPen()
28: {
29: return new Pen(Brushes.Black, GetPenWidth());
30: }
31:
```

```
32: public bool HasOutline
33: {
34: get
35: {
36: return hasOutline;
37: }
38: set
39: {
40: hasOutline = value;
41: Invalidate();
42: }
43: }
44:
45: private void DrawButtonOutline(Graphics graphics)
46: {
47: graphics.DrawEllipse(GetPen(), 1, 1,
48: Bounds.Width - 2, Bounds.Height - 2);
49: }
50:
51: private void DrawButton(Graphics graphics)
52: {
53: graphics.FillEllipse(GetBrush(down), 0, 0,
54: Width, Height);
55: if(hasOutline)DrawButtonOutline(graphics);
56: }
57:
```

```
58: protected override void OnPaint(
59: System.Windows.Forms.PaintEventArgs e)
60: {
61: base.OnPaint(e);
62: DrawButton(e.Graphics);
63: DrawGraphic(e.Graphics);
64: }
65:
66: protected override void OnResize(System.EventArgs e)
67: {
68: GraphicsPath path = new GraphicsPath();
69: path.AddEllipse(0, 0, Bounds.Width, Bounds.Height
70: Region = new Region(path);
71: Invalidate();
72: base.OnResize(e);
73: }
74:
75: protected override void OnMouseDown(
76: System.Windows.Forms.MouseEventArgs e)
77: {
78: base.OnMouseDown(e);
79: down = true;
80: Invalidate();
81: }
82:
83: protected override void OnMouseUp(
```

```
84: System.Windows.Forms.MouseEventArgs e)
85: {
86: base.OnMouseUp(e);
87: down = false;
88: Invalidate();
89: }
90:
91: protected virtual void DrawDownGraphic(
92: Graphics graphics)
93: {
94: Matrix m = new Matrix();
95: m.Scale(1.03F,1.03F);
96: graphics.Transform = m;
97:
98: }
99:
100: protected virtual void DrawGraphic(
101: Graphics graphics)
102: {
103: if(down) DrawDownGraphic(graphics);
104: }
105:
106: protected virtual Brush GraphicBrush()
107: {
108: return Enabled ? Brushes.Black: Brushes.Silver;
109: }
```

```
110: }
```

Let's break this down:

Constructors, Destructors, and Dispose

Because a control is nothing more than a class, it will have its own constructors, destructors, and a *Dispose* method, same as any class. The *RoundButton* class does not use any of these (aside from the defaults), but this is a specific case. In other classes in *Control.dll* (itself a part of *VideoKiosk.dll*) you will notice their presence. Let's go through what they do:

Control Constructors

A constructor serves to initialize any objects that a class requires. A control constructor serves the same purpose. Defining a constructor can be ignored in situations where new members do not require initialization, or no particular method is required to be performed during initialization. Constructors are usually public, have no return type, and have the same name as their class. For our RoundButton class we do not need to have a constructor since we do not need any special form of initialization.

Constructors can be overloaded (i.e. multiples of the same name can be defined and have different functionalities, inputs, and outputs), and calls to the constructors are resolved depending on what types of data are given as arguments.

A good idea is to use the constructor to call an initializer method instead of stuffing it full of code. For example, Form1 uses the InitializeComponent method using its constructor, since it has multiple controls that need to be initialized.

Control Dispose Methods

C#, being a .NET language, employs non-deterministic destruction, i.e. objects are destroyed by predefined, and a programmer does not need to define a function to do so. A "garbage collector" removes cached data, though using this explicitly is tricky, inconvenient, and dangerous, and thus requires us to write a block of code calling it. Oftentimes we write code that requires deterministic destruction anyway. This is where we turn towards *Dispose* methods, as opposed to destructors. Dispose is a public method that

can be easily called, whereas destructors are unreliable to use in a .NET language.

Destructor

To cover our bases, we'll describe what a destructor is. It serves to remove and delete cached data after its use is complete. We won't often see it as compared to Dispose, but this is what it looks like:

```
~RoundButton()
{
}
```

A few rules about using destructors are:

- There can only be one destructor per class.
- It cannot take any arguments.
- It cannot be overloaded, nor inherited.
- It cannot be explicitly called.
- The garbage collector uses the destructor, and the user cannot call it.

Conditional Logic and using Arrays

There is a technique where instead of using if-else logic, we can turn to arrays. This makes the code more streamlined and lean and the IL code for conditional statements is shorter than that for using arrays. Take the example of the following, which checks a *down* field and returns a color. There's two ways we can do it:

```
protected virtual Color Getcolor()
{
Color[] colors = {Color.Silver, Color.Gray};
return colors[Convert.ToInt32(down)];
}
```

```
protected virtual Color Getcolor()
{
```

```
if( down )
return Color.Gray;
else
return Color.Silver;
}
```

The former utilizes our method, whereas the latter uses basic statements to achieve the same purpose and is clearer to understand. The IL code is different between the two because C# doesn't allow for typed arrays, and the Boolean must be converted to an integer first before it will work to index the array.

Also, for very simple conditionals exists the ternary operator (*? :*). It evaluates a variable as a Boolean - if true, the first option applies, and if false, the second instead. This produces the least amount of IL code, but it is difficult to use for multiple, complex conditionals (one might use *case* instead).

```
protected virtual Color Getcolor()
{return down ? Color.Gray : Color.Silver;
}
```

You can use any of these methods as you please, and since the method is well-named, you can code the routine as densely as you like. To summarize, keep the following in mind when choosing a method:

- Array indexing produces concise and specific code.
- If-else statements are simple for a few conditions, but can get difficult to decipher when nesting goes too deep and conditions get too many.
- Case statements allow you to order multiple choices.
- The ternary operator is a short and sweet method for simple binary choices.

A note about colors: a list of colors with their names is stored in the Color structure, which can be called using the *Getcolor* method. We can also

specify alpha blend (transparency of the color), red, blue, and green hues (on a scale of 0 to 255, or base 8) to create a color.

Another note: we have used a *Convert* class in Getcolor. This class consists of methods to convert datatypes to a particular type, such as *Convert.ToInt32* converting data to a 32-bit integer. These can accept most datatypes due to them being overloaded, as we have explained previously. When it isn't able to, an *InvalidCastException* is called.

Using Graphics Objects

A Graphics object in GDI+ is like a canvas, and multiple iterations of the object are required to paint on it. To create a custom drawing, you'll need to start by creating a Graphics object by either using a *CreateGraphics* method from something that has a canvas, such as Windows Forms, or as a property of the *PaintEventArgs* argument passed to the Paint event handler, or other methods. Once you've created an instance of the object, you can use the class methods to create your desired graphic. Take care not to cache this object nor use it in recursive methods where a previous one is called since these objects are stateless and remember nothing. We'll show you how to request a Graphics object from the Paint event handler, and every time a Paint operation is performed, below.

Paint Event Handler

This handler receives two input arguments:

- An object.
- A PaintEventArgs instance.

This handler is used most commonly in cases where one wants to create their own drawing. For now, let us concern ourselves with only two methods, which are directly implemented onto the PlayControl interface.

```
private void PlayControl_Paint(
object sender, System.Windows.Forms.PaintEventArgs e)
{
OutlineControl(e.Graphics);
DrawTrademark(e.Graphics);
```

```
}
```

The *Refactored* methods in the code above describe their functionality. Then, to create a nice, rounded shape, we add the first statement to use PlayControl in such a manner (we could not have created this shape otherwise). Lastly, to add the trademark "VK", we write the second statement. The following code elaborates on these functions:

```
1: private Point[] GetPoints(int shift)
2: {
3: return new Point[]
4: {
5: new Point(3 + shift, 0 + shift),
6: new Point(Bounds.Width - 3 + shift, 0 + shift),
7: new Point(Bounds.Width + shift, 3),
8: new Point(Bounds.Width + shift, Bounds.Height - 3 + shift),
9: new Point(Bounds.Width - 3 + shift, Bounds.Height + shift),
10: new Point(3 + shift, Bounds.Height + shift),
11: new Point(0 + shift, Bounds.Height - 3 + shift),
12: new Point(0 + shift, 3 + shift),
13: new Point(3 + shift, 0 + shift)
14: };
15: }
16:
17: private void OutlineControl(Graphics graphics)
18: {
19: graphics.SmoothingMode = SmoothingMode.AntiAlias;
20: graphics.DrawPolygon(new Pen(Brushes.Black, 2),
21: GetPoints(-1));
```

```
22: graphics.DrawPolygon(new Pen(Brushes.White, 2),
23: GetPoints(1));
24: }
25:
26: private void DrawShadowText(Graphics graphics,
27: string s, Font font, Brush foreBrush, Brush backBrush, int x, int y)
28: {
29: graphics.DrawString(s, font, backBrush, new Point(x, y));
30: graphics.DrawString(s, font, foreBrush, new Point(x-1, y-1));
31: }
32:
33: private void DrawTrademark(Graphics graphics)
34: {
35: Font font = new Font("Haettenschweiler", 24,
36: FontStyle.Bold);
37: DrawShadowText(graphics, "VK", font,
38: Brushes.DarkSlateBlue, Brushes.White, 5, 5);
39: }
```

OutlineControl modifies the Graphics object to set the *SmoothingMode* parameter to *AntiAlias* , which blends multiple pixels in a polygon to smoothen it out. The new, smoothened shape is then returned using *GetPoints* .

DrawTrademark creates shadows and an inset effect by applying multiple layers of text. These text layers are created using a method called *DrawShadowText* that offsets the text to a side or throughout. The inset effect is created by writing that text offset by -1 and inputting a brighter color.

Overloading OnPaint

One can either create multiple custom effects on a form every time the graphic is instanced or create custom controls to handle all of that. To do so, we would need to overload the OnPaint method. This is the method we use to remake a control (such as our RoundButton) every time it is painted.

In our PlayControl code listing, we notice that OnPaint is not an event handler, as this method is a part of the class that is being used to create a custom painting. We do not need to give it an event flag, but we need to give it a PaintEventArgs object, with the Graphic object we're using. If you want a default drawing method followed, you can use an inherited OnPaint method instead that will also call the event handler. After we've made the base drawing, we will insert this code that makes our custom painting, as we've shown in the PlayControl listing by using *DrawButton* and *DrawGraphic* . We'll go over GDI+ tools to do this soon.

Invalidating a Control

We use the *Invalidate* method to change a control's graphics. It calls OnPaint, causing a Paint event, and forces the control to be repainted. In the PlayControl listing, this is shown on line 72, in the *HasOutline* subroutine.

Drawing Shapes and Filling Them

While working with the Graphics class we have many methods to create polygonal and circular drawings and methods to fill said drawings. For example, for making our buttons we used *DrawEllipse* to draw the outline circle of the button (depending on whether HasOutline returns true) and *FillEllipse* to color in this region.

These methods work by using a Pen (in the case of Draw methods) or Brush (for Fill methods) to create their respective graphics. You can use either integer arguments or floating-point inputs to define the path they need to follow or fill, the difference being that the math in floating-point inputs is more accurate). How these graphics are created is particularly dependent on what kind of arguments these methods receive. T there are multiple overloaded functions for each kind of shape, which are selected and used by the compiler.

For rectangular regions, you can use two points to define its input instead of four, which are called Rectangle structures and Point structure, respectively (for integer inputs, or RectangleF and PointF).

Moving Graphics Objects on the Screen using Transforms

GDI+ defines its position using multiple sets of coordinates, such as world coordinates, page coordinates, and device coordinates. A Graphics method receives a world coordinate when it is called to function, and these coordinates are translated into page coordinates and then to device coordinates. All of this happens internally, and allows us as programmers to make changes to things such as the origin location, pixel size, etc. We use these transforms to provide an illusion of movement. Specifically, for our example of RoundButton, we use a scaled matrix transform of the Graphics object's position and size to do so.

```
protected virtual void DrawDownGraphic(Graphics graphics)
{
Matrix m = new Matrix();
m.Scale(1.03F,1.03F);
graphics.Transform = m;
}
protected virtual void DrawGraphic(Graphics graphics)
{
if(down) DrawDownGraphic(graphics);
}
protected override void DrawGraphic(Graphics graphics)
{
base.DrawGraphic(graphics);
graphics.FillPolygon(GraphicBrush(), GetPoints());
}
```

```
protected override void OnPaint(System.Windows.Forms.PaintEventArgs
e)
{
base.OnPaint(e);
DrawButton(e.Graphics);
DrawGraphic(e.Graphics);
}
protected override void
OnMouseDown(System.Windows.Forms.MouseEventArgs e)
{
base.OnMouseDown(e);
down = true;
Invalidate();
}
```

When the button is clicked, the RoundButton control calls the *OnMouseDown* method (in this case, we use the base class' OnMouseDown method, which allows the user to intercept that event), causing the down field to be set to True to invalidate and update the button. This calls OnPaint to repaint the control; the base.OnPaint method first, then the custom method. When down is True (I.e. the button is pushed in), the *DrawDownGraphic* method creates a matrix object. This object is scaled and fed to the Graphics object via *graphics.Transform* to create the requisite effect. Using transforms is something of an art and you'll need a good eye, practice, and retries to get the perfect fit for you.

Creating Shaped Forms, using the GraphicsPath Object

GraphicsPath is a class stored in the System.Drawing. Drawing2D namespace, used to create a conjoined series of lines and curves. Basically, it is the curve-centric analogue to Windows Forms.

The objects of this class combine straight lines and not-as-straight curves to create a shaped form. This combination is sent to the object and is used to define a clipping region for a shape. For example, the PlayControl's rounded edges are created like so:

```
private void ShapeForm()
{
GraphicsPath path = new GraphicsPath();
path.AddPolygon(GetPoints(0));
Region = new Region(path);
}
```

This code creates a GraphicsPath object *Path* . This object receives points using GetPoints, and is sent into *Region* to define a region to clip in order to create a form. This is a simple example to help you understand its basic functionality in the context of our PlayControl, but is fairly lacking on its own. We will elaborate on this shaped form below.

Defining Clipping Regions

Windows Forms includes a Region property class which represents the control's clipping region. These class objects are useful because scaling them is fairly easy owing to them being expressed in word coordinates. Defining a region yields a form, not just an arbitrary shape.

The following block of code redefines a form's clipping region and changes it to the one shown below. It is clipped in such a way that the frame is included as well. The form responds to the user's input and moves the clipping region when it is dragged around like it was the original image.

```
private void Form1_Load(object sender, System.EventArgs e)
{
BackColor = Color.Red;
GraphicsPath path = new GraphicsPath();
path.AddString("C#", Font.FontFamily, 1, 75,
new Point(0, 0), new StringFormat());
```

```
Region = new Region(path);
}
private void Form1_Click(object sender, System.EventArgs e)
{
MessageBox.Show("Clicked!");
}
```

This block of code also changes the background color in the Form Load event. We initialize a new GraphicsPath object and provide it a string and some information on the font it is to use. This object is used to create a Region, which is assigned to the form's Region field.

Linear-scaling Gradient Bushes

As we've mentioned previously, we fill and color shapes using Brush objects. There's a few kinds of these, such as *HatchBrush* , *LinearGradientBrush* , *PathGradientBrush* , *SolidBrush* , and *TextureBrush* . All of these produce a different effect when filling a graphic shape. To use a brush, we either create an instance of a brush type or use the Brushes class explicitly, which contains methods for these.

To begin with, let's talk about LinearGradientBrush. This brush type produces a gradient that transforms from one color to another over a region in a user-defined direction. In the PlayControl listing, between lines 45 and 50, an example of this in use is shown. The gradient's start and end points are provided, and the colors on either end defined.

This Brush can be modified in various ways:

- *Blend* defines how much of either is to exist at any point along the gradient region.
- *GammaCorrection* is a simple yes-or-no Boolean answer to the question of whether or not gamma correction is applied to the gradient.
- *InterpolationColors allows a user to create gradients of more colors than just two.*

- *LinearColors* is an array of the starting and ending colors of the gradient.
- *Transform* allows us to positionally transform (skew, move, rotate, scale, etc.) a region.
- *Rectangle* defines a rectangular region and a gradient between the two points used to define the region.

We have used LinearGradientBrush to paint RountButton and have also added a condition to use different gradients depending on the state of the down field.

The following listing shows how we can use LinearGradientBrush methods *RotateTransform* and *SetTriangularShape* .

```
1: using System;
2: using System.Drawing;
3: using System.Drawing.Drawing2D;
4: using System.Collections;
5: using System.ComponentModel;
6: using System.Windows.Forms;
7: using System.Data;
8:
9: namespace GradientBrushDemo
10: {
11: public class Form1 : System.Windows.Forms.Form
12: {
13:
14: [ Chopped out code that was generated by the forms designer]
15: private System.Windows.Forms.Timer timer1;
16:
17: private float angle = 0;
18:
```

```
19: private LinearGradientBrush GetBrush()
20: {
21: return new LinearGradientBrush(
22: new Rectangle( 20, 20, 200, 100),
23: Color.Orange,
24: Color.Yellow,
25: 0.0F,
26: true);
27: }
28:
29: private void Rotate( Graphics graphics,
30: LinearGradientBrush brush )
31: {
32: brush.RotateTransform(angle);
33: brush.SetBlendTriangularShape(.5F);
34: graphics.FillEllipse(brush, brush.Rectangle);
35: }
36:
37: private void Rotate(Graphics graphics)
38: {
39: angle += 5 % 360;
40: Rotate(graphics, GetBrush());
41: }
42:
43: private void timer1_Tick(object sender, System.EventArgs e)
44: {
```

```
45: Rotate(CreateGraphics());
46: }
47:
48: private void Form1_Paint(object sender,
49: System.Windows.Forms.PaintEventArgs e)
50: {
51: Rotate(e.Graphics);
52: }
53: }
54: }
```

Notice how we've used a Timer control to recreate an ellipse at regular intervals. We have used Yellow and Orange as the two gradient colors. Every time we use the *Rotate* method, we increment the floating-point value by five, which is kept in check using a modulo operator. RotateTransform rotates the brush and SetBlendTriangularShape creates a triangular-shaped gradient, the three outer points transitioning from one color to the other as it approaches the center. Finally, we use FillEllipse to fill an ellipse using this brush.

Pens

Pens work similarly to Brushes; you can either create an instance of one to use or call one explicitly from the Pens class. The code below creates a Pen object, initializing it using a LinearGradientBrush, and uses it to draw an ellipse.

```
LinearGradientBrush brush =
new LinearGradientBrush(new Rectangle(0, 0, 2, 2),
Color.White, Color.Green, LinearGradientMode.ForwardDiagonal);
Pen p = new Pen(brush, 10F);
e.Graphics.DrawEllipse( p, 5, 5, 100, 200);
```

Tracker Control

We recall that we used the Control class to create our RoundButton class object. We could've improved by using a ButtonBase class, since its functionality and structures are similar and thus easier to modify. You should always consider what base class to use, so that the modification of your inherited structure is much easier to achieve. You can also create a Control class from scratch, which is described later in the book. Sometimes neither of these work and you will need to create an inherited class with a base that is tiered higher on the control hierarchy, such as *System.Windows.Forms.Control* .

Defining Tracker Control

Think of a tracker as a custom-made control. In the PlayControl example, it is the slider at the top of the interface. It looks and functions similarly to a progress bar to indicate a relative value between two extremes (i.e. there are three variables, a minimum value, a maximum value, and a current position). This is an example that will grant you a lot of insight into GDI+ methods and functionalities.

For our tracker, the minimum defines the left, the maximum defines the right, and the progress indicator is somewhere in between, which can be determined using a percentage. A complete score listing for the tracker is included in Tracker.cs, part of Controls.dll package in VideoKiosk.sln.

Consider the range between your minimum and maximum is 2000. To avoid invalidation and repaint calls (up to 2000 unique calls), we instead use an arbitrary number of subdivisions and only repaint the progress indicator when there's a significant change.

SetStyle

We've made a considerable effort to create an appealing combination of foreground control elements and a background image in creating PlayControl. However, there is no native support for transparent backgrounds, which forces us into either making the controls look like the background or just let the background image show. The latter is a less resource-expensive option and happens to be the cleanest solution. To do this, we'll need to set up an attribute of the control called

ControlStylesSupport.SupportsBackTransparentColor . Setting this up for the tracker will allow the background to show through.

```
public Tracker()
{
this.SetStyle(ControlStyles.SupportsTransparentBackColor, true);
}
```

ControlPaint

ControlPaint is a GDI+ class that contains methods to draw 3D shapes such as buttons or sliders. This class is defined in the *Systems.Windows.Forms* and consists of methods for designing buttons, checkboxes, menu glyphs, radio buttons, size grips, etc. We'll use ControlPaint to make the visual interface of the Tracker.

Drawing a 3D Border

We want a 3D border for our Tracker. The Tracker class has a *DrawBar* method that draws a bar groove for the Progress Indicator to move along. This method takes a Graphics object as an argument and calls a *DrawBorder3D* method to operate on it.

```
protected virtual void DrawBar(Graphics graphics)
{
ControlPaint.DrawBorder3D(graphics, GetX(), GetY(),
GetWidth(), GetHeight(), Border3DStyle.Etched, Border3DSide.All);
}
```

This method is static, and thus requires no special ControlPaint object to be created for it. Boundaries for the region are specified using *GetX* , *GetY* , *GetWidth* , and *GetHeight* methods relative to the tracker's size. The last two arguments are enumerated values, selected from a list of available members according to our design considerations. This helps create a visual

style that indicates movement along the line, and thus we implement the etcborder-style along all sides of the border.

Drawing a Button Control

We can go about creating a button for the Tracker in multiple ways. One way involves scaling an image that is inserted into the control according to the size of the tracker. For instance, let this image be the shape of a button. To draw this, the control uses *Tracker.drawTracker* to call *ControlPaint.DrawButton* which draws the indicator.

```
protected virtual void DrawTracker(Graphics graphics)
{
ControlPaint.DrawButton(graphics,
TrackPosition(), (Height-8)/2, 5, 10, ButtonState.Normal);
}
```

The vertical offset is (Height – 8)/2, which puts the button squarely along the line. Tracking the horizontal position is more complex, and we will use *TrackPosition* to achieve this. *ButtonState.Normal* draws the button in an unpushed state.

Graphics.DrawImage

The buttons included in the PlayControl are of various types, differing most visibly by the image. Thus, instead of creating unique classes for each button, we can make one class with a subroutine to change what image it needs to display. The following code describes an *ImageButton* object, which is a derivative of RoundButton.

```
1: public class ImageButton : DarkButton
2: {
3: // TODO: Make sure that the image can be deleted!
4: private Image graphic = null;
5:
6: public ImageButton()
```

```
7: {
8:
9: }
10:
11: protected override Color Getcolor()
12: {
13: Color[] colors = {Color.MidnightBlue, Color.Black};
14: return colors[Convert.ToInt32(down)];
15: }
16:
17: private int GetX()
18: {
19: return (Width - graphic.Size.Width) / 2;
20: }
21:
22: private int GetY()
23: {
24: return (Height - graphic.Size.Height) / 2;
25: }
26:
27: private ImageAttributes GetImageAttribute()
28: {
29: ImageAttributes attribute = new ImageAttributes();
30: attribute.SetcolorKey(Color.White, Color.White,
31: ColorAdjustType.Default);
32: return attribute;
```

```
33: }
34:
35: private Rectangle GetRectangle()
36: {
37: return new Rectangle(GetX() + 1, GetY() + 1,
38: graphic.Size.Width, graphic.Size.Height);
39: }
40:
41: protected override void DrawGraphic(Graphics graphics)
42: {
43: base.DrawGraphic(graphics);
44: if( graphic == null ) return;
45:
46: graphics.DrawImage(graphic,
47: GetRectangle(), 0, 0,
48: graphic.Size.Width, graphic.Size.Height,
49: GraphicsUnit.Pixel,
50: GetImageAttribute());
51: }
52:
53: public Image Graphic
54: {
55: get
56: {
57: return graphic;
58: }
```

```
59: set
60: {
61: graphic = value;
62: Invalidate();
63: }
64: }
65: }
```

Of key importance is how to alter the image so that it fits into the button's display. For example, some parts of an image are perhaps unnecessary to display. To remove these parts, you can screen them out by defining an *ImageAttributes* object which defines what colors need to be removed. Also, notice how overloaded DrawImage is so as to account for a variety of input arguments.

ImageButton, now, uses the ImageAttributes object we created to screen our desired color (white, in this case) and removes it. If we want only to remove a specific part of the image, make that part match the same color as given to the ImageAttributes method.

Secondary Topics

We've given you a fairly solid overview of how VideoKiosk.sln works but do keep in mind that the listing for this structure, which includes Control.dll as well, is well over a few hundred lines of code. These would allow you to simplify your operation of making all of your control buttons from one base RoundButton object, but we wanted to show you how to use and appreciate GDI+. However, it does use a few structures from .NET to round out its functionality, which is tangential to our topic. We'll discuss them here for your perusal.

The Elapsed Time Clock

VideoKiosk.sln contains an Elapsed Time clock that is updated using a Timer tick. The Timer control isn't very resource-intensive, but it does get the job done for one small clock.

It works by raising a flag at time intervals defined in milliseconds. When this interval (*Timer.Interval*) completes, a *SetPosition* function changes the Tracker object's position using a value defined by *Player.Elapsed that* also updates the digital clock that expresses the Tracker's time. This works by defining a position in seconds so it can be used for the digital clock and the extreme values of the Tracker. We create the clock using a method called *TimeSpan* , which is initially provided the elapsed time in seconds:

```
private TimeSpan GetSpan(int position)
{
return new TimeSpan(0,0,0,position, 0);
}
private string GetMask()
{
return "{0}:{1,2:0#}:{2,2:0#}";
}
private string GetFormatted(TimeSpan span)
{
return string.Format(GetMask(), span.Hours,
span.Minutes, span.Seconds);
}
private void ChangePosition()
{
labelElapsed.Text = GetFormatted(GetSpan(Position));
}
```

This method also knows how to convert seconds to other larger units of time. Seconds are converted to minutes and hours automatically. *GetMask* provides a formatting string to control the appearance of TimeSpan, which is then used to update the clock. We'll discuss this formatting string soon.

Using TimeSpan to Tell Time

TimeSpan can measure time differences to an accuracy of 100 nanoseconds. This is a ”tick” of the TimeSpan, like the tick of any other clock. TimeSpan can be represented as either a positive or negative span of time that can measure out to days. The string that TimeSpan stores its time in is *d:hh:mm:ss.ff* , where *ff* refers to fractions of a second. If we needed this string we could use *TimeSpan.ToString()* to get time in that format. Performing arithmetic on *DateTime* values also yields a string like that.

Formatting Strings

We choose not to use TimeSpan.ToString() and instead do it manually for a more educational view into the string's inner workings.*Format* and formatting masks.

GetMask provides you with a formatting *mask* , which serves to receive data values from the string. This mask is “{0}:{1,2:0#}:{2,2:0#}”: values in the bracket are format characters that can be modified, and everything else is literals. The first value represents the zero-based argument to be changed, and any value after the comma represents the size of the argument. The third value represents the number and adds a zero to the number if there’s space for it (in the case of the number being too short). For example, {1,2:0#} represents the first replaceable parameter, length 2, and currently filled with 0.

Tooltip Control

Windows Forms contains a ToolTip structure. This structure is added to the component tray of a program, rather than the main form, which causes the Form’s controls to have the ToolTip property added: this allows you to display a tooltip textbox when the mouse hovers over a control. They are invisible components and do not show up during runtime automatically. Using it seems unintuitive at first, and Windows Forms does not seem to support it at a cursory glance, but it is as simple as adding it to the Component Tray.

Adding Controls to a Toolbox

We create a custom control to add a set of functionalities to a container. Adding a control automatically adds it to the Windows Forms tab of a toolbox.

Adding controls to a Toolbox requires you to create a Control Class library, into which you will add the methods you need. You can then access the Customise Toolbox option from the Toolbox context menu and add this library to the list you want to display in the Toolbox. This will be expounded upon in later sections of the book.

Catching and Handling Exceptions

We've learned how to work with exception flags using the resource protection block and the *try-finally* structure. VideoKiosk.sln used the *try-catch* structure instead. The Try block is the place where you place your main code to run, and the Catch block is where you add exception-handling code. If an exception is raised in Try, the Catch block is called. The basic syntax for this is:

```
try
{
// code to try
}
catch
{
// code to run if an error occurs
}
```

You can use multiple Catch blocks for multiple exceptions that each work for a particular class of exceptions. The following is an example of how to do so - in the method, SetMaximum is a function to catch an exception called *OverflowMaximum* the input integer is larger than 32 bits.

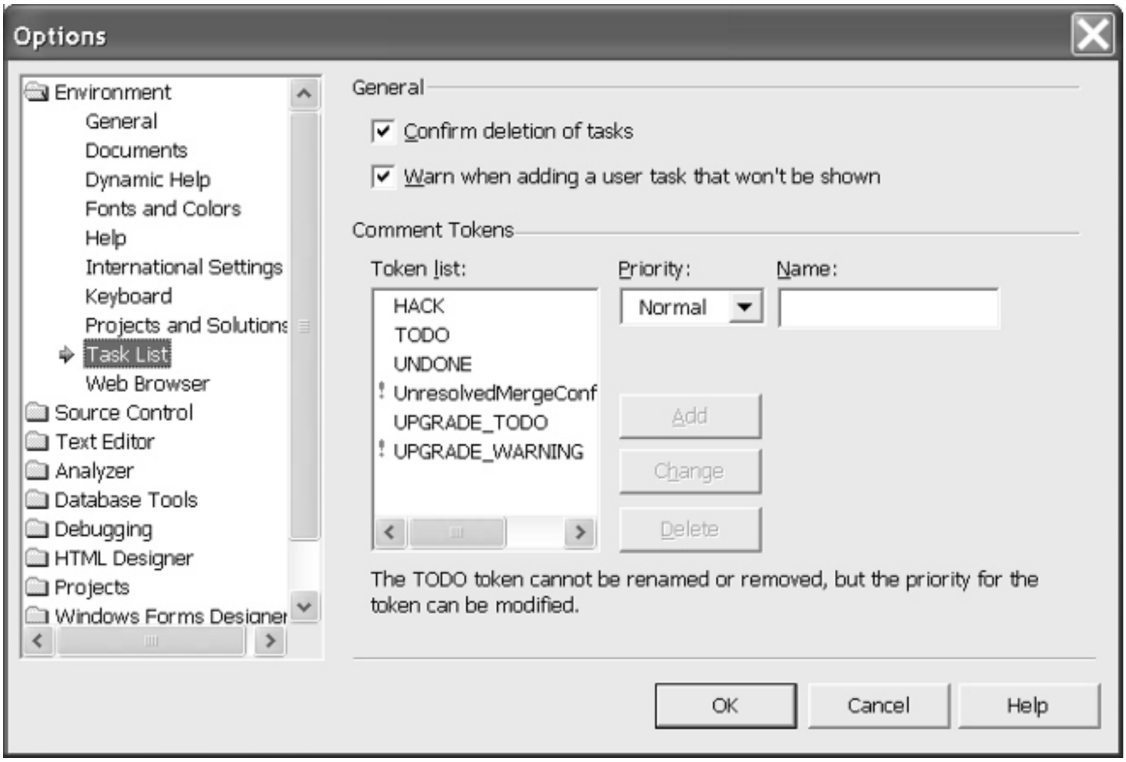
```
private void SetMaximum(double duration)
{
try
{
tracker1.Minimum = 0;
```

```
tracker1.Maximum = Convert.ToInt32(duration);
}
catch(OverflowException e)
{
MessageBox.Show(e.Message, "Media Error!",
MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
}
}
```

As a final note, do not use an exception handling structure if there isn't an exception to handle. This serves to clean up code and keep it lean.

Making a TODO List

You can add comments as reminders or headings in Visual Studio .NET using comment tokens, which are special keywords. These token keywords include *TODO* and any other



keyword we have defined. To use them, place them immediately after your double forward-slash comment token (//). This places the comments in a task list. We can look through this task list by going through View > Show Tasks > All, or View > Show Tasks > Comment, in the Visual Studio environment. To make custom tokens, define them in the Options list or the Environment > Task List page:

The following code demonstrates an update to the previous code for the ImageButton class, where we remembered to edit it using a comment. We'll add the ability to delete the Image property in the class, which is done by assigning it a *DefaultValueAttribute* .

```
1: public class ImageButton : DarkButton
2: {
3: private Image graphic = null;
4:
5: public ImageButton()
6: {
7:
8: }
9:
10: protected override Color Getcolor()
11: {
12: Color[] colors = {Color.MidnightBlue, Color.Black};
13: return colors[Convert.ToInt32(down)];
14: }
15:
16: private int GetX()
17: {
18: return (Width - graphic.Size.Width) / 2;
19: }
```

```
20:
21: private int GetY()
22: {
23: return (Height - graphic.Size.Height) / 2;
24: }
25:
26: private ImageAttributes GetImageAttribute()
27: {
28: ImageAttributes attribute = new ImageAttributes();
29: attribute.SetcolorKey(Color.White, Color.White,
30: ColorAdjustType.Default);
31: return attribute;
32: }
33:
34: private Rectangle GetRectangle()
35: {
36: return new Rectangle(GetX() + 1, GetY() + 1,
37: graphic.Size.Width, graphic.Size.Height);
38: }
39:
40: protected override void DrawGraphic(Graphics graphics)
41: {
42: base.DrawGraphic(graphics);
43: if( graphic == null ) return;
44:
45: graphics.DrawImage(graphic,
```

```
46: GetRectangle(), 0, 0,
47: graphic.Size.Width, graphic.Size.Height,
48: GraphicsUnit.Pixel,
49: GetImageAttribute());
50: }
51:
52: [DefaultValue(null)]
53: public Image Graphic
54: {
55: get
56: {
57: return graphic;
58: }
59: set
60: {
61: graphic = value;
62: Invalidate();
63: }
64: }
65:
66: }
```

The problem is resolved in line 52 of the above code. After this is done, remember to remove the TODO token from your code.

A tip for using comments is to add custom tokens to classify what kind of problem something is, and how it should be handled.

The Process Class

The Process Class, introduced in a previous section of the book, is used to actually receive an impetus for the rest of the code to run. *Process.Start* is a static method attached to the Click event handler for the leftmost ImageButton control on the PlayControl interface.

COM Interop

COM Interop (for interoperability) is a concept in .NET languages that uses COM libraries and controls in .NET environments since .NET uses Reflection and metadata in its assemblies to achieve the same goal. These concepts were explained in a previous section as well if you would like to review them.

Let's take the example of the ActiveX Media Player from VideoKiosk.sln. This is a COM-based control, and to use it we need to import these controls to our .NET toolbox by selecting Customize Toolbox from the Toolbox menu. This will import the type library for that COM control, and allows their usage in a .NET environment. This, of course, comes with the caveat that you will be using outdated COM structures, and you also will need to create an installer program to register these commands and methods.

To conclude the discussion we made so far in this chapter, GDI+ is an improvement and a streamlining of Windows' powerful GDI toolkit, making it much easier and more lenient to handle due to the advent of .NET and the addition of new methods and structures. GDI+ is used for imaging, typography, and vector-based graphics. The structures GDI+ uses are stateless and require constant streams of their information. Use CreateGraphics to create a Graphics object, or take the object received by Paint event handlers.

Chapter 4: Personalizing the Visual Studio .NET Framework Environment

There has been considerable investment and research made towards code automation capabilities for Visual Studio by Microsoft. This serves to provide quality-of-life changes to the way programmers can code things. We are given the capability to replace tedious and monotonous processes with automated Wizards, macros, and templates that we can create and use ourselves. These macros are extremely flexible owing to an extensibility model that lets you customize every detail of Visual Basic .NET. This chapter will not go over every detail that you can customize, but rather how to create objects such as Wizards and macros yourself, providing you with practical information that you can apply and develop yourself.

In this chapter, we will create project templates. Specific topics we will cover are:

- Creating Custom Wizards using IDTWizard.
- Creating a custom-fit Project Template.
- Registering your Wizards and Testing them.
- Writing Macros, using Macros IDE

Making a Wizard

A *Wizard* , colloquially, refers to a series of dialogues that are fed information to create a result. These structures have become immensely popular thanks to Windows, and the above definition is a result of that fame. Another more accurate definition is that a Wizard is a series of linear subprograms, as compared to spatial-based programming that existed at the time. The advantage that the linear model offers is that it reduces the accessibility of the program at a given step, which is beneficial in cases such as outcome-based tasks.

An esoteric definition for programmers is that a Wizard is an application or subprogram that breaks down a task or goal into multiple steps for the user to follow and complete, which is accomplished using IDTWizard. The user' interaction with the Wizard is very easy and they only need to click a Next button to continue most of the time. Developers have a harder time selecting a Wizard, since all of the back-end needs to be reformulated in

order to compress the commands into one button press. However, it's actually quite simple to use IDTWizard to create a Wizard, and the artistic vision of the programmer can be implemented into the console.

The following piece of code is a basic introduction to using IDTWizard to make a Wizard. A more advanced example is also provided later in this chapter.

```
using System;
using EnvDTE;
using System.Windows.Forms;
namespace HelloWizard
{
public class HelloWizard : IDTWizard
{
public void Execute(object Application,
int hwndOwner, ref object[] ContextParams,
ref object[] CustomParams, ref WizardResult retval)
{
MessageBox.Show("Hello Wizard");
retval = EnvDTE.WizardResult.WizardResultSuccess;
}
}
```

As you can see, a functional definition of a Wizard is a class that implements the IDTWizard interface, as defined in EnvDTE. There is only one method defined in IDTWizard: execute. Putting in the rest is up to the programmer, as is testing if it works.

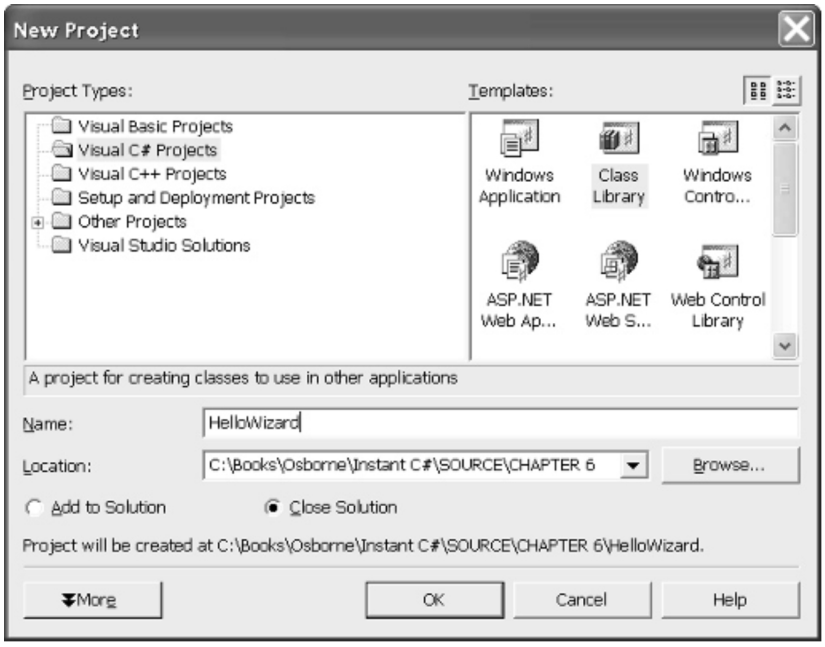
A Wizard Project

Wizards are basically a collection of classes, so to make one we'll need to create a class library. To start making a Class Library Project, go to the New

Project dialog, select the Class Library template, name your project, and press OK. You're now on your way to making your own Wizard.

Using IDTWizard

To use IDTWizard to make your Wizard you will need to reference the design-time environment, i.e. Visual Studio .NET's extensibility model. To do so, add a reference to the envdte.dll assembly to the project's references and refer to the library with the keyword.



There is only one function in IDTWizard, *Execute* , and it must be provided an array of context, an array of custom parameters for it to use, an owner handle, and a return value that uses a pass-by reference modifier.

Pass-By Referencing

We need to create a method that returns a value of a datatype other than void. The parameter that this method returns can also be given the keyword *ref* to allow it to return multiple values, by virtue of them being able to be affected by the caller. Equivalents to *ref* in other languages include passing pointers in C++, *ByRef* in VisualBasic .NET, and *var* in Delphi. An example code will describe this method below.

Testing the Wizard

After adding your functionality to the Wizard, you can run it to find out if it works. The Singleton object DTE contains a method called *LaunchWizard* ,

which does exactly that (specifics are given later in the chapter). The next listing of code is a macro for testing the HelloWorld Wizard we made earlier, and the one after that is an example of stuff that you can put in a Wizard.

```
Imports System
Imports EnvDTE
Imports System.Diagnostics
Public Module Module1
Public Sub Test()
Const Path As String = _
"C:\Temp\HelloWizard.vsz"
Debug.WriteLine("This is running!")
Dim ContextParams(1) As Object
ContextParams(0) = Nothing
DTE.LaunchWizard(Path, ContextParams)
End Sub
End Module
```

This macro can be found in Macro IDE: go to Tools > Macros > Macros IDE to open the menu. Now we need to create a Wizard launch file. This launch file has the extension *.vsz* and has the same name as the Wizard's class library.

```
VSWIZARD 7.0
Wizard=HelloWizard.HelloWizard
Param=<Nothing>
```

Place this code in a file named HelloWizard.vsz, and place the file in the same directory as the HelloWizard.dll. HelloWizard.dll has been registered

using regasm.exe, a Visual Studio .NET utility. The process of registering your Wizards will be explored later in the chapter.

Creating a Project Template for Visual Studio .NET Wizards

Visual Studio .NET contains a lot of methods, tutorials, templates, etc. to help you create all kinds of projects. It does not, however, have a template for a Wizard. We will show you how to create a template yourself that you can then access using the New Project dialog.

Visual Studio .NET contains a basic Wizard called VsWizard.dll that you can use to create project templates. It can be found in "…/Microsoft Visual Studio .NET/Common7/IDE/VsWizard.dll". This wizard uses sample files, scripts, and other inputs to create a project from a predefined template. Thus, we can use it to create a template which saves us from writing another engine to do so, but we also need to feed it a set of rules, files, and samples to follow. Our process for creating a project template includes, in no particular order, the following steps:

- Find and use a pre-existing template that is as close to our goal template as possible. We will use the Class Library template, since a Wizard is already a class library or DLL.
- Modify the pre-existing template by adding new files. For our Wizard we'll need to add a .cs file, containing a class that uses IDTWizard. We can use our HelloWizard code for this.
- Update the templates.inf list that contains the files used by our new template.
- Rewrite common.js (.js being Jscript .NET), a structure containing generic methods used by VsWizard.dll. We will need to add code to it, in order to make a Class Library project that references envdte.dll.
- Rewrite default.js to include our template's generic methods.
- Add an icon for our template.
- Modify the VsWizard.dll launcher so that it is associated with our template.
- Update VSDir to contain our project template, so that it shows in the New Project Dialog.

This process is, admittedly, quite tedious, but this is a generic and therefore extensible list, which can be used to create any project template that you

would like to. The following sections will expand upon this checklist, along with how to complete the task that each asks of you, and a further description of the process. We will also follow the checklist in order.

Creating a New Template from a Pre-Existing Template

Project templates are the methods represented by the icons in the New Project dialog. You can also create an entire template from scratch as a text file, but this is easier. Take care to consider what project template you will base yours off as different versions of Visual Studio have different templates.

To create a template, select one close to yours, which in our case is the Class Library template. To access this template, go to "…Files\Microsoft Visual Studio .NET\VC#\

VC#Wizards". Here, the Class Library is represented by CSharpDLLWiz. Follow the below steps in order to create a new template using this one.

- Copy the library folder to the same directory, which will name it "Copy of CSharpDLLWiz" by default.
- Name it whatever you want. We recommend naming this folder CSharpDLLWizWiz, keeping the convention

These folders have a Scripts and Templates subdirectory that contains a language identifier. For English users, this identifier is most likely in 1033. The scripts\1033 folder contains the template's default.js file, which we will learn to modify later. The templates\1033 folder contains actual code files that the engine will use to create a template. We will discuss how this works just after this section. Remember that right now these are still just the same files from the Class Library template.

Your folder should start to look something like this:

```
…\Microsoft Visual Studio .NET\VC#\VC#Wizards
CSharpDLLWizWiz\Scripts\1033
default.js
CSharpDLLWizWiz\Templates\1033
assemblyinfo.cs
```

```
file1.cs
Templates.inf
```

Adding Files to the Wizard Library Template

We can add files that we want for our template to Templates\1033. For now, we don't need to add any such file and we can simply use the existing contents. We do, however, need to change the file1.cs file here so that it contains code for a Wizard template. It would look something like this:

```
using System;
using EnvDTE;
namespace [!output SAFE_NAMESPACE_NAME]
{
/// <summary>
/// Summary description for [!output SAFE_CLASS_NAME].
/// </summary>
public class [!output SAFE_CLASS_NAME] : IDTWizard
{
public [!output SAFE_CLASS_NAME]()
{
//
// TODO: Add constructor logic here
//
}
public void Execute(object Application, int hwndOwner,
ref object[] ContextParams,
ref object[] CustomParams, ref wizardResult retval)
{
```

```
//
// TODO: Add your custom code here
//
retval = EnvDTE.wizardResult.wizardResultSuccess;
}
}
}
```

Notice how this is very similar to our original HelloWizard code, with the addition of SAFE_CLASS_NAME replaceable tokens. Again, you don't need to change any files in this folder, but you can inspect them to see what they do and modify them if you want to.

Modifying default.js

default.js is stored in Scripts\1033. To modify it for our template, we need to make one small but important modification to it that allows our Wizard projects to reference envdte.dll. We need to coordinate this with common.js as well, which is described in the next section. For now, let's see how we will modify default.js:

```
1: // (c) 2001 Microsoft Corporation
2:
3: function OnFinish(selProj, selObj)
4: {
5: var oldSuppressUIValue = true;
6: try
7: {
8: oldSuppressUIValue = dte.SuppressUI;
9: var strProjectPath = wizard.FindSymbol("PROJECT_PATH");
10: var strProjectName = wizard.FindSymbol("PROJECT_NAME");
```

```
11: var strSafeProjectName = CreateSafeName(strProjectName);
12: wizard.AddSymbol("SAFE_PROJECT_NAME", strSafeProjectName);
13:
14: var bEmptyProject = 0; //wizard.FindSymbol("EMPTY_PROJECT");
15:
16: var proj = CreateCSharpProject(strProjectName,
17: strProjectPath, "defaultDll.csproj");
18:
19: var InfFile = CreateInfFile();
20: if (!bEmptyProject)
21: {
22: AddReferencesForWizard(proj);
23: AddFilesToCSharpProject(proj,
24: strProjectName, strProjectPath, InfFile, false);
25: }
26: proj.Save();
28: }
29: catch(e)
30: {
31: if( e.description.length > 0 )
32: SetErrorInfo(e);
33: return e.number;
34: }
35: finally
36: {
37: dte.SuppressUI = oldSuppressUIValue;
```

```
38: if( InfFile )
39: InfFile.Delete();
40: }
41: }
42:
43: function GetcSharpTargetName(strName, strProjectName)
44: {
45: var strTarget = strName;
46:
47: switch (strName)
48: {
49: case "readme.txt":
50: strTarget = "ReadMe.txt";
51: break;
52: case "File1.cs":
53: strTarget = "Class1.cs";
54: break;
55: case "assemblyinfo.cs":
56: strTarget = "AssemblyInfo.cs";
57: break;
58: }
59: return strTarget;
60: }
61:
62: function DoOpenFile(strName)
63: {
```

```
64: var bOpen = false;
65:
66: switch (strName)
67: {
68: case "Class1.cs":
69: bOpen = true;
70: break;
71: }
72: return bOpen;
73: }
74:
75: function SetFileProperties(oFileItem, strFileName)
76: {
77: if(strFileName == "File1.cs" || strFileName == "assemblyinfo.cs")
78: {
79: oFileItem.Properties("SubType").Value = "Code";
80: }
81: }
```

The template wizard is designed to use external JScript functions. There are four of them mentioned here: *OnFinish* , *GetcSharpTargetName* , *DoOpenFile* , and *SetFileProperties* . These are used exclusively by the template wizard to call for the Wizard Library.

- SetFileProperties sets up a flag to mention that the .cs files contain code.
- DoOpenFile opens the class module file in Visual Studio .NET.
- GetcSharpTargetName changes the name of the file. We use it to change file1.cs' name to class1.cs instead.

- OnFinish completes the substitutions for the methods and parameters in the file, and ties up the package.

The small change that we needed to make to file1.cs is replace *AddReferenceForClass(proj)* to *AddReferenceForWizard* on line 22. This method isn't defined anywhere yet and the next section modifies common.js to define and include this method.

Modifying common.js

The structure of the Project Template is based on the shared wizard engine, the folder structure, the template files, and two JScript files that combine to create a project template. This is dense, unwieldy system, but it is what we have.

common.js is a script common to all templates, stored in ..\Microsoft Visual Studio .NET\VC#\VC#Wizards\1033. We have to modify this file to include references to our new template and subsequent projects. Add the following code to common.js to include the AddReferenceForWizard method.

```
Description:
oProj: Project object
**********************************************************/
function AddReferencesForWizard(oProj)
{
var refmanager = GetcSharpReferenceManager(oProj);
var bExpanded = IsReferencesNodeExpanded(oProj)
refmanager.Add("System");
refmanager.Add("System.Data");
refmanager.Add("System.XML");
refmanager.Add("envdte");
if(!bExpanded)
CollapseReferencesNode(oProj);
```

```
}
```

We made this using the AddReferenceForClass method, adding a call to the refmanager to refer to envdte.dll. We can do this because the Class Library structure is already very similar to ours. If we're making a template from scratch, we would need to create other methods and classes.

Now all that's left is to create a VSDir file and the Wizard Launcher file.

The Wizard Launch File

We've already discussed how to create a Wizard using IDTWizard in a previous section titled "Creating a Custom Wizard". However, we do not need to create a Wizard from scratch. We'll modify the Class Library template's Wizard and use that. We will only be telling the New Project dialog the Wizard it needs to select. The Wizard Launch files for all Project Templates are stored at the directory "…\Microsoft Visual Studio .NET\VC#\CSharpProjects". The Wizard Launch file for the Class Library is stored in *CSharpDLL.vsz* . Copy this file, rename it to something like *CSharpDLLWiz.vsz* , and modify the code within to refer to our template, as follows:

```
VSWIZARD 7.0
Wizard=VsWizard.VsWizardEngine
Param="WIZARD_NAME = CSharpDLLWizWiz"
Param="WIZARD_UI = FALSE"
Param="PROJECT_TYPE = CSPROJ"
```

The only change we made to the original file is to change the Wizard to refer to our template's wizard. The other parameters are just conditional values. Now all that's left is to create VSDir and test our template.

Creating a VSDir File

These files define what information is received by the New Project and Add Item dialogs and how items in those lists are displayed. They require two things - an icon and a name. Specify the connection between these two

things and the template you want them to refer to. The listing for VSDir is as below, written in a single line - we will explain how this works after:

```
CSharpDLLWiz.vsz|{FAE04EC1-301F-11d3-BF4B-
00C04F79EFBC}|Wizard Library|
20|#2323|{FAE04EC1-301F-11d3-BF4B-00C04F79EFBC}|4547|
|WizardLibrary
```

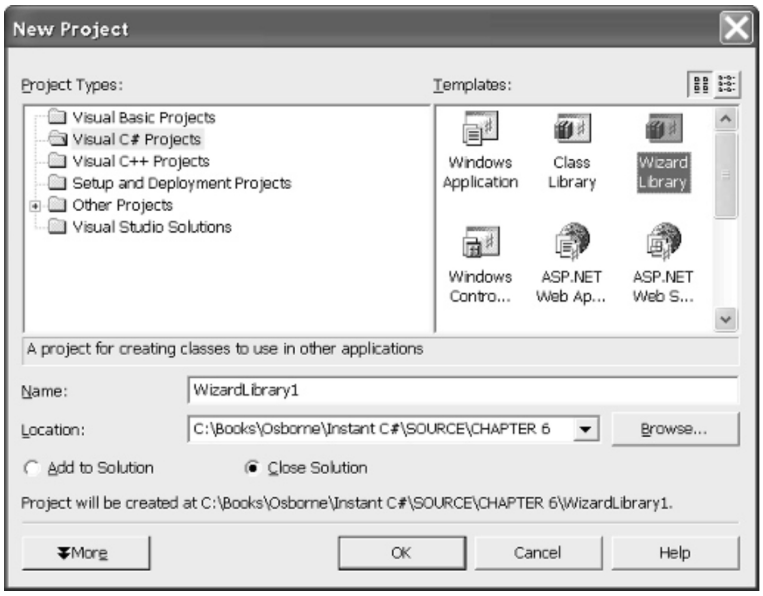
This one line contains the existing project template's items. VSDir files are stored in "…\Microsoft Visual Studio .NET\VC#\CShaprProjects\CSharpEx.vsdir" and this folder will also contain your Wizard Launch file.

This one line is divided into fields by "pipes" (the " | " symbol). The following table breaks these fields down in order of appearance, and describes what they do.

Relative Path Name	CSharpDLLWiz.vsz	Specifies the wizard launch file using a relative path. Store the .vsz file in the same directory as the .vsdir file.
{FAE04EC1-301F-11d3-BF4B-00C04F79EFBC}	CLSID package	A GUID representing a resource DLL. This particular GUID refers to something named the C Sharp Editor Factory.
Wizard Library	Localized name	This is the template's localized name and the name that will appear in the New Project dialog.
20	Sort priority	Determines the order in which elements are

		listed. A sort priority of 1 is the highest.
#2323	Description	A string or resource identifier containing a localized text description of the template.
{FAE04EC1-301F-11d3-BF4B-00C04F79EFBC}	DLLPath or CLSID package	Contains a GUID reference to an EXE or DLL resource that contains icons for the template.
4547	Icon resource ID	Indicates which icon to use from the resource file referred to from the previous entry.
<blank>	Flags	Bitwise flags, not used by our template. You can refer to the Visual Studio .NET help for VSDir files for individual flag values.
WizardLibrary	Suggested base name	Specifies the default name for the template. Our entry represents a project, therefore, the root name for Wizard Library projects will be WizardLibrary. The first new project will be WizardLibrary1, and so on.

This template is used to create the following entry in the New Project dialog box:



Now that everything is done, lets test our template by creating a new project. Go to File > New > Project, click on the Wizard Library icon, and press OK. You should now have a new Class Library project with an assemblyinfo.cs file and a file named WizardLibrary1.cs, which should contain a basic execution of IDTWizard. Do double check that a reference to envdte.dll exists.

A final note for this section regarding the project's template is that it's a combination of shared files from across the base class and the environment. An important one of these is DefaultDLL.csproj file, which is located in "…\Microsoft Visual Studio .NET\VC#\VC#Wizards folder. All DLL-based projects refer to this template. Also, most of the process of creating a template is copy and pasting things, rather than creating things from scratch. Make a template if you are going to use a custom project structure often to simplify tedious tasks and streamline your process.

Now that all of this works, let's use these to make something that automates tasks for you.

Extending Visual Studio .NET with Wizards

A wizard is a simple way to create a task automation that benefits newer programmers. Creating a wizard allows your team to use the more unique aspects of .NET programming more easily.

Let's make a wizard that creates a *strongly-typed collection* , a collection that inherits from *ReadOnlyCollectionBase* or *CollectionBase* . Methods in these classes include *IList* , *IListSource* , *IEnumberable* , which results in you creating a collection structure that behaves like an array and can be sorted and used by a DataGrid control and can be compiled into an ADO.NET dataset. This wizard already exists and it is named *TypedCollectionWizard* and it implements methods from System.Collections.CollectionBase. It works by creating a new class that inherits from the previous one, adding an indexer and an Add method. We will use this example to understand how to make macros and program against the Common Environment Object model.

Writing a Macro

We now already have a basic understanding of how to create, write, and test wizards, which will serve us well in writing macros as well, alongside things such as C# programming and Windows Forms interfacing. Let's continue with how to write a macro.

Macro support is a part of Visual Studio .NET's extensibility model, alongside but separate from Macros IDE. Macros are written using Visual Basic .NET. Macros IDE is a project-centric environment, and the macro files created are stored with *.vsmacro* extensions, though Visual Basic code can also be communicated to it.

To start, open Macros IDE from Tools > Macros > Macros IDE. Using this is similar to Visual Studio .NET. In this new environment, select the Project Manager from the View menu. To create a new macro, select an existing module or create a new one (a *module* is a sourcecode file). Add a simple subroutine (read: method) to it and see if it works by starting a debug process from Debug > Start. Here's an example of a macro which simply produces a message box that says "Hello World!":

```
Sub HelloWorld()
MsgBox("Hello World!")
End Sub
```

Let's move on to a more practical example of a macro that adds a new class to an existing project, similar to how we will reconstruct TypedCollectionWizard later. The following macro does exactly that: it adds a new class file to an active project's directory, and the second (empty) entry is where the name of the new class would go. By default, an appropriate name is produced by the method itself:

```
Sub TestNewFile()
DTE.ItemOperations.AddNewItem( _
"Local Project Items\Class", "")
End Sub
```

Notice the DTE namespace keyword. The following section is devoted to giving a brief overview of this keyword, referring to the design-time environment.

The Common Environment Model

A DTE is a design-time environment object, referred to by internal documentation as a Common Environment Model, or the automation model, or the extensibility model. We will continue to refer to it as a DTE. It is the root of a project, representing an object model allowing access to the capabilities of Visual Studio .NET. DTE uses interfaces to create an object-oriented accessibility within the Visual Studio .NET environment, which is, in effect, a method for automation. The documentation for this model is very large, and perusal is at one's own risk.

For our current concern, we'll only use a small section of that manual to apply DTE for macros and C# code, which should serve to introduce you to the core concepts of using it. A good way to think about DTE is as a mask that interacts with Visual Studio .NET, including menus, documents, macros, etc.

Writing Code, using Macros

We've learned how to reuse Project templates to generate new code, but we can also change code on the fly during runtime using DTE. This would look something like as follows:

```
```

```
1: Public Sub TestWriteCode()
2: Try
3: Dim item As ProjectItem = _
4: DTE.ItemOperations.AddNewItem( _
5: "Local Project Items\Code File")
6:
7: Dim defaultNamespace = _
8: item.ContainingProject.Properties. _
9: Item("DefaultNamespace").Value
10:
11: Dim code As CodeNamespace = _
12: item.FileCodeModel.AddNamespace(defaultNamespace)
13:
14: Dim editPoint As EditPoint = _
15: code.StartPoint.CreateEditPoint
16:
17: editPoint.LineDown(2)
18:
19: editPoint.Insert(DateTime.Now.ToString() + vbCrLf)
20:
21: Catch e As Exception
22: MsgBox(e.Message, MsgBoxStyle.OKOnly, "Error")
23: End Try
24: End Sub
```

What this code does is that it:

- creates a C# file, by calling for a new one and obtaining a reference to the ProjectItem for it.
- adds a default namespace, and formats a namespace element.
- inserts text into that file, by assigning edit points. Text can be added here either manually or using Add methods.

This file is added to the current project, but it cannot create a project on its own. Running this code results in the following:

```
namespace TestWizardOuput
{
5/9/2002 5:12:23 PM
}
```

The TypedCollectionWizard uses an incomplete template of code alongside a DTE to generate its strongly-typed collection. Let's move on to how we can generate functional code automatically.

Code Generator

You can create a code generator by centering it around structures that exist in all instances of your code. Add string-based inputs to work with these structures and add dynamic functionality to it. For example, the templates' code generator Wizard is written as follows:

```
1: using System;
2:
3: namespace TypedCollectionWizard
4: {
5: /// <summary>
6: /// Summary description for Wizard.
7: /// </summary>
8: public class Wizard
9: {
```

```
10: /// <summary>
11: /// Parameter 0 is the class name; parameter 1 is the type name;
12: /// parameter 2 is the optional setter; parameter 3 is the comment.
13: /// </summary>
14: private const string template =
15: " // Code generator by pkimmel@softconcepts.com\r\n" +
16: " {3}\r\n" +
17: " public class {0} : System.Collections.CollectionBase\r\n" +
18: " {{\r\n" +
19: " public {1} this[int index]\r\n" +
20: " {{\r\n" +
21: " get{{ return ({1})InnerList[index]; }}\r\n" +
22: " {2}" +
23: " }}\r\n\r\n" +
24: " public int Add({1} value)\r\n" +
25: " {{\r\n" +
26: " return InnerList.Add(value);\r\n" +
27: " }}\r\n" +
28: " }}\r\n";
29:
30: private const string setter =
31: "set{ InnerList[index] = value; }\r\n";
32:
33: private string name;
34: private string typeName;
35: private bool hasGetter = true;
```

```
36: private bool hasSetter = true;
37: private string comment;
38:
39: public Wizard(string name, string typeName, bool hasGetter,
40: bool hasSetter, string comment)
41: {
42: this.name = name;
43: this.typeName = typeName;
44: this.hasGetter = hasGetter;
45: this.hasSetter = hasSetter;
46: this.comment = comment;
47: }
48:
49: private string GetSetter()
50: {
51: return hasSetter ? setter : "\r\n";
52: }
53:
54: private string Getcode()
55: {
56: return
57: string.Format(template, name, typeName,
58: GetSetter(), comment );
59: }
60:
61: public string GetTemplate
```

```
62: { get{ return template; } }
63:
64: public string Code
65: { get{ return Getcode(); }}
66:
67: }
68: }
```

This code is fairly straightforward, making a template that uses parameters that vary from class to class, and using them when needed upon calling Getcode. More importantly, we can use this standalone to test whether it works for the Wizard. This results in the following code, which is syntactically correct and works as intended for the Wizard.

```
public class Customers : System.Collections.CollectionBase
{
public Customer this[int index]
{
get{ return (Customer)InnerList[index]; }
set{ InnerList[index] = value; }
}
public int Add(Customer value)
{
return InnerList.Add(value);
}
}
```

Let's now make an interface to interact with this code:

Implementing the Wizard UI

We've used the Add Indexer Wizard's User Interface in our example. To implement it, we've used a few tricks here and there. The following section discusses them:

User Interface Strategies:

We use property methods to interact with controls and any change in the control only requires a singular change in its reference. The following is the code for the name of the collection typed into the Textbox UI.

```
private string ClassName
{ get{ return textBoxClassName.Text; }}
```

Any changes to the control of the Textbox only need to be referred in one place in the GUI code.

Another thing you can do is to validate field input in the UI as well as the class, using regular alphanumeric expressions. The following code confirms this.

```
private void DoValidateField(Control control, string value,
string expression, string errorText)
{
if( !Regex.IsMatch(value, expression))
```

```
{
control.Focus();
throw new Exception(errorText);
}
}
private void ValidateClassName()
{
DoValidateField(textBoxClassName,
ClassName, @"^\w$|^\w[A-Za-z0-9]+$",
"Valid class name is required");
}
```

Dialog Form Strategies

We use FormWizard to create a dialog form. A tip for dialog forms is to make a static method, called *Execute* , which accepts inputs and returns outputs for the form. The user only needs to worry about the inputs and outputs, not the form as Execute handles them. The following demonstrates:

```
public static bool Execute(ref string generatedCode)
{
using(Form1 form = new Form1())
{
if( form.ShowDialog() == DialogResult.OK)
{
generatedCode = form.Code;
return true;
}
else
```

```
return false;
}
}
```

Ultimately, the user will be the TypedCollectionWizard. When IDTWizard is implemented it only needs to use this static method, feed it an input, and it will return an output, which it can then use. This can also be applied to Windows Forms applications. The alternative involves rewriting and re-evaluating the code, in effect writing the code twice.

The form is displayed using the following piece of code.

```
string generatedCode = string.Empty;
If(FormWizard.Execute(generatedCode))
// do something
```

Implementing the Wizard

We've set up the code generator using a static method that requires nothing more than an input to function. Everything is handled internally. With this completed, the rest of the Wizard is fairly easy to implement. Adding functionality to a Wizard occurs by using the IDTWizard.Execute method to implement our code, such as follows:

```
1: using System;
2: using System.IO;
3: using EnvDTE;
4: using System.Windows.Forms;
5:
6: namespace TypedCollectionWizard
7: {
8: /// <summary>
9: /// Summary description for Class1.
```

```
10: /// </summary>
11: public class CollectionWizard : IDTWizard
12: {
13: // Implements IDTWizard.Execute
14: public void Execute(object Application,
15: int hwndOwner, ref object[] ContextParams,
16: ref object[] CustomParams, ref wizardResult retval)
17: {
18:
19: string generatedCode = string.Empty;
20: if( FormWizard.Execute(ref generatedCode))
21: {
22: WriteCode((DTE)Application, generatedCode);
23: retval = EnvDTE.wizardResult.wizardResultSuccess;
24: }
25: else
26: {
27: retval = EnvDTE.wizardResult.wizardResultCancel;
28: }
29: }
30:
31: private void WriteCode(DTE dte, string generatedCode)
32: {
33: try
34: {
35: DoWriteCode(dte, generatedCode);
```

```
36: }
37: catch(Exception e)
38: {
39: MessageBox.Show(e.Message, "Error Writing Code",
40: MessageBoxButtons.OK, MessageBoxIcon.Error);
41: }
42: }
43:
44:
45: private void DoWriteCode(DTE dte, string generatedCode)
46: {
47: ProjectItem item = dte.ItemOperations.AddNewItem(
48: @"Local Project Items\Code File", "" );
49:
50: string defaultNamespace =
51: (string)item.ContainingProject.Properties.Item(
52: "DefaultNamespace").Value;
53:
54: CodeNamespace code =
55: item.FileCodeModel.AddNamespace(defaultNamespace, 0);
56:
57: EditPoint editPoint = code.StartPoint.CreateEditPoint();
58: editPoint.LineDown(2);
59: editPoint.Insert(generatedCode);
60: }
61: }
```

```
62: }
```

This code completes using IDTWizard, displays the form, writes code using the code generator if Execute is true using WriteCode, which manages error handling using a try-catch structure, and generates a code file using DoWriteCode.

We've used DTE in this code by casting an Application parameter onto a DTE object, and using that to generate source code.

This demonstrates the automation capabilities of Visual Studio .NET, and it is one of the reasons it is so beloved by software programmers. Now to test the Wizard!

Completing the Sample Wizard Launch File

Creating a Wizard Launch file is simple and we only need to provide the wizard's name and mention that it doesn't need to be passed arguments. This file has the extension .vsz, and it is read every time the Wizard is loaded. The following describes the Wizard Launch file for TypedCollectionWizard, which is saved in the same directory as its library file. This isn't necessary, but it's an easy shortcut.

```
VSWIZARD 7.0
Wizard=TypedCollectionWizard.CollectionWizard
Param=<Nothing>
```

1. The first line must be added as is.
2. The second line defines the assembly name, and the class name that contains the IDWizard implementation.
3. The other parameters define what kind of parameters are to be sent to the Wizard, if any.

Let's move on to a more complicated example. The following is the Wizard Launcher file for the C# Indexer Wizard that works alongside the Wizard Engine. You can pass any kind of parameters into the Wizard using this, such as the Wizard's name and the Project type.

```
VSWIZARD 7.0
```

```
Wizard=VsWizard.VsWizardEngine
Param="WIZARD_NAME = CSharpIndexerWiz"
Param="PROJECT_TYPE = CSPROJ"
```

Registering Wizards

The Wizard requires access to the Registry, but does not need to be a member of the Global Assembly Cache (or GAC). To this end, we can register it using *regasm* as follows:

```
c:\winnt\Microsoft.NET\v1.0.3705\regasm
c:\temp\TypedCollectionWizard.dll /codebase
```

This is the default path for the .NET framework. If there's a different directory in place of *c:\temp* , use that. /codebase stores these path in the registry using *file://* . Again, the Wizard is still a .NET structure, only accessing the registry. It is not a COM component. There will be an explanation at the end of the chapter.

A quick note, the GAC is a cache for *all* the assemblies being used by the machine. A Wizard is not a permanent part of that list, and thus it doesn't need to be registered. The GAC is, in effect, a giant folder that contains and catalogues assemblies for easy use. We recommend looking into a .NET utility called GACutil.exe.

Testing the Wizard using a Macro

Let's use our macro that we made at the beginning of the chapter to test our Wizard. Replace the macro's *constPath* variable with the address to the TypedCollectionWizard, and run it.

Running the Wizard from the Command Window

We can use our macro from the previous section to run our Wizard from the Command Window. Select View > Other Windows > Command Window. We've created a test macro called MyMacro, stored it in a module named Module1, and named the method inside TestCollectionWizard. All of this combined, we call our Wizard with this line of code:

```
```

```
Macros.MyMacros.Module1.TestCollectionWizard
```

A list of commands will pop up when you type the word Macros, via Intellisense.

Secondary Topics

The chapter introduced into a wide variety of tools and concepts such as JScript, gacutil.exe, regasm.exe, etc. These are structures with extensive documentation, and a more thorough introduction to these concepts is in order:

Returning to JScript

We used a bit of JScript to implement our templates, which is done by calling an external compiler. However, JScript is a completely different and just-as-fleshed-out language as compared to Visual Basic .NET and C#. You're likely to find code written in this language in things such as clientside code for webpages and ACT (a testing automation application developed by Microsoft). The following checklist describes how you may go about writing some JScript .NET code.

- Open NotePad.
- Write *print.(" Hello World!")* .
- Save this file as HelloWorld.js, and remember its directory. Close the file.
- Open the command prompt, and run the file by calling the compiler as follows:

```
c:\winnt\Microsoft.NET\Framework\V1.0.3705\jsc Hello.js
```

Using the regasm Utility

Wizards are an important part of Visual Studio .NET that are attached to it using COM. To do so, we use the regasm utility. Two forms that works best for this are:

```
regasm assemblyname /codebase
regasm assemblyname /unregister
```

1) *regasm* simply refers to the utility. *assemblyname* is the name of the assembly containing the Wizard DLL. /*codebase* adds the code to the registry. It is technically all that is required to make a complete call, though it does provide a warning to use a strongly-worded name (by using sn.exe, for example) to avoid version problems.
2) Similar to the above, except it uses */unregister* to unregister the Wizard from the registry.

The wizard does not use COM. It is required to register the DLL into the registry to load it into the Visual Studio .NET environment.

To read the registry assembly, do:

```
regasm assemblyname /regfile:regfile
```

where assemblyname would refer instead to your assembly and the second regfile refers to a .reg extension file.

Programming has come a long, long way from what it was ten years ago, and it continues to grow and develop at an exponential pace. We've developed things such as exception handlers, that have a contingency plan for pretty much every situation that we can imagine.

As we've mentioned, the field of programming is ballooning at an extremely rapid pace, and it's quickly becoming less and less probable that one person can know everything about programming. Specialization is the norm now as it is simply impossible for someone to know everything about everything. Just in this one chapter, we've touched on three massive languages - Visual Basic .NET, JScript (or JavaScript), C# - and utility programs GACutil.exe and regasm.exe. A programmer now only has a basic understanding of the underlying concepts but needs to opt in to one of these fields. They can focus on game development, or web design, or even more utilitarian application/software programming. Anything you can imagine.

Automation also is beginning to form a real concern to programmers (and humans as a whole), based on the strength of modern computers and the speed at which they can grasp concepts given a ruleset. In games requiring intellect and complex decision-making, such as chess, the Chinese game Go, and DotA 2, programs that have used advanced automation to repeat

processes and develop optimal strategy have far overtaken the capability of humans. Programming can be left to programs themselves, now.

Conclusion

We have now come to the end of our journey. Even though you might have felt C# to be a bit difficult and at times, frustrating to understand, regardless you have increased your programming knowledge. In the world of programming, experience is the best teacher you can ever find. While we have tried our best to provide you with adequate explanations of concepts and advanced topics, you might still feel that something was missing. If that's the case, then you are on the right track of learning. Anything that you feel is ambiguous, will serve as an impetus to a specific topic for research which will expose you to even more things, causing a domino effect. Every chapter in this book has tried to cover all of the aspects of C# programming to a certain extent, however, it is out of the scope of this book to completely dominate every advanced programming element of C#. Thus, it is important to provide the readers with the C# programming's most important central information, if not the complete package. It is highly recommended that you research the aspects which confuse you, this will help clarify your concepts at a level which would otherwise have not been possible.

In short, we have explored the fundamental principles of object-oriented programming in the first chapter and learned how C# implements these principles through various functions. In the next chapters, we discovered the extended functionalities in C# programming such as using the 'Reflection' function in the .**NET** framework to extract information from assemblies and use them. Finally, we move on to truly advanced C# programming topics such as using '**GDI+** ,' working with Visual Studio .NET and the Event logger, and last but not least, understanding how to effectively use '**UserControls** ' to create the interface of an application. All of these topics collectively define the C# programming paradigm and we hope that the reader learns and understands the core concept of every topic and discussion in these chapters. Exposure to a programming language can go a long way in a person's programming career as someone might have to reverse engineer a mechanism built using programming languages like C#, C++, etc. So it's recommended always to have the eagerness to learn a programming language no matter how intriguing it may seem.

References

Advanced C# Programming by Author: **Paul Kimmel**