PREV
Chapter 1 Manage program flow

NEXT
Chapter 3 Debug applications and implement security

CHAPTER 2
Create and use types

The first thing you have to learn when you start programming is how to create the instructions that tell the computer what to do with incoming data. In Chapter 1, "Manage program flow," the focus was on how to control the flow of execution through code in terms of single programs, but also when to use threads, tasks, and events.

In this chapter the focus is on how to use object-oriented techniques to make it easier to create large scale solutions that contain many different elements that can be worked on by a large number of developers.

You will learn how to create custom data types to hold information that matches the problem requirements. These types may also have behaviors, allowing them to operate as software components that have integrity and provide services for other objects.

You will also learn how to programmatically generate object behaviors, and how to manage the life-cycle of objects in a solution. Finally, you'll discover the powerful features behind the string type provided as part of the C# language.

Skills in this chapter:

SKILL 2.1: CREATE TYPES

C# provides a range of "built-in" types. You have already encountered such types as `int` and `float` and have seen how to use these types to process data when your program runs. From a software point of view, there is no technical reason why every program in the world can't be written using the "built-in" types provided by C#. The ability to create our own data types in the programs that we write makes software development much easier and less prone to error.

This section covers how to:

- Create value types, including structs and enum
- Create reference types, generic types, constructors, static variables, methods, classes, and extension methods
- Create optional and named parameters
- Create indexed properties
- Create overloaded and overridden methods

Value and reference types

The first question to be addressed when creating a type in a program is whether the data you are representing should be stored using a *value* type or a *reference* type. A proper understanding of the difference between these types is crucial. The key to understanding the difference between value and reference types is appreciating what happens during the assignment process.

Consider the following two assignment statements. The first statement sets the variable `x` to equal the variable `y`. The second statement sets the `Data` property of the variable `x` to have the value `100`.

If the variables `x` and `y` are value types, the result of the first assignment is that the value of `y` is copied into the variable `x`. The result of the second assignment is that the Data property of the variable `x` is then set to the value `100`. The Data property in the variable y is not affected by the second assignment.

If the variables `x` and `y` are reference types, the result of the first assignment is that the variable `x` is made to refer to the same object as that referred to by variable `y`. The result of the second assignment is that the `Data` property of this single object is set to `100`.

The program in Listing 2-1 below shows these behaviors in action. The two assignment statements are performed on structure and class variables. One of these types is managed by reference, and the other is managed by value.

LISTING 2-1 Value and reference types

Click here to view code image

```
using System;

namespace LISTING_2_1_Value_and_reference_types
{
    class Program
    {
        struct StructStore
        {
            public int Data { get; set; }
        }

        class ClassStore
        {
            public int Data { get; set; }
        }

        static void Main(string[] args)
        {
            StructStore xs, ys;
            ys = new StructStore();
            ys.Data = 99;
            xs = ys;
            xs.Data = 100;
            Console.WriteLine("xStruct: {0}", xs.Da
            Console.WriteLine("yStruct: {0}", ys.Da

            ClassStore xc, yc;
            yc = new ClassStore();
            yc.Data = 99;
            xc = yc;
            xc.Data = 100;
            Console.WriteLine("xClass: {0}", xc.Dat
            Console.WriteLine("yClass: {0}", yc.Dat

            Console.ReadKey();
        }
    }
}
```

When you run this program, you will see the following output:

```
xStruct: 100
yStruct: 99
xClass: 100
yClass: 100
```

This shows that structure variables are managed by value (because changes to the variable `xStruct` do not affect the value of `yStruct`) and class variables are managed by reference (because `xClass` and `yClass` both refer to the same object, so changes via the `xClass` reference will affect the value referred to by `yClass`). You should study this code until you understand exactly why it behaves the way that it does, and why the action of the assignment operation is so different for each of the two types.

Value and reference types in .NET

You can learn more about the use of these types by considering some .NET types. The .NET libraries contain a range of types for programmers to use. Some are value types, and others are reference types.

A good example of a value type is the `DateTime` structure provided by the .NET library. This holds a value that represents a particular date and time. You can represent date values in the form of a collection of individual values, with one for the year, another for the month and so on, but assigning one date to another will be time-consuming because the program has to transfer each value in turn. Having a `DateTime` type that represents a date means that you can move a date value from one variable to another by performing a single assignment. When an assignment to a `DateTime` variable is performed, all of the values that represent the date are copied into the destination variable.

A good example of a reference type is the `Bitmap` class from the `System.Drawing` library in .NET. The `Bitmap` class is used to create objects that hold all of the pixels that make up an image on the screen. Images can contain millions of pixels. If a `Bitmap` is held as a value type,

simply makes the destination reference refer to the same object as the source reference.

Type design

You can build an understanding of value and reference types by considering some data items to store in your program and deciding whether or not they should be held in value or reference types. Perhaps you've been employed to write a "space shooter" game in which the player must shoot missiles at invading aliens.

The first item to consider is the location on the screen of an alien invader. Each alien is to be drawn at a particular X and Y coordinate on the screen. Should this coordinate be stored in a value type, or a reference type?

The answer is value type. This is because the program will want to treat the position of an alien as a single value, and you don't want to share coordinates between different aliens.

The next item to consider is the sound effect that is played when an alien is shot by the player. The sound effect will be held in the program as a large array of integers that represent the particular sound sample. This should be stored as a reference type. You may want to have several aliens share the same sound effect. This is very easy to achieve using references because each alien just has to contain a reference to the one sound effect object held in memory.

It turns out that value types are great for working with objects that you want to think of in terms of values, and reference types are great for working with objects that you want to manage by reference. In other words, the clue is in the name. In a C# program, value types are enumerated types and structures. Reference types are classes.

Immutability in types

When talking about value and reference types we need to mention *immutability*. An immutable type is one whose instances cannot be changed. The `DateTime` structure in the .NET libraries is an example of an immutable type. Once you have created a `DateTime` instance you cannot change any of the elements of that instance. You can read the year, month, and day elements of the instance, but you can't edit them. The only way to edit a `DateTime` value is by creating a new `DateTime` value that contains the updated values.

Listing 2-2 shows how a date can be advanced to the next day. The `DateTime` structure provides a method called `AddDays` to add days to a date, but this does not change the contents of a `DateTime` instance, and instead it returns a new date with the updated value.

LISTING 2-2 Immutable `DateTime`

Click here to view code image

```
// Create a DateTime for today
DateTime date = DateTime.Now;

// Move the date on to tomorrow
date = date.AddDays(1);
```

The modification of an immutable value will require the creation of a new object each time, which is less efficient than just changing a value inside an object. However, using an immutable type for a type removes any possibility of elements in a variable changing once it has been created.

Immutable types bring advantages when writing concurrent programs. In Skill 1-2, "Manage multithreading," you saw how race conditions can cause data corruption. An immutable object can never be corrupted because it cannot be changed. The string type in C# is immutable—it is not possible to change the content of a string once it has been created. We will discuss this in more detail when we consider strings later in this section.

Creating value types

There are two kinds of value types that can be created in a C# program. There are structures and enumerated types. Let's take a look at structures first.

Structures

We have already created a simple structure in the program in Listing 2-1. Structures can contain methods, data values, properties and can have constructors. When comparing a structure with a class (which can also contain methods, data values, properties, and constructors) there are some differences between the classes and structures:

- The constructor for a structure must initialize all the data members in the structure. Data members cannot be initialized in the structure.
- It is not possible for a structure to have a parameterless constructor. However, as you shall see, it is possible for a structure to be created by calling a parameterless constructor on the structure type, in which case all the elements of the structure are set to the default values for that type (numeric elements are set to zero and strings are set to null).
- It is not possible to create a structure by extending a parent structure object.
- Structure instances are generally created on the program stack unless they are used in closures (see "Closures" in Skill 1-4, "Create

the required items. See "Memory allocation" later in this section for more detail on this.

The program in Listing 2-3 creates a structure called `Alien`, which could be used to keep track of an `Alien` in the computer game mentioned earlier. An `Alien` has an X and Y position on the screen, which is set when the alien is created, along with a number of lives set to 3 by the `Alien` constructor. Note that the program in the listing creates two `Alien` instances. The first one, called `a`, is declared and then has its X, Y, and Lives data members set up. The second `Alien`, called `x`, is created at the position 100,100 on the screen. The sample program then creates an array of Aliens called `swarm`. Each element in the array is initialized with the default values for each data member type (in other words the values of X, Y and Lives will be 0).

LISTING 2-3 Creating a structure

Click here to view code image

```
using System;

namespace LISTING_2_3_Creating_a_structure
{
    struct Alien
    {
        public int X;
        public int Y;
        public int Lives;

        public Alien(int x, int y)
        {
            X = x;
            Y = y;
            Lives = 3;
        }

        public override string ToString()
        {
            return string.Format("X: {0} Y: {1} Liv
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Alien a;
            a.X = 50;
            a.Y = 50;
            a.Lives = 3;
            Console.WriteLine("a {0}", a.ToString()

            Alien x = new Alien(100, 100);
            Console.WriteLine("x {0}", x.ToString()

            Alien[] swarm = new Alien[100];
            Console.WriteLine("swarm [0] {0}", swar

            Console.ReadKey();
        }
    }
}
```

Note that the structure definition contains an override for the `ToString` method. This is perfectly acceptable; although a structure cannot be used in a class hierarchy, because it is possible to override methods from the parent type of `struct`. For a more detailed description of overriding, consult "Create overload and overridden methods" later in this section.

Enumerated types

Enumerated types are used in situations where the programmer wants to specify a range of values that a given type can have. For example, in the computer game you may want to represent three states of an alien: sleeping, attacking, or destroyed. You can use an integer variable to do this and adopt the convention that: the value 0 means sleeping, 1 means attacking, and 2 means destroyed. This works, but it raises the possibility that code in the program can set the state of an alien to the value of 4, which would be meaningless.

Listing 2-4 shows a program that creates an enum type to represent the state of an alien and then creates a variable of this type and sets its state to attacking.

LISTING 2-4 Creating an enum

Click here to view code image

```
using System;

namespace LISTING_2_4_Creating_an_enum
{
    enum AlienState
    {
        Sleeping,
        Attacking,
        Destroyed
```

```
    {
        static void Main(string[] args)
        {
            AlienState x = AlienState.Attacking;
            Console.WriteLine(x);

            Console.ReadKey();
        }
    }
}
```

This program prints the `ToString` result returned by the `AlienState` variable, which will output the string "Attacking" in this case.

Unless specified otherwise, an enumerated type is based on the `int` type and the enumerated values are numbered starting at 0. You can modify this by adding extra information to the declaration of the `enum`.

You would do this if you want to set particular values to be used in JSON and XML files when enumerated variables are stored. The code here creates an `AlienState` enum that is stored in a `byte` type, and has the given values for sleeping, attacking, and destroyed.

Click here to view code image

```
enum AlienState :
byte {
    Sleeping=1,
    Attacking=2,
    Destroyed=4
};
```

A program can use casting (see the Cast Types section) to obtain the numeric value that is held in an enum variable.

Creating reference types

The basis of C# reference types is the C# class. A class is declared in a very similar manner to a structure, but the way that classes are manipulated in a program is significantly different. You can declare the `Alien` type as a class by changing one word in the declaration of the Alien type. Listing 2-5 declares an `Alien` class and then creates a single `Alien` reference called `x`, which is made to refer to a new `Alien` object. This is followed by the creation of an array called `swarm`, which contains 100 `Alien` references. Each reference in the array is then made to refer to a new `Alien` object. The use of the `new` keyword to create a new object is said to create a new *instance* of a class.

LISTING 2-5 Creating a reference

Click here to view code image

```
using System;

namespace LISTING_2_5_Creating_a_reference
{
    class Alien
    {
        public int X;
        public int Y;
        public int Lives;

        public Alien(int x, int y)
        {
            X = x;
            Y = y;
            Lives = 3;
        }

        public override string ToString()
        {
            return string.Format("X: {0} Y: {1} Liv
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Alien x = new Alien(100, 100);
            Console.WriteLine("x {0}", x);

            Alien[] swarm = new Alien[100];

            for (int i = 0; i < swarm.Length; i++)
                swarm[i] = new Alien(0, 0);

            Console.WriteLine("swarm [0] {0}", swar

            Console.ReadKey();
        }
    }
}
```

anything. When we created the alien swarm we had to explicitly set each element in the array to refer to an `Alien` instance.

Memory allocation

Memory to be used to store variables of value type is allocated on the *stack*. The stack is an area of memory that is allocated and removed as programs enter and leave blocks. Any value type variables created during the execution of a block are stored on a local *stack frame* and then the entire frame is discarded when the block completes. This is an extremely efficient way to manage memory.

Memory to be used to store variables of reference type is allocated on a different structure, called the *heap*. The heap is managed for an entire application. The heap is required because, as references may be passed between method calls as parameters, it is not the case that objects managed by reference can be discarded when a method exits. Objects can only be removed from the heap when the garbage collection process determines that there are no references to them.

Generic types

Generic types are used extensively in C# collections, such as with the `List` and `Dictionary` classes. They allow you to create a `List` of any type of data, or a `Dictionary` of any type, indexed on any type.

Without generic types you either have to reduce the type safety in your programs by using collections that manage only objects, or you have to waste a lot of time creating different collection classes for each different type of data that you want to store.

The program in Listing 2-6 shows how generics can be used to create a stack type (called `MyStack`) that can be used to hold a stack of any type of object. The generic type to be used is specified in the declaration of `MyStack` (in the example below it is given the name `T`). The name `T` is then used within the `MyStack` declaration to represent the type that will be supplied when a variable of type `MyStack` is declared. The `Push` and `Pop` methods in the `MyStack` class also work with objects of type `T`. The program creates a `MyStack` that can hold strings.

LISTING 2-6 Using generic types

Click here to view code image

```
using System;

namespace LISTING_2_6_Using_generic_types
{
    class Program
    {
        class MyStack<T>
        {
            int stackTop = 0;
            T[] items = new T[100];

            public void Push(T item)
            {
                if (stackTop == items.Length)
                    throw new Exception("Stack full
                items[stackTop] = item;
                stackTop++;
            }

            public T Pop()
            {
                if (stackTop == 0)
                    throw new Exception("Stack empt
                stackTop--;
                return items[stackTop];
            }
        }

        static void Main(string[] args)
        {
            MyStack<string> nameStack = new MyStack
            nameStack.Push("Rob");
            nameStack.Push("Mary");
            Console.WriteLine(nameStack.Pop());
            Console.WriteLine(nameStack.Pop());
            Console.ReadKey();
        }
    }
}
```

Generic Constraints

The `MyStack` class in Listing 2-6 can hold any type of data. If you want to restrict it to only store reference types you can add a constraint on the possible types that T can represent. The `MyStack` declaration restricts the stack to holding reference types, so it isn't now possible to store integers (which are value types) in the stack.

Click here to view code image

```
class MyStack<T> where T:class
```

There are other constraints that you can use, as shown in Table 2-1.

TABLE 2-1 Generic constraints

Constraint	Behavior
where T : class	The type T must be a reference type.
where T : struct	The type T must be a value type.
where T : new()	The type T must have a public, parameterless, constructor. Specify this constraint last if you are specifying a list of constraints
where T : <base class>	The type T must be of type base class or derive from base class.
where T : <interface name>	The type T must be or implement the specified interface. You can specify multiple interfaces.
where T : unmanaged	The type T must not be a reference type or contain any members which are reference types.

Investigating how the generic collection classes work is a great way to build your understanding of how the generic features in C# are used.

Constructors

A constructor allows a programmer to control the process by which objects are created. Constructors can be used with value types (structures) and reference types (classes). A constructor has the same name as the object it is part of but does not have a return type.

Constructors can perform validation of their parameters to ensure that any objects that are created contain valid information. If the validation fails, the constructor must throw an exception to prevent the creation of an invalid object. The code in Listing 2-7 shows a constructor for the `Alien` class that throws an exception if an attempt is made to create an Alien with negative coordinate values.

LISTING 2-7 Constructors

Click here to view code image

```
using System;

namespace LISTING_2_7_Constructors
{
    class Alien
    {
        public int X;
        public int Y;
        public int Lives;

        public Alien(int x, int y)
        {
            if (x < 0 || y < 0)
                throw new ArgumentOutOfRangeExcepti

            X = x;
            Y = y;
            Lives = 3;
        }

        public override string ToString()
        {
            return string.Format("X: {0} Y: {1} Liv

        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Alien x = new Alien(100, 100);
            Console.WriteLine("x {0}", x);
            Console.ReadKey();
        }
    }
}
```

Constructors can be given access to modifiers (see "Enforce encapsulation" later in this section). If an object only has a private constructor it cannot be instantiated unless the object contains a public *factory method* that can be called to create instances of the class.

Constructors can be overloaded, so an object can contain multiple versions of a constructor with different signatures. Listing 2-8 shows an `Alien` class that allows an alien to be constructed with a particular number of lives, or a value of three lives, depending on which constructor

Click here to view code image

```
class Alien
{
    public int X;
    public int Y;
    public int Lives;

    public Alien(int x, int y, int lives)
    {
        if (x < 0 || y < 0)
            throw new ArgumentOutOfRangeException ('

        X = x;
        Y = y;
        Lives = lives;
    }

    public Alien(int x, int y)
    {
        X = x;
        Y = y;
        Lives = 3;
    }
}
```

A program can avoid code repetition by making one constructor call another constructor by use of the keyword `this`. The `this` keyword is used in the constructor method signature as shown in Listing 2-9. It forms a call of another constructor in the object. In the program below the parameters to the call of one constructor are passed into a call of another, along with an additional lives value. Note that this means the actual body of the constructor is empty, because all of the work is performed by the call to the other constructor. Another way to provide default values to a constructor is to make use of optional parameters. These are described in "Optional and named parameters" in the section below.

LISTING 2-9 Calling constructors

Click here to view code image

```
class Alien
{
    public int X;
    public int Y;
    public int Lives;

    public Alien(int x, int y, int lives)
    {
        if (x < 0 || y < 0)
            throw new ArgumentOutOfRangeException ('

        X = x;
        Y = y;
        Lives = lives;
    }

    public Alien(int x, int y) : this(x, y, 3)
    {
    }

    public override string ToString()
    {
        return string.Format("X: {0} Y: {1} Lives:
    }
}
```

When creating objects that are part of a class hierarchy, a programmer must ensure that information required by the constructor of a parent object is passed into a parent constructor. This will be discussed in more detail in Skill 2.4, Create and implement a class hierarchy."

Static constructors

A class can contain a *static constructor* method. This is called once before the creation of the very first instance of the class. The `Alien` class in Listing 2-10 contains a static constructor that prints a message when it is called. When the program runs, the message is printed once, before the first alien is created. The static constructor is not called when the second alien is created.

LISTING 2-10 Static constructors

Click here to view code image

```
class Alien
{
    // Alien code here
    static Alien()
    {
        Console.WriteLine("Static Alien constructor
    }
}
```

static members of the class, as described in the next section.

Static variables

A static variable is a member of a type, but it is not created for each instance of a type. A variable in a class is made static by using the keyword `static` in the declaration of that variable.

As an example of a situation where static is useful, you may decide that you want to set a maximum for the number of lives that an `Alien` is allowed to have in our video game. This is a value that should be stored once for all aliens. A static variable is a great place to store such a value, since it will be created once for all class instances. Listing 2-11 shows how a static variable (`Max_Lives`) can be added to the `Alien` class and used in the `Alien` constructor to reject any attempts to create aliens with too many lives.

LISTING 2-11 Static variables

Click here to view code image

```
class Alien
{
    public static int Max_Lives = 99;

    public int X;
    public int Y;
    public int Lives;

    public Alien(int x, int y, int lives)
    {
        if (x < 0 || y < 0)
            throw new Exception("Invalid position")

        if(lives > Max_Lives)
            throw new Exception("Invalid lives");

        X = x;
        Y = y;
        Lives = lives;
    }
}
```

Code outside of the `Alien` class must refer to the `Max_Lives` static variable via the class name, rather than the name of any particular instance of the class. The statement next changes the value of Max_Lives to 150:

Click here to view code image

```
Alien.Max_Lives = 150;
```

Making a variable `static` does not stop it from being changed when the program runs (to achieve this use the `const` keyword or make the variable `readonly`). Rather, the word static in this context means that the variable is "always present." A program can use a static variable from a type without needing to have created any instances of that type. Types can also contain static methods. These can be called without the need for an instance of the object containing the method. Libraries of useful functions are often provided as static members of a library class.

Static variables are very useful for validation values for a type, such as the maximum number of lives, and also for default values. They can be made private to a class so that their values can be managed by the class.

Methods

You have seen methods used in several of the types that you have worked with in this section. A method is a member of a class. It has a *signature* and a *body*. The signature defines the type and number of parameters that the method will accept. The body is a block of code that is performed when the method is called. If the method has a type other than `void`, all code paths through the body of the code must end with a `return` statement that returns a value of the type of the method.

Listing 2-12 shows a method called `RemoveLives`, which is called to remove lives from an `Alien`. The method is provided with a parameter that gives the number of lives to remove. If the number of lives that are left is less than zero, the lives value is set to zero and the `Alien` is moved off the display screen so that it is not visible any more. The `RemoveLives` method is of type Boolean and returns `true` if the alien is still alive and `false` if it is not.

LISTING 2-12 Simple method

Click here to view code image

```
class Alien
{
    public int X;
    public int Y;
    public int Lives;

    public bool RemoveLives(int livesToRemove)
    {
        Lives = Lives - livesToRemove;
```

```
            X = -1000;
            Y = -1000;
            return false;
        }
        else
        {
            return true;
        }
    }
}
```

This method can be called on an `Alien` instance to remove lives and determine if the given alien is still alive.

Click here to view code image

```
Alien x = new Alien(100, 100);
Console.WriteLine("x {0}", x);
if (x.RemoveLives(2))
{
    Console.WriteLine("Still alive");
}
else
{
    Console.WriteLine("Alien destroyed");
}

Console.WriteLine("x {0}", x);
```

You might like to consider a design approach where an alien fires an event when it is destroyed, using delegates as described in Skill 1-4, "Create and implement events and callbacks."

The name of a method is best expressed in a "verb-noun" manner, with an action followed by the thing that the action is acting on. Names such as "DisplayMenu," "SaveCustomer," and "DeleteFile" are very descriptive of what the method does. When talking about the method signature and the code body of a method we will talk in terms of the *parameters* used in the method. In the case of the call of a method we will talk in terms of the *arguments* supplied to the call. In other words, in the example shown in Listing 2-12 the parameter to the method is called `livesToRemove` and the argument to the method call is the value 2.

Classes

You can think of a class as providing the template or plans that are required to create an instance of that class. Note that declaring a class does not create any instances of that class. An instance of a class is created when the `new` keyword is used. When the `new` keyword is used to create an instance of a class the following sequence is performed:

1. The program code that implements the class is loaded into memory, if it is not already present.
2. If this is the first time that the class has been referenced, any static members of the class are initialized and the static constructor is called.
3. The constructor in the class is called.

A class can contain members that are methods, data variables, or properties. A class method allows a class to provide behaviors that can be used by code running in other classes. Data variables allow a class to maintain state and manage the storage of information and properties provide a means for managing access to data within a class.

Extension methods

A class can provide methods for other classes to call. In Skill 2-4, "Create and implement a class hierarchy," you will see how a class can be extended by a child class that can add members to the base class, including new methods that can implement additional behaviors. However, extension methods provide a way in which behaviors can be added to a class without needing to extend the class itself. You can think of the extension methods as being "bolted on" to an existing class.

Listing 2-13 shows how to create an extension method for the `String` class. The first parameter to the method specifies the type that the extension method should be added to, by using the keyword `this` followed by the name of the type. The extension method created, called `NoLines`, counts the number of lines in a string, and then returns this result as an integer. The method works by splitting the string on the linefeed character and then counting the number of elements in the resulting array. The extension method is declared in a static class (in this case called `MyExtensions`).

LISTING 2-13 Extension method

Click here to view code image

```
using System;

namespace ExtensionMethods
{
    public static class MyExtensions
    {
        public static int LineCount(this String str
        {
            return str.Split(new char[] { '\n' },
                             StringSplitOptions.Rem
```

```
    }
}
```

Once the extension method has been created it can be used from the namespace in which the class containing the method is declared. When the program calls an extension method the compiler searches the included namespaces for a matching method for that type, and then generates a call of that method. When the program below runs, it prints out the number of lines in the string text.

Click here to view code image

```
namespace LISTING_2_13_Extension_method
{
    using ExtensionMethods;

    class Program
    {
        static void Main(string[] args)
        {
            string text = @"A rocket explorer calle
Once travelled much faster than light,
He set out one day,
In a relative way,
And returned on the previous night";

            Console.WriteLine(text.LineCount());

            Console.ReadKey();
        }
    }
}
```

Note that extension methods are never part of the object they are attached to, since they don't have any access to private members of the method class. An extension method can never be used to replace an existing method in a class. In other words, if the String type already contained a method called `LineCount`, the extension method created in Listing 2-13 is not called. Instead, the `LineCount` method inside `String` is used.

Extension methods allow you to add behaviors to existing classes and use them as if they were part of that class. They are very powerful. LINQ query operations are added to types in C# programs by the use of extension methods.

Optional and named parameters

When you create a method with parameters, the signature of the method gives the name and type of each parameter in turn (see Listing 2-14)

LISTING 2-14 Named parameters

Click here to view code image

```
static int ReadValue (
    int low,    // lowest allowed value
    int high,    // highest allowed value
    string prompt // prompt for the user
    )
{
    // method body...
}
```

The `ReadValue method` has been defined as having three parameters. A call of the method must have three argument values: a prompt, a low value, and a high value. This means that the following call of `readValue` is rejected by the compiler:

Click here to view code image

```
x = ReadValue("Enter your age: ", 1, 100);
```

This is because the prompt string is defined as the last parameter to the method call, not the first. If you want to make method calls without worrying about the order of the arguments, you can name each one when you call the method:

Click here to view code image

```
x = ReadValue(low:1, high:100, prompt: "Enter your
```

Now the compiler is using the name of each argument, rather than its position in the list. Another programmer reading your code can now see the meaning of each argument value. Using this format also removes the possibility of any confusion of the ordering of the values in the method call. You should use named parameters whenever you call a method that has more than one parameter.

Sometimes the value of an argument might have a sensible default value. For example, if you only want the `readValue` to fetch a value from the user and not display a prompt, you can do this by providing an empty string:

```
x = readValue(low:25, high:100, prompt: "");
```

This, however, is a bit messy. Instead, you can change the definition of the method to give a default value for the prompt parameter as shown in Listing -2-15.

LISTING 2-15 Optional parameters

Click here to view code image

```
static int readValue (
    double low,           // lowest allowed value
    double high,          // highest allowed value
    string prompt = "",   // optional prompt for th
    )
{
    // method body...
}
```

You can now call the method and leave the prompt out:

```
x = readValue(25, 100);
```

When the method runs, the prompt will be set to an empty string if the user doesn't provide a value. Optional parameters must be provided after all of the required ones.

Indexed properties

A program can access a particular array element by using an index value that identifies the element. The code here shows how this works:

Click here to view code image

```
int [] array = new int[20];
array[0] = 99;
```

A class can use the same indexing mechanism to provide indexed property values. The code in Listing 2-16 shows how this works. The `IntArrayWrapper` class is a wrapper around an integer array. The indexer property accepts an integer value (called `i` in Listing 2-16), which is used to index the array that stores the value.

LISTING 2-16 Indexed properties

Click here to view code image

```
using System;

namespace LISTING_2_16_Indexed_properties
{
    class IntArrayWrapper
    {
        // Create an array to store the values
        private int[] array = new int[100];

        // Declare an indexer property
        public int this[int i]
        {
            get { return array[i]; }
            set { array[i] = value; }
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            IntArrayWrapper x = new IntArrayWrapper
            x[0] = 99;
            Console.WriteLine(x[0]);
            Console.ReadKey();
        }
    }
}
```

Note that there is nothing stopping the use of other types in indexed properties. This is how the `Dictionary` collection is used to index on a particular type of key value. Listing 2-17 shows an indexer property that is extended to allow all of the elements in the integer array to be accessed by name. The first couple of values only have been implemented for reasons of space.

LISTING 2-17 Indexing on strings

Click here to view code image

```
class NamedIntArray
{
    // Create an array to store the values
    private int[] array = new int[100];
```

```
            get
            {
                switch (name)
                {
                    case "zero":
                        return array[0];
                    case "one":
                        return array[1];
                    default:
                        return -1;
                }
            }
            set
            {
                switch (name)
                {
                    case "zero":
                        array[0] = value;
                        break;
                    case "one":
                        array[1] = value;
                        break;
                }
            }
        }
    }
```

A program can now access elements in the array by specifying a text indexer for the location. This program stores the value 99 in location "zero:"

Click here to view code image

```
NamedIntArray x = new NamedIntArray();
x["zero"] = 99;
Console.WriteLine(x["zero"]);
```

Create overload and overridden methods

It's very important to understand the difference between *overloading* and *overriding* when applied to methods in a class. Overloading means, "providing a method with the same name, but a different signature in a given type." It is useful when you want to provide several ways of performing a particular behavior, depending on the circumstances in which the behavior is being used. We've seen overloading in the context of constructor methods earlier, in order to construct an `Alien` with a specified number of lives or construct an `Alien` with a default number of lives.

The `DateTime` structure provided by .NET has a large number of overloaded constructors, because there are many different ways a programmer might want to initialize a `DateTime` value. You might have the number of ticks since January 1, 0001 at 00:00:00.000 in the Gregorian calendar. Alternatively, and perhaps more likely, you might have year, month, and day values. Listing 2-18 shows how these values are used to create `DateTime` values. Both of the `DateTime` values are set to the same date and time by this code.

LISTING 2-18 Overloaded `DateTime` constructor

Click here to view code image

```
using System;

namespace LISTING_2_18_Overloaded_DateTime_construc
{
    class Program
    {
        static void Main(string[] args)
        {
            DateTime d0 = new DateTime(ticks:636679
            DateTime d1 = new DateTime(year:2018, m
            Console.WriteLine(d0);
            Console.WriteLine(d1);
            Console.ReadKey();
        }
    }
}
```

The overriding of methods takes place when class hierarchies are used. In a class hierarchy a child class is derived from a parent or base class. A method in a base class is *overridden* by a method in a child when the child class contains a method with exactly the same name and signature as a method in the parent class. Only methods that have been marked as *virtual* in the parent class can be overridden.

The key to understanding overriding is to discover why you need to use it. The underlying principle of a class hierarchy is that classes at the top of the hierarchy are more abstract, and classes toward the bottom of the hierarchy are more specific. So, a base class might be called `Document` and child class might be called `Invoice`.

The `Document` class will hold all of the behaviors that are common to all documents, for example, all documents must provide a method that gets the date when the document was created. However, the print behavior of an Invoice will have to be different from the print behavior of the more

Listing 2-19 shows how method overriding works. The `GetDate` method is declared in `Document`. The method `DoPrint` is declared in `Document` and overridden in `Invoice`. The program creates an instance of `Invoice` and then calls the `GetDate` and `DoPrint` methods on the instance. When `GetDate` is called, the `GetDate` from `Document` is called. When `DoPrint` is called on the `Invoice`, the `DoPrint` from `Invoice` is called, because this overrides the `DoPrint` in the parent class.

LISTING 2-19 Method overriding

Click here to view code image

```
using System;

namespace LISTING_2_19_Method_overriding
{
    class Document
    {
        // All documents have the same GetDate beha
        // this method will not be overridden
        public void GetDate()
        {
            Console.WriteLine("Hello from GetDate i
        }

        // A document may have its own DoPrint beha
        // this method is virtual so it can be over
        public virtual void DoPrint()
        {
            Console.WriteLine("Hello from DoPrint i
        }
    }

    // The Invoice class derives from the Document
    class Invoice:Document
    {
        // Override the DoPrint method in the base
        // to provide custom printing behaviour for
        public override void DoPrint()
        {
            Console.WriteLine("Hello from DoPrint i
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            // Create an Invoice
            Invoice c = new Invoice();
            // This will run the SetDate method fro
            c.GetDate();
            // This will run the DoPrint method fro
            c.DoPrint();

            Console.ReadKey();
        }
    }
}
```

The output from the program is shown here:

Click here to view code image

```
Hello from GetDate in Document
Hello from DoPrint in Invoice
```

If you create a `PrePaidInvoice` to represent pre-paid invoices, you can extend the `Invoice` class and provide an override of the `DoPrint` method in the `PrePaidInvoice` class. The `DoPrint` method in `PrePaidInvoice` can use the `DoPrint` method in the parent class by using the `base` keyword, as shown in Listing 2-20 below.

LISTING 2-20 Using base

Click here to view code image

```
class PrePaidInvoice: Invoice
{
    public override void DoPrint()
    {
        base.DoPrint();
        Console.WriteLine("Hello from DoPrint in Pr
    }
}
```

When the `DoPrint` method is called on an instance of the `PrePaidInvoice` class, it first makes a call of the `DoPrint` method in the parent object. The listing below shows how these calls are made.

Click here to view code image

```
p.GetDate();
p.DoPrint();
```

The output from the calls to the methods is shown here:

Click here to view code image

```
Hello from GetDate in Document
Hello from DoPrint in Invoice
Hello from DoPrint in PrePaidInvoice
```

The use of method overriding allows the behavior of different items to be customized for each particular item, and you can also re-use code in parent objects by using the base keyword. You can find more discussion of class hierarchies in Skill 2.4, "Create and Implement a Class Hierarchy."

SKILL 2.2: CONSUME TYPES

Everything in C# is an object of a particular type and the compiler ensures that interactions between types are always meaningful. In the previous section you discovered value and reference types and considered how to create types that hold data and provide behaviors.

In this section we're going to build on this knowledge and learn how to use differently typed objects in our programs. We're going to find out about the process performed in C# when a value type is converted into a reference type, and how to use casting to inform the compiler of our intentions when working with types. Finally, we are going to discover how the programs from the statically typed world of C# can be made to interact with environments that have a dynamically typed way of working; notably Component Object Model (COM) interfaces.

This section covers how to:

- Box or unbox to convert between value types
- Cast types
- Convert types;
- Handle dynamic types
- Ensure interoperability with unmanaged code that accesses COM APIs

Boxing and unboxing

From Skill 2.1, "Creating types," we know that C# programs can use value types and reference types. We know that value types are managed directly in terms of their value, whereas reference types are managed in terms of a reference that refers to an object that holds the data. The built-in types `int`, `float`, and `double` are value types, as are structures that we create. Classes are used to define reference types.

From a computational point of view, value types such as `int` and `float` have the advantage that the computer processor can manipulate value types directly. Adding two `int` values together can be achieved by fetching the values into the processor, performing the addition operation, and then storing the result.

It can be useful to treat value types as reference types, and the C# runtime system provides a mechanism called *boxing* that will perform this conversion when required. Listing 2-21 shows boxing and unboxing being performed in a program. The first statement takes the value 99 and casts this into an object called `o` (you can find out more about casting in the next section). The second statement takes the object `o` and casts it back into an integer. The process of converting from a reference type (reference `o`) into a value type (the integer `oVal`) is called *unboxing*.

LISTING 2-21 Boxing and unboxing

Click here to view code image

```
namespace LISTING_2_21_Boxing_and_unboxing
{
    class Program
    {
        static void Main(string[] args)
        {
            // the value 99 is boxed into an object
            object o = 99;

            // the boxed object is unboxed back int
            int oVal = (int)o;
            Console.WriteLine(oVal);

            Console.ReadKey();
        }
    }
}
```

Boxing and unboxing at this point might seem a bit confusing and magical. The program code in Listing 2-21, which works correctly and prints the value 99, implies that an object is magically able to store an

can assign the object `o` to a floating-point number, or a double precision number.

You can get a better understanding of just what is happening by inspecting the actual code that is produced by the C# compiler when the program in Listing 2-21 is compiled. Listing 2-22 shows the output from a program called `ildasm`, which is supplied as part of the .NET Software Development Kit. The `ildasm` program (the name is short for "Intermediate Language Disassembler") will take a compiler output file and show the actual instructions that will be obeyed when the program runs.

A .NET program processes data by moving values onto a stack (called the evaluation stack), performing actions on them, and then storing the results. The instruction at `IL_0001` in Listing 2-22 takes the value `99` and places it on the evaluation stack. The instruction at `IL_0003` is the box instruction that creates an object of type `Int32` on the heap, sets the value of this object to `99` and then places a reference to this object on the evaluation stack. This reference is then stored in storage location 0, which is where the variable o is stored. The instruction at `IL_000a` performs the reverse of this action, unboxing a reference to an `Int32` object and placing the value that results from the unboxing on the evaluation stack.

LISTING 2-22 Compiled and unboxing

Click here to view code image

```
method private hidebysig static void  Main(string[]
{
  .entrypoint
  // Code size       30 (0x1e)
  .maxstack  1
  .locals init ([0] object o,
           [1] int32 oVal)
  IL_0000:  nop
  IL_0001:  ldc.i4.s   99
  IL_0003:  box        [mscorlib]System.Int32
  IL_0008:  stloc.0
  IL_0009:  ldloc.0
  IL_000a:  unbox.any  [mscorlib]System.Int32
  IL_000f:  stloc.1
  IL_0010:  ldloc.1
  IL_0011:  call       void [mscorlib]System.Consol
  IL_0016:  nop
  IL_0017:  call       valuetype [mscorlib]System.C
  IL_001c:  pop
  IL_001d:  ret
} // end of method Program::Main
```

It's worth spending some time examining this code to see how .NET programs are executed. Note that the `box` instruction (convert a value type to a reference type) and `unbox` instruction (convert a reference type to a value type) are always given a destination (box) or source (unbox) type to use. Each built-in C# value type (`int`, `float` etc) has a matching C# type called its *interface type* to which it is converted when boxing is performed. The interface type for `int` is `int32`.

The good news for programmers is that boxing and unboxing process happens automatically when variables are converted between types. The bad news is that if a program spends a lot of time boxing and unboxing values it slows the program down. The need to box and unbox values in a solution is a symptom of poor design, you should be clear in your design which data items are value types and which references, and work with them correctly.

Cast types

The C# language has been designed to reduce the ways in which a programmer can make mistakes, which will cause a program to produce invalid results. One aspect of this design is the way that a C# program will not allow a programmer to perform a conversion between types that result in the loss of data.

The two statements below show a situation in which data is lost when an assignment is made. It is an example of *narrowing*, when a value is transferred into a type which offers a narrower range of values. The integer type does not handle the fractional part of a value, and so the .9 part of the value x is discarded when the assignment takes place. These statements will not compile successfully.

Click here to view code image

```
float x = 9.9f;
int i = x;
```

C# is perfectly capable of performing the conversion from floating point to integer, but it requires confirmation from the programmer that it is meaningful for this conversion to be performed. This is called *explicit conversion*, in that the programmer is making an explicit request that the conversion be performed. In C# the explicit conversion is requested by the use of a *cast*, which identifies the desired type of value being assigned. The type is given, enclosed in brackets, before the value to be converted. The statements here show how a cast is added to allow a floating-point value to be assigned to an integer.

Click here to view code image

A conversion that performs *widening*, in which the destination type has a wider range of values than the source, does not require a cast, because there is no prospect of data loss.

Note that casting cannot be used to convert between different types, for example with an integer and string. In other words, the following statement will fail to compile:

Click here to view code image

```
int i = (int) "99";
```

Casting is also used when converting references to objects that may be part of class hierarchies or expose interfaces. You can find out more about this form of casting in Skill 2.4, "Create and implement a class hierarchy."

Convert types

The .NET runtime provides a set of conversion methods that are used to perform casting. You can also write your own type conversion operators for your data classes so that programs can perform implicit and explicit conversions between types. Listing 2-23 shows how to do this.

The type `Miles` contains a double precision distance value in the property `Distance`. The `Miles` class also contains an implicit operator called `Kilometers`, which returns a value of type `Kilometers`, representing the same distance. A program can then assign a variable of type `Miles` value into a variable of type `Kilometer`. During the assignment the implicit conversion operator is called automatically. The type `Miles` also contains an explicit conversion that returns the distance value as an integer. This conversion is explicit because it is a narrowing operation that will result in a loss of data, as the fractional part of the double precision distance value is truncated. To use an explicit conversion the programmer must use a cast, as shown in Listing 2-23.

LISTING 2-23 Type conversion

Click here to view code image

```
using System;

namespace LISTING_2_23_Type_conversion
{
    class Miles
    {
        public double Distance { get; }

        // Conversion operator for implicit convers
        public static implicit operator Kilometers(
        {
            Console.WriteLine("Implicit conversion
            return new Kilometers( t.Distance * 1.6
        }

        public static explicit operator int(Miles t
        {
            Console.WriteLine("Explicit conversion
            return (int)(t.Distance + 0.5);
        }

        public Miles(double miles)
        {
            Distance = miles;
        }
    }

    class Kilometers
    {
        public double Distance { get; }

        public Kilometers(double kilometers)
        {
            Distance = kilometers;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Miles m = new Miles(100);

            Kilometers k = m; // implicity convert
            Console.WriteLine("Kilometers: {0}", k.

            int intMiles = (int)m;  // explicitly c
            Console.WriteLine("Int miles: {0}", int
            Console.ReadKey();
        }
    }
}
```

When you run the program, you will see that it writes messages indicating that the implicit and explicit conversions are being performed.

Click here to view code image

```
Implicit conversion from miles to kilometers
Kilometers: 160
Explicit conversion from miles to int
Int miles: 100
```

Convert types with `System.Convert`

The `System.Convert` class provides a set of static methods that can be used to perform type conversion between .NET types. As an example, the code next converts a string into an integer:

Click here to view code image

```
int myAge = Convert.ToInt32("21");
```

The convert method will throw an exception if the string provided cannot be converted into an integer.

Handle dynamic types

C# is a strongly typed language. This means that when the program is compiled the compiler ensures that all actions that are performed are valid in the context of the types that have been defined in the program. As an example, if a class does not contain a method with a particular name, the C# compiler will refuse to generate a call to that method. As a way of making sure that C# programs are valid at the time that they are executed, strong typing works very well. Such a strong typing regime, however, can cause problems when a C# program is required to interact with systems that do not have their origins in C# code. Such situations arise when using Common Object Model (COM) interop, the Document Object Model (DOM), working with objects generated by C# reflection, or when interworking with dynamic languages such as JavaScript.

In these situations, you need a way to force the compiler to interact with objects for which the strong typing information that is generated from compiled C# is not available. The keyword `dynamic` is used to identify items for which the C# compiler should suspend static type checking. The compiler will then generate code that works with the items as described, without doing static checking to make sure that they are valid. Note that this doesn't mean that a program using dynamic objects will always work; if the description is incorrect the program will fail at run time.

Listing 2-24 shows how the dynamic keyword is used in this situation. The class `MessageDisplay` contains a single method, called `DisplayMessage`. The variable `m` is set to refer to an instance of this class, and the program calls the `DisplayMessage` method on this reference. The compiler is very happy with this code, as it can see that the `MessageDisplay` class contains the required method. The variable `d` is declared as `dynamic` and set to refer to a `MessageDisplay` instance. The program then contains a call of a method called `Banana` on the variable `d`. Normally this would not compile, because the compiler can see that this method is not present in the class. Because the variable `d` has been declared as `dynamic`, however, the program will compile with no errors.

LISTING 2-24 Bad dynamic code

Click here to view code image

```
using System;

namespace LISTING_2_24_Bad_dynamic_code
{
    class MessageDisplay
    {
        public void DisplayMessage(string message)
        {
            Console.WriteLine(message);
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            MessageDisplay m = new MessageDisplay()
            m.DisplayMessage("Hello world");

            dynamic d = new MessageDisplay();
            d.Banana("hello world");
        }
    }
}
```

This program will compile, but when the program is executed an exception will be generated when the `Banana` method is called. Figure 2-1 shows the result of running the program in the Visual Studio 2017 debugger.

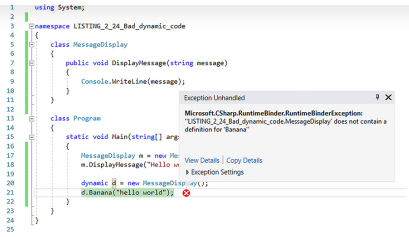

FIGURE 2-1 The Bad Dynamic code application

This aspect of the dynamic keyword makes it possible to interact with objects that have behaviors, but not the C# type information that the C# compiler would normally use to ensure that any interaction is valid. There is, however, more to the dynamic type than this.

A variable declared as dynamic is allocated a type that is inferred from the context in which it is used. This is a very similar behavior to that of variables in languages such as Python or JavaScript. The code in Listing 2-25 shows how this works. The variable `d` is declared as `dynamic` and used first as an integer and secondly as a string. This program will compile and execute with no errors, printing out the results that you would expect. The type of the variable `d` will change according to what is stored in it, and the addition operator works as expected, adding an integer to an integer and a string to a string. If the program behaves incorrectly (for example by trying to add "Rob" to the integer incarnation of `d`) an exception is thrown when the program runs.

LISTING 2-25 Using dynamic variables

Click here to view code image

```
dynamic d = 99;
d = d + 1;
Console.WriteLine(d);

d = "Hello";
d = d + " Rob";
Console.WriteLine(d);
```

Note, that just because you can do this doesn't mean that you should. The flexibility of the dynamic type was not created so that C# programmers can stop worrying about having to decide on the type of variables that they use in their programs. Instead, the flexibility was added to make it easy to interact with other languages and libraries written using the Component Object Model (COM).

Use ExpandoObject

The `ExpandoObject` class allows a program to dynamically add properties to an object. The code next, which is also in Listing 2-25, shows how this is done. The `dynamic` variable `person` is assigned to a new `ExpandoObject` instance. The program then adds `Name` and `Age` properties to the `person` and then prints out these values.

Click here to view code image

```
dynamic person = new ExpandoObject();

person.Name = "Rob Miles";
person.Age = 21;

Console.WriteLine("Name: {0} Age: {1}", person.Name,
```

A program can add `ExpandoObject` properties to an `ExpandoObject` to create nested data structures. An `ExpandoObject` can also be queried using LINQ and can exposes the `IDictionary` interface to allow its contents to be queried and items to be removed. `ExpandoObject` is especially useful when creating data structures from markup languages, for example when reading a JSON or XML document.

Interoperability with unmanaged code that accesses COM APIs

The Component Object Model (COM) is a mechanism that allows software components to interact. The model describes how to express an interface to which other objects can connect. COM is interesting to programmers because a great many resources you would like to use are exposed via COM interfaces.

The code inside a COM object runs as *unmanaged code,* having direct access to the underlying system. While it is possible to run .NET applications in an *unmanaged* mode, .NET applications usually run inside a *managed* environment, limiting the level of access that the applications have to the underlying system.

When a .NET application wants to interact with a COM object it has to perform the following:

1. Convert any parameters for the COM object into an appropriate format
2. Switch to unmanaged execution for the COM behavior
3. Invoke the COM behavior

5. Convert any results of the COM request into the correct types of .NET objects

This is performed by a component called the Primary Interop Assembly (PIA) that is supplied along with the COM object. The results returned by the PIA can be managed as dynamic objects, so that the type of the values can be inferred rather than having to be specified directly. As long as your program uses the returned values in the correct way, the program will work correctly. You add a Primary Interop Assembly to an application as you would any other assembly. Figure 2.2 shows the interop assembly added to a Visual Studio solution.

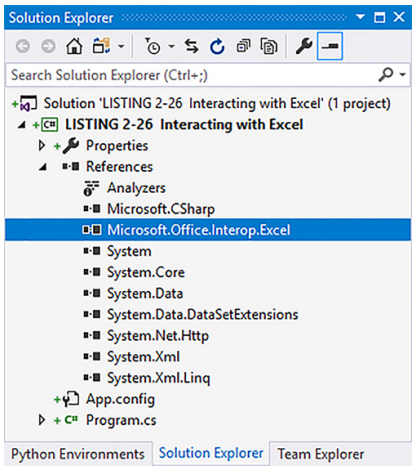

FIGURE 2-2 Adding a COM interop

The C# code uses dynamic types to make the interaction with the Office application very easy. There is no need to cast the various elements that the program is interacting with, as they are exposed by the interop as dynamic types, so conversion is performed automatically based on the inferred type of an assignment destination. Listing 2-26 shows code that opens Excel, creates a new spreadsheet, and adds text into two cells. The program is self-explanatory and very simple to use. You can see the use of the interop object when the `excelApp` object is created.

LISTING 2-26 Interacting with Excel

Click here to view code image

```
namespace LISTING_2_26_Interacting_with_Excel
{
    class Program
    {
        static void Main(string[] args)
        {
            // Create the interop
            var excelApp = new Microsoft.Office.Int

            // make the app visible
            excelApp.Visible = true;

            // Add a new workbook
            excelApp.Workbooks.Add();

            // Obtain the active sheet from the app
            // There is no need to cast this dynami
            Microsoft.Office.Interop.Excel.Workshee

            // Write into two cells
            workSheet.Cells[1, "A"] = "Hello";
            workSheet.Cells[1, "B"] = "from C#";
        }
    }
}
```

You might be wondering about the use of dynamic in the code in Listing 2-26. Figure 2-3 shows the Intellisense for the `Activesheet` component of the application. This is shown as a dynamic type, which is then assigned to a variable of Worksheet type without the need for any casting.

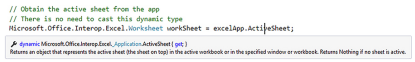

FIGURE 2-3 Using dynamic types

When this program runs it starts Excel running and then sets the values of the top two cells in the spreadsheet as shown in Figure 2-4.

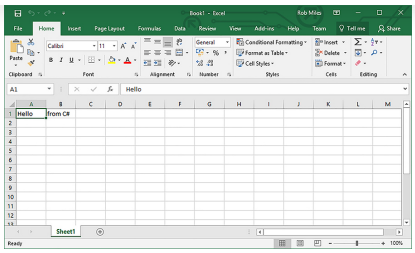

FIGURE 2-4 Interacting with Excel

Embedding Type Information from assemblies

You can create applications that interact with different versions of Microsoft Office by embedding the Primary Interop Assembly in the application. This is achieved by setting the Embed Interop Types option of the assembly reference to True. This removes the need for any interop assemblies on the machine running the application.

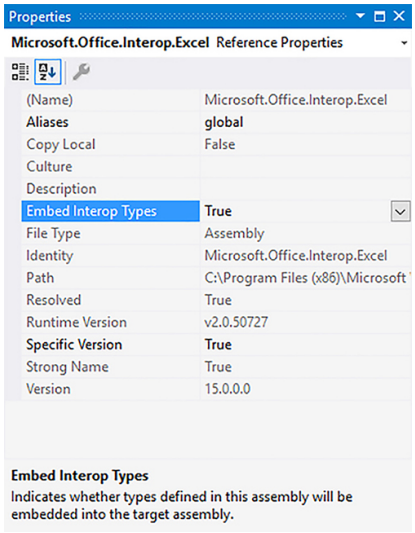

FIGURE 2-5 Embed interop assemblies

SKILL 2.3: ENFORCE ENCAPSULATION

You are now building up an understanding of how to create and use objects in applications. An object can hold data values by the use of member variables and provide services for other objects by the use of member methods. You can use constructors to set up the contents of an object and ensure that the object has valid contents from the point of creation. You should also understand the difference between objects managed by value and reference, and how dynamic types allow C# programs to interact with external languages and libraries.

One definition of the word encapsulate is to "enclose in a capsule." In this skill you are going to build on this knowledge to discover how to use encapsulation to improve the safety of programs that you write. You will discover how to enclose your software objects in capsules.

When discussing safety, a car designer will talk about *active* and *passive* safety systems. An active safety system is one that stops the car from crashing. Active safety systems include things like stability controls and anti-lock brakes. Passive safety is provided by systems in the car that are activated in the event of a crash. Passive safety systems include things like air-bags and crumple zones designed to absorb impacts.

In Skill 1-5, "Implement Exception Handling," you saw how exception handlers can be added to a program that will allow the program to respond in a managed way when errors occur as the program is running. You can think of this as "passive" program safety, in that exceptions allow you to create handlers that take control in the event of errors occurring.

You can think of encapsulation as a technique that provides *active* program safety. Use encapsulation to reduce the possibility of program errors occurring. Encapsulation lets you hide the elements that actually implement the behaviors of an object, so that the object only exposes the behaviors that it provides by a well-defined interface.

Encapsulation provides protection against accidental damage, where a programmer directly changes an internal element of an object without having a proper understanding of the effect of the change. Encapsulation also provides protection against malicious attacks on an application, where code is written with the intent to corrupt or damage the contents of an object. In this section you will discover the C# features that allow you to implement encapsulation in created objects.

This section covers how to:

- Enforce encapsulation by using properties
- Enforce encapsulation by using accessors, including

- Enforce encapsulation by using explicit interface implementation

Access modifiers

C# allows objects to encapsulate data and methods by providing programmers with the ability to mark members of a type with *access modifiers* that control access to them.

Public and private access

There are a number of different access modifiers, and you can start by considering *public* and *private*. A member of a type given the public access modifier can be accessed by code that is outside that type, whereas a private member of a type can only be accessed by code running inside the type.

Enforce encapsulation by using properties

A property in an object provides a way that a programmer can encapsulate data. Let's start by considering how public data is accessed in a class. Listing 2-27 shows a program that uses a class called `Customer`, which you might use to store customer records. At the moment the `Customer` class contains a single data member called `Name`, which is a string containing the name of the customer. The `Name` member has been made public, so that it can be set to the name of the customer.

LISTING 2-27 Public data members

Click here to view code image

```
using System;

namespace LISTING_2_27_Public_data_members
{
    class Customer
    {
        public string Name;
    }

    class Program
    {
        static void Main(string[] args)
        {
            Customer c = new Customer();
            c.Name = "Rob";
            Console.WriteLine("Customer name: {0}",

            Console.ReadKey();
        }
    }
}
```

This program works well, but it doesn't provide any control over the contents of the customer name. The name of a customer can be set to any string, including an empty string. You should stop users of the `Customer` object from setting a customer name to an empty string. We call this "enforcing business rules" on our applications. You may have other rules to enforce, which restrict the characters that can be used in a name and set limits for the minimum and maximum length of a customer name. But for now, we will just focus on how to manage access to the name of a customer and stop invalid customer names from being created.

The program in Listing 2-28 shows how to create a `Name` property in the `Customer` class that performs validation of the name. A property is declared as having a `get` behavior and a `set` behavior. The `set` behavior is used when a program sets a value in the property and the `get` behavior is used when a program gets a value from the property. The `get` behavior for the `Name` property returns the value of a `private` class member variable called `_nameValue`, which holds the value of the name of the customer. Within the `set` behavior for the `Name` property keyword `value` represents the value being assigned to the property. If this value is an empty string the set behavior throws an exception to prevent an empty string begin set as a Name. If the value is valid, the set behavior sets `_nameValue` to the incoming name. Note that there is a C# convention that private members of a class have identifiers that start with an underscore (_) character.

LISTING 2-28 Using a property

Click here to view code image

```
using System;

namespace LISTING_2_28_Using_a_property
{
    class Customer
    {
        private string _nameValue;

        public string Name
        {
            get
            {
                return _nameValue;
            }
```

```
                throw new Exception("Invalid cu
            _nameValue = value;
        }
    }
}
```

Adding validation to the `Name` property does not change how the property is used in a program. The code here sets the name of customer `c` to `Rob` and then tries to set the name to an empty string, which causes an exception to be thrown:

Click here to view code image

```
Customer c = new Customer();
c.Name = "Rob";
Console.WriteLine("Customer name: {0}", c.Name);

Console.ReadKey();
// the following statement will throw an exception
c.Name = "";
```

The `Name` property can throw an exception when an attempt is made to set an empty string. Do this to ensure that the user of the `Customer` class is always made aware of any error conditions. You can make the `set` behavior ignore invalid names. or set the name to a default string when an invalid name is provided.

Properties provide a powerful way to enforce encapsulation, which is very natural in use. The user of a property might not even be aware that code is running when they perform what seems like a simple assignment. You can provide "read only" properties by creating properties that only contain a `get` behavior. These are useful if you want to expose multiple views of the data in an object, for example a `Thermometer` class can provide different properties that give the temperature value in Fahrenheit and Centigrade. You can also create "write only properties" by only providing the `set` behavior, although this ability is less frequently used. It is also possible to set different access modifiers for the get and set behaviors, so that a get behavior can be public for anyone to read, but the set behavior is private so that only code running inside the class can assign values to the property.

In the program in Listing 2-28 the private data member `_nameValue` holds the value of the name that is being managed by the property. This value is called the *backing value* of the property. If you just want to implement a class member as a property, but don't want to get control when the property is accessed, you can use *auto-implemented* properties. The statement here creates an integer property called `Age`. The C# compiler automatically creates the backing values. If you want to add get and set behaviors and your own backing value later, you can do this.

```
public int Age {get; set;}
```

Enforce encapsulation by using accessors

As a general rule, data held within a type should be made private and methods (which allow managed access to data inside the type) should be made public. Properties provide a way to manage access to individual values in a class, so you can consider how to use accessor methods to provide access to elements in a class.

The program in Listing 2-29 shows how `public` and `private` access modifiers can be used to create a very simple bank account application that uses methods to provide access to the account balance value. The member variable `_accountBalance` is made `private` to the `BankAccount` class. This means that it cannot be accessed by code running outside the `BankAccount`. The method members `PayInFunds`, `GetBalance`, and `WithdrawFunds` are declared as `public`, which means that code running outside the `BankAccount` class can use these methods to interact with the account balance value in a managed way.

LISTING 2-29 Creating accessor methods

Click here to view code image

```
using System;

namespace LISTING_2_29_Creating_accessor_methods
{
    class BankAccount
    {
        private decimal _accountBalance = 0;

        public void PayInFunds(decimal amountToPayI
        {
            _accountBalance = _accountBalance + amo
        }

        public bool WithdrawFunds(decimal amountToW
        {
            if (amountToWithdraw > _accountBalance)
                return false;

            _accountBalance = _accountBalance - amo
            return true;
```

```
        public decimal GetBalance()
        {
            return _accountBalance;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            BankAccount a = new BankAccount();
            a.PayInFunds(50);
            Console.WriteLine("Pay in 50");
            a.PayInFunds(50);
            if (a.WithdrawFunds(10))
                Console.WriteLine("Withdrawn 10");
            Console.WriteLine("Account balance is:
            Console.ReadKey();
        }
    }
}
```

Making a data member of a class `private` will stop direct access to that data member. In other words, the following statements will not compile. The cunning attempt to increase the balance of an account by a million is not permitted by the compiler because the `_accountBalance` member has been declared as `private` to the `BankAccount` class.

Click here to view code image

```
BankAccount a = new BankAccount();
a._accountBalance = a._accountBalance + 1000000;
```

Of course, there is nothing to stop a wayward bank programmer from adding the following statement to the banking application. This would add a million to the account and is accepted by the C# compiler as completely legal code.

Click here to view code image

```
a.PayInFunds(1000000);
```

In a real bank a request to pay in funds is accompanied by some details that accredit the transaction, and the `PayInFunds` behavior logs each transaction for audit. This means that, from a design perspective, making a class member `private` and only providing `public` methods that allow access to that member is a good first step to creating secure code, but you also may also have to make sure that you provide a secure workflow that manages access to the data. In this respect the behavior of the `BankAccount` class in Listing 2-29 is sadly lacking; you might like to consider the effect of withdrawing a negative amount of money from the account. You might also like to consider how to fix this issue.

Default access modifiers

If you don't specify an access modifier for a member of a type, the access to that member will default to `private`. In other words, if you want to make a member visible outside the type, it must be done explicitly by adding `public`. This means that you don't actually have to add the access modifier `private` to your private members, but I strongly advise you to do this.

Protected access

Making a member of a class `private` will prevent code in any external class from having access to that data member. The `protected` access modifier makes a class member useable in any classes that extend the parent (base) class in which the member is declared. Listing 2-27 shows an `OverdraftAccount` that adds an overdraft facility to the `BankAccount` class. The `OverdraftAccount` contains an override of the `WithdrawFunds` method that allows the account holder to draw out more than they have in their account, up to the limit of their overdraft. This works because the `accountBalance` member of the `BankAccount` class now has the `protected` access modifier.

LISTING 2-30 Protected access

Click here to view code image

```
class OverdraftAccount: BankAccount
{
    decimal overdraftLimit = 100;

    public override bool WithdrawFunds(decimal amou
    {
        if (amountToWithdraw > _accountBalance + ov
            return false;

        accountBalance = accountBalance - amountToW
        return true;
    }
}
```

balance. Doing this makes it very easy for a malicious programmer to gain access to the protected member by extending the parent class. I tend to use the protected access modifier to limit access to helper methods that have no meaningful use outside of the class hierarchy.

Internal access

The `internal` access modifier will make a member of a type accessible within the assembly in which it is declared. You can regard an assembly as the output of a C# project in Visual Studio. It can be either an executable program (with the language extension .exe) or a library of classes (with the language extension .dll). Internal access is most useful when you have a large number of cooperating classes that are being used to provide a particular library component. These classes may want to share members which should not be visible to programs that use the library. Using the access modifier `internal` allows this level of sharing.

Readonly access

The `readonly` access modifier will make a member of a type read only. The value of the member can only be set at declaration or within the constructor of the class.

Access modifiers and classes

The private, public, protected, and internal access modifiers can also be applied to classes that are declared nested inside other classes. LISTING 2-29 shows how a `BankAccount` class could contain an `Address` class that is used to hold address information for account holders. If this `Address` class is made protected it can only be used in the `BankAccount` class and in classes that extend that class.

The classes in Listing 2-31 show how this would work. The protected `Address` class is defined in the `BankAccount` class. The `OverdraftAccount` class can contain a variable of type `Address` because it is a child of `BankAccount`. The `OverDraft` account contains a member called `GuarantorAddress` that that gives the address of the person nominated by the account holder to guarantee the overdraft.

LISTING 2-31 Protected class

Click here to view code image

```
class BankAccount
{
    protected class Address
    {
        public string FirstLine;
        public string Postcode;
    }

    protected decimal accountBalance = 0;
}
class OverdraftAccount : BankAccount
{
    decimal overdraftLimit = 100;

    Address GuarantorAddress;
}
```

Enforce encapsulation by using explicit interface implementation

Looking at Skill 2.4, "Create and implement a class hierarchy," you will see that a class can be defined as implementing an interface. Check out this section now if you're not clear on interface details.

When a class implements an interface it contains methods with signatures that match the ones specified in the interface. You can use an explicit interface implementation to make methods implementing an interface only visible when the object is accessed via an interface reference.

If this sounds confusing, then the best way to understand what is happening is to consider why you would want to do this. You might make an `IPrintable` interface that specifies methods used to print any object. This is a good idea, because now a printer can be asked to print any item that is referred to by a reference of `IPrintable` type. In other words, any object that implements the methods in `IPrintable` can be printed.

If you think about it, the methods in the `IPrintable` interface only have meaning when being used by something trying to print an object. It is not sensible to call the printing methods in any other context than via the `IPrintable` reference. You can achieve this by making the implementation of the printing methods *explicit*, thus adding the interface name to the declaration of the method body.

The code here shows a `Report` class that implements the `IPrintable` interface. The `Report` class contains two methods: `GetPrintableText` and `GetTitle`, which are declared in the `IPrintable` interface. These methods have been made explicit implementations of the interface by preceding the method name with the name of the interface they are implementing.

Click here to view code image

```
class Report : IPrintable
{
    string IPrintable. GetPrintableText(int pageWid
    {
        return "Report text to be printed";
    }
```

```
    {
        return "Report title to be printed";
    }
}
```

Once you have done this, the only way to access these methods in a `Report` instance is by a reference of `IPrintable` type. Figure 2-6 below shows the effect in Visual Studio of making the `GetPrintableText` and `GetTitle` methods explicit implementations of the `IPrintable` interface. The Intellisense shows that a reference to `Report` does not expose either of these methods.

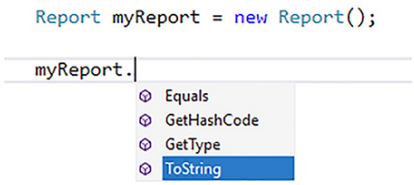

FIGURE 2-6 Explicit method implementation

If, however, you use Intellisense on a reference to an object that implements the `IPrintable` interface, you will see how the two methods in the interface are now available. Figure 2.7 shows the effect of using a reference to the `IPrintable` interface. The Intellisense for this reference includes the two interface methods.

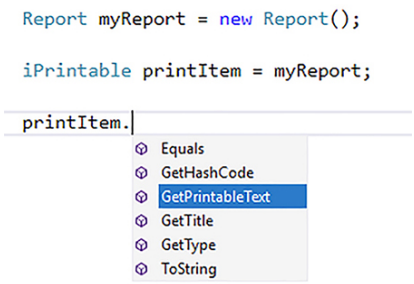

FIGURE 2-7 Using a reference to an interface

When you use an interface in a program you should ensure that all the implementations of any interface methods are explicit. This reduces the chances of the interface methods being used in an incorrect context.

Resolving duplicate method signatures by using explicit implementation

When a class implements an interface it must contain an implementation of all methods that are defined in the interface. Sometimes a class may implement multiple interfaces, in which case it must contain all the methods defined in all the interfaces. This can lead to problems, in that two interfaces might contain a method with the same name. For example the `IPrintable` and `IDisplay` interfaces might both specify a method called `GetTitle` to get the title of a document. The designers of the interfaces will have added these methods to the interface because a printed page and an on-screen menu might both need to have titles that need to be displayed. This can lead to confusion when an object implements both interfaces; either a document or a menu might end up with the incorrect title. Listing 2-32 shows how the `Report` class can contain explicit implementations of both `GetTitle` methods.

LISTING 2-32 Printing interface

Click here to view code image

```
interface IPrintable
{
    string GetPrintableText(int pageWidth, int page
    string GetTitle();
}

interface IDisplay
{
    string GetTitle();
}

class Report : IPrintable, IDisplay
{
    string iPrintable.GetPrintableText(int pageWidt
    {
        return "Report text to be printed";
    }

    string IPrintable.GetTitle()
    {
        return "Report title to be printed";
    }

    string IDisplay.GetTitle()
```

```
        }
    }
```

SKILL 2.4: CREATE AND IMPLEMENT A CLASS HIERARCHY

You now know how to create your own types, consume them, and encapsulate their data and state to reduce the chances of accidental or malicious damage to objects in your program.

In this section you will discover how to create component objects that can be managed in terms of their abilities. You will also find out how to reduce the amount of code that you need to write for an application by using class hierarchies to reuse code in objects. Finally, we will explore some of the interfaces that are used by the .NET framework to manage objects and consider how these can be usefully implemented by created objects.

This section covers how to:

- Design and implement an interface
- Inherit from a base class
- Create and implement classes based on the `IComparable`, `IEnumerable`, `IDisposable`, and `IUnknown` interfaces

Design and implement an interface

Many items in the real world are designed as "components" that are interchangeable. You can connect a wide variety of devices to the main power in our house because, as far as the power supply is concerned, each device can be regarded in the same way – as a device that requires a particular voltage to work. You can think of the voltage and power elements of a power specification as an "interface" that defines how an electrical device can be connected to a power supply.

For this to work the people who make main boards and the people who make graphics adapters must agree on an interface between two devices. The interface specification describes exactly how the two components interact, for example which signals are inputs, which signals are outputs, and so on. Any main board which contains a socket built to the interface standard can accept a graphics card built to the same standard.

So, from the point of view of hardware, components are possible because we have created standard interfaces to describe exactly how they fit together. Software components are exactly the same.

At the moment you might not see a need for software components. When creating a system you must work out what each of the parts of it need to do, and then create those parts and fit them together. It may not be obvious why components are required. However, systems designed without components are like a computer with a graphics adapter that is part of the main board. It is not possible to change the main boad or the graphics adapter because they are "hard wired" together.

Software components and interfaces

One point to consider here is that we are not talking about the user interface to our program. The user interface is the way a person using a program makes it work for them. These are usually either text-based (the user types in commands and gets responses) or graphical (the user clicks "buttons" on a screen using the mouse).

An interface in a C# program specifies how a software component could be used by another software component. So, instead of starting to build an application by designing classes. you should instead be thinking about describing their interfaces (what each software component will do). How the component performs its function can be encapsulated inside the component.

A C# interface contains a set of method signatures. If a class contains an implementation of all of the methds described in the interface it can be defined as "implementing" that interface. Interfaces allow a program to regard objects in terms of their abilities (or the interfaces that they implement), rather than what type an object actually is. Humans do this all the time. If you hire someone to paint your house you will be dealing with that person in terms of their ability to paint, not who they really are.

As an example of a situation where interfaces are useful, consider a printing service that will print objects in an application. You can create an interface containing the definition of the methods that can be used to print a document on paper. Listing 2-33 shows the definition of an `IPrintable` interface that allows a printer to get the text to be printed and the title of a document from an object that wishes to be printed.

LISTING 2-33 `IPrintable` interface

Click here to view code image

```
interface IPrintable
{
    string GetPrintableText(int pageWidth, int page
    string GetTitle();
}
```

printer will put onto paper. The `Report` class shown below implements the `IPrintable` interface and contains implementations of the `GetPrintableText` and `GetTitle` methods.

Click here to view code image

```
class Report : IPrintable
{
    public string GetPrintableText(int pageWidth, i
    {
        return "Report text";
    }

    public string GetTitle()
    {
        return "Report title";
    }
}
```

A printer object can now be created that will accept and print reports. along with any other objects that implements the `IPrintable` interface. The `ConsolePrinter` class below provides a method called `PrintItem` that will output printable items onto the console. You can create other printers that print onto paper, web pages, or Adobe PDF documents.

Click here to view code image

```
class ConsolePrinter
{
    public void PrintItem( iPrintable item)
    {
        Console.WriteLine(item.GetTitle());
        Console.WriteLine(item.GetPrintableText(pag
    }
}
```

Note that the interface decouples the printer from the object being printed. If you create new document types that need to be printed, they can be added to the application without modifying the printer class. Conversely, you can add new types of printer and be sure that the new printer type can be used to print any document object.

You can even create test objects that can be used at either end of the interface. A test printer object can be created to make sure that an object implements all the print methods correctly and a test document object can be used to test the behaviors of a printer. The printing ability of a printer (for example add a document to the print queue and check if a document has been printed) can even be exposed as another interface, to make it even easier to switch between printing devices.

Design an interface

You can build your knowledge of interfaces by considering how to create an interface to describe the behavior of a class that will implement a bank account. In Listing 2-29 we looked at some behaviors that can be used to implement a bank account. You can express these behaviors in the form of an interface that sets out the methods that a bank account class should implement.

LISTING 2-34 IAccount interface

Click here to view code image

```
public interface IAccount
{
   void PayInFunds ( decimal amount );
   bool WithdrawFunds ( decimal amount );
   decimal GetBalance ();
}
```

Listing 2-34 says that the `IAccount` interface is comprised of three methods: one to pay money in, another to withdraw it, and a third to return the balance on the account. From the balance management point of view this is all you need. Note that at the interface level we are not saying how any of these tasks should be performed, but we are just identifying the tasks.

An interface is placed in a source file just like a class, and compiled in the same way. It sets out a number of methods that relate to a particular task or role, which in this case is what a class must do to be considered a bank account. There is a convention in C# programs that the name of an interface starts with the letter I. It is interesting to note that one of the refactoring tools available in Visual Studio is one that will extract the method signatures from a class and create an interface that contains those methods.

In the case of the bank account, you can now create a class implementing the interface, so that it can be thought of as an account component, irrespective of what it really is.

Click here to view code image

```
public class BankAccount: IAccount
{
    private decimal balance = 0;
```

```
    {
        if ( _balance < amount )
        {
            return false ;
        }
        _balance = _balance - amount ;
        return true;
    }

    void IAccount .PayInFunds ( decimal amount )
    {
        _balance = _balance + amount ;
    }

    decimal IAccount.GetBalance ()
    {
        return _balance;
    }
}
```

This code does not look much different from the previous BankAccount class. One difference is on the top line:

Click here to view code image

```
public class BankAccount: IAccount
{
...
```

This tells the compiler that this class implements the `IAccount` interface. This means that the class contains implementations of all the methods described in the interface. Note that I have added the inferface name to all the interface methods so that they are *explicitly implemented*. This means that these methods are only exposed when the `BankAccount` object is referred to by a reference of type `IAccount`. You can refer to the section "Enforce encapsulation by using explicit interface implementation" in Skill 2.4 for more details of how this works.

References to Interfaces

A program can now work with a `BankAccount` instance as an object of type `BankAccount` or an object that implements the `IAccount` interface. People do this all the time. You can think of me as Rob Miles the individual or Rob Miles the writer.

If you think of me as a writer, you would use an interface that contains methods with names such as `WriteBook`. And you can use these same methods with any other person who can behave like a writer. From the point of view of a publisher, which has to manage a large number of interchangeable writers, it is much more useful to think of me as a writer, rather than Rob Miles the individual. Should I ever join a golf club, that club will think of me in terms of my "`IGolfer`" interface.

So, with interfaces you are moving away from considering classes in terms of what they are, and starting to think about them in terms of what they can do. In the case of your bank, this means that you want to deal with objects in terms of `IAccount`, (the set of account abilities) rather than `BankAccount` (a particular account class).

In C# terms this means that you need to create reference variables that refer to objects in terms of interfaces they implement, rather than the particular type that they are. It turns out that this is quite easy:

Click here to view code image

```
IAccount account = new BankAccount ();
account.PayInFunds(50);
Console.WriteLine("Balance: " + account.GetBalance(
```

The `account` variable is allowed to refer to objects that implement the `IAccount` interface. The compiler will check to make sure that `BankAccount` does this, and if it does, the compilation is successful.

Note that there will never be an instance of `IAccount` interface. It is simply a way that you can refer to something which has that ability (i.e. contains the required methods).

This is the same in real life. There is no such physical thing as a "writer," merely a large number of people who can be referred to as having that particular ability or role.

Inherit from a base class

One of the fundamental principles of software development is to ensure that you create every piece of code in an application precisely once. Rather than copying code from one part of a program to another, you would create a method and then call that method.

The reason to do this is quite simple. If you find a bug in a piece of your program you only want to have to fix the bug once. You don't want to have to search through the program looking for all of the places where you have used a particular lump of code. If you have to do this, you may miss one place and end up with a program that is mostly fixed, but will still fail sometimes. A class hierarchy is a great way to reuse code so that you only have to create a behavior in one place and can then reuse the behavior

want to work with different types of bank account, or stock item, or customer, or space alien, you can make use of inheritance to save you writing a lot of code. For an example of how this might work, consider the `BankAccount` class that we have been working with. You may need to create another type of account that is exactly the same as an ordinary account but is used by very small children. This type of account, called `BabyAcccount` works in exactly the same way as a `BankAccount`, but it doesn't allow withdrawals of more than 10. Listing 2-35 shows how to create a `BabyAccount` that has the required behaviors.

LISTING 2-35 BabyAccount

Click here to view code image

```
public class BabyAccount : IAccount
{
    private decimal _balance = 0;

    bool IAccount.WithdrawFunds(decimal amount)
    {
        if (amount > 10)
        {
            return false;
        }
        if (_balance < amount)
        {
            return false;
        }
        _balance = _balance - amount;
        return true;
    }

    void IAccount.PayInFunds(decimal amount)
    {
        _balance = _balance + amount;
    }

    decimal IAccount.GetBalance()
    {
        return _balance;
    }
}
```

The good news is that because the `BabyAccount` is a component that implements the `IAccount` interface, it will work with any of the classes that work with accounts. When you create an account you just have to ask if a standard account or a baby account is required. The rest of the system can then pick up this object and use it without caring about what it is.

The bad news is that you have duplicated a lot of code. The `BabyAccount` class contains `GetBalance` and `PayInFunds` methods that are copies of the ones in the `BankAccount` class. Why write the same code twice?

What you really want to do is pick up all the behaviors in the `BankAccount` and then just change the one method that needs to behave differently. This can be done in C# using inheritance. When you create the `BabyAccount` class, you can tell the compiler that it is based on the `BankAccount` class:

Click here to view code image

```
public class BabyAccount : BankAccount, IAccount
{
}
```

The key thing here is the word `BankAccount` after the class name. This is the name of the class that `BabyAccount` is extending. This means that everything that `BankAccount` can do, `BabyAccount` can do. `BankAccount` is called the *base* or *parent* class of `BabyAccount`.

You can now write code like:

Click here to view code image

```
IAccount b = new BabyAccount();
b.PayInFunds(50);
```

This works because, although `BabyAccount` does not have a `PayInFunds` method, the base class does. This means that the `PayInFunds` method from the `BankAccount` class is used at this point.

Instances of the `BabyAccount` class have abilities which they pick up from their base class. In fact, at the moment, the `BabyAccount` class has no behaviors of its own; it gets everything from its base class.

The is and as operators

A program can use the `is` and `as` operators when working with class hierarchies and interfaces. The `is` operator determines if the type of a given object is in a particular class hierarchy or implements a specified interface. You apply the `is` operator between a reference variable and a type or interface and the operator returns true if the reference can be made to refer to objects of that type.

The code below prints out a message if the variable `x` refers to an object

```
if (x is IAccount)
    Console.WriteLine("this object can be used as a
```

The `as` operator takes a reference and a type and returns a reference of the given type, or `null` if the reference cannot be made to refer to the object.

The code below creates an `IAccount` reference called `y` that either refers to `x` (if `x` implements the `IAccount` interface) or is `null` (if `x` does not implement the IAccount interface).

Click here to view code image

```
IAccount y = x as IAccount;
```

You might be wondering why we have the `as` operator, when we can already use casting to convert between reference types (see the "Cast types" section earlier for details on casting). The difference is that if a cast cannot be performed, a program will throw an exception, whereas if the `as` operator fails it returns a null reference and the program continues running.

Overriding methods

You saw how to override methods in Skill 2.1 in the "Overloading and overriding methods" section. Overriding replaces a method in a base class with a version that provides the behavior appropriate to a child class. In the case of the `BabyAccount`, you want to change the behavior of the one method that you are interested in. You want to replace the `WithdrawFunds` method with a new one.

Listing 2-36 shows how this is done. The keyword `override` means "use this version of the method in preference to the one in the base class." This means that when the code here runs the call, `PayInFunds` will use the method in the parent (since that has not been overridden), but the call of `WithdrawFunds` will use the method in `BabyAccount`.

LISTING 2-36 Overridden WithdrawFunds

Click here to view code image

```
public class BankAccount : IAccount
{
    protected decimal _balance = 0;

    public virtual bool WithdrawFunds(decimal amoun
    {
        if (_balance < amount)
        {
            return false;
        }
        _balance = _balance - amount;
        return true;
    }

    void IAccount.PayInFunds(decimal amount)
    {
        _balance = _balance + amount;
    }

    decimal IAccount.GetBalance()
    {
        return _balance;
    }
}

public class BabyAccount : BankAccount, IAccount
{
    public override bool WithdrawFunds(decimal amou
    {
        if (amount > 10)
        {
            return false;
        }

        if (_balance < amount)
        {
            return false;
        }
        _balance = _balance - amount;
        return true;
    }
}
```

There are some other things to be aware of in Listing 2-36:

- The C# compiler needs to know if a method is going to be overridden. This is because it must call an overridden method in a slightly different way from a "normal" one. The `WithDrawFunds` method in the `BankAccount` class has been declared as *virtual* so that the compiler knows it may be overridden. It might be overridden in classes which are children of the parent class.
- The C# language does not allow the overriding of explicit

`WithDrawFunds` method in the `BankAccount` class is declared as `virtual`, but it has not been declared as an interface method.

- The `WithdrawFunds` method in the `BabyAccount` class makes use of the `_balance` value that is declared in the parent `BankAccount` class. To make this possible the `_balance` value has had its access modifier changed from `private` to `protected` so that it can be used in classes that extend the `BankAccount` class.

The ability to override a method is very powerful. It means that you can make more general classes (for example the `BankAccount`) and customize it to make them more specific (for example the `BabyAccount`). Of course, this should be planned and managed at the design stage. This calls for information to be gathered from the customer and used to decide which parts of a behavior need to be changed during the life of the project.

You can make the `WithDrawFunds` method virtual because you discovered that different accounts might need to withdraw funds in a different way. Note that the `PayInFunds` and `GetBalance` methods have not been made `virtual` because you will always be using the versions of these declared in the `BankAccount` class.

Using the base method

Remember the importance of only writing any given piece of code once? Well, it looks as if I'm breaking my own rules here, in that the `WithDrawFunds` method in the `BabyAccount` class contains all of the code of the method in the parent class.

We have already noted that we don't like this much, in that it means that the balance value has to be made more exposed than we might like. It is now protected (visible to any class in the `BankAccount` class hierarchy) rather than private to just the `BankAccount` class. Fortunately, the designers of C# have thought of this and have provided a way that you can call the base method from one which overrides it.

The word `base` in this context means "a reference to the thing which has been overridden." I can use this to make the `WithDrawFunds` method in my `BabyAccount` much simpler.

Listing 2-37 shows how this works. The very last line of the `WithDrawFunds` method makes a call to the original `WithDrawFunds` method in the parent class (the one that the method overrides). This attempts to perform a withdrawal and returns true if the withdrawal works.

LISTING 2-37 The base method

Click here to view code image

```
public override bool WithdrawFunds(decimal amount)
{
    if (amount > 10)
    {
        return false;
    }
    else
    {
        return base.WithdrawFunds(amount);
    }
}
```

It's important to understand what we're doing here, and why we're doing it:

- I don't want to have to write the same code twice
- I don't want to make the _balance value visible outside the `BankAccount` class.

The use of the word `base` to call the overridden method solves both of these problems rather beautifully. Because the method call returns a bool result you can just send whatever it delivers. By making this change you can put the `_balance` back to `private` in the `BankAccount` because it is not accessed by the `WithdrawFunds` method.

Note that there are other useful spin-offs here. If I need to fix a bug in the behavior of the `WithDrawFunds` method I just fix it once, in the top-level class, and then it is fixed for all the classes which call back to it.

Replacing methods in base classes

C# allows a program to replace a method in a base class by simply creating a new method in the child class. In this situation there is no overriding, you have just supplied a new version of the method. In fact, the C# compiler will give you a warning that indicates how you should provide the keyword new to indicate this):

Click here to view code image

```
public class BabyAccount : BankAccount,IAccount
{
   public new bool WithdrawFunds (decimal amount)
   {
      if (amount > 10)
      {
         return false ;
      }
      if ( balance < amount)
```

```
        }
        _balance = _balance - amount ;
        return true;
    }
}
```

Note that a replacement method is not able to use `base` to call the method that it has overridden, because it has not overridden a method, it has replaced it. I cannot think of a good reason for replacing a method, and I'm mentioning this feature of C# because I feel you need to know about it; and not because you should use it.

Stopping overriding

Overriding is very powerful. It means that a programmer can just change one tiny part of a class and make a new one with all the behaviors of the parent. This goes well with a design process, so as you move down the "family tree" of classes you get more and more specific implementations.

However, overriding/replacing is not always desirable. Consider the `GetBalance` method in the `BankAccount` class. This is never going to need a replacement. And yet a malicious programmer can write their own and override or replace the one in the parent:

Click here to view code image

```
public new decimal GetBalance ()
{
    return 1000000;
}
```

This is the banking equivalent of the bottle of beer that is never empty. No matter how much cash is drawn out of the account, it always has a balance value of a million pounds! A programmer could insert this into a child class and enjoy a nice spending spree. What this means is that you need a way to mark some methods as not being able to be overridden. C# does this by giving us a sealed keyword which means "You can't override this method any more".

You can only seal an overriding method and sealing a method does not prevent a child class from replacing a method in a parent. However, you can also mark a class as sealed. This means that the class cannot be extended, so it cannot be used as the basis for another class. The `BabyAccount` below cannot be the base of any other classes.

Click here to view code image

```
public sealed class BabyAccount : CustomerAccount,I
{
    .....
}
```

Constructors and class hierarchies

A constructor is a method which gets control during the process of object creation. It is used to allow initial values to be set into an object. You can add a constructor to the `BankAccount` class that allows an initial balance to be set when an account is created. Listing 2-38 shows the constructor method in the `BankAccount` class.

LISTING 2-38 BankAccount constructor

Click here to view code image

```
public class BankAccount : IAccount
{
    private decimal _balance ;
    public BankAccount ( decimal initialBalance)
    {
        _balance = initialBalance;
    }
}
```

You can now set the initial balance of an account when one is created. The following statement creates a new bank account with an initial balance of 100.

Click here to view code image

```
IAccount a = new BankAccount(100);
```

Unfortunately, adding a constructor like this to a base class in a class hierarchy has the effect of breaking all the child classes. The reason for this is that creating a child class instance involves creating an instance of the `base` class. When the program tries to create a `BabyAccount` it must first create a `BankAccount`. Creating a `BankAccount` involves the use of its constructor to set the initial balance of the `BankAccount`. The `BabyAccount` class must contain a constructor that calls the constructor in the parent object to set that up. The code here shows how this would work. The constructor for the `BabyAccount` makes a call of the constructor for the `base` class and passes the initial balance into that constructor.

```
public class BabyAccount : BankAccount, IAccount
{
    public BabyAccount(int initialBalance) : base(i
    {
    }
}
```

In "Skill 2.1 Creating Types" we discussed the use of the use of the keyword `this` when writing constructors. You saw that the keyword is used to allow a constructor in a class to call other constructors in that class. The `base` keyword in this context is analogous to the `this` keyword, except that the constructor that is called is in the base class.

Note that in this case, the actual constructor body for the `BabyAccount` does nothing. However, it might be that other information needs to be stored in a `BabyAccount` (perhaps the name of a parent or guardian of the account holder). This can be set by the `BabyAccount` constructor in the `BabyAccount` constructor body.

Abstract methods and classes

At the moment we are using overriding to modify the behavior of an existing parent method. However, it is also possible to use overriding in a slightly different context. You can use it to force a set of behaviors on items in a class hierarchy. If there are some things that an account must do then we can make these abstract and then force the child classes to provide the implementation.

For example, in the context of the bank application you might want to provide a method that creates the text of a warning letter to the customer telling them that their account is overdrawn. This will have to be different for each type of account (you don't want to use the same language to a baby account holder as you do for a normal account). This means that at the time you create the bank account system you know that you need this method, but you don't know what it does in every situation.

You can provide a virtual "default" method in the `BankAccount` class and then rely on the programmers overriding this with a more specific message, but you then have no way of making sure that they really do perform the override. C# provides a way of flagging a method as *abstract*. This means that the method body is not provided in this class, but will be provided in a child class:

Click here to view code image

```
public abstract class BankAccount
{
    public abstract string WarningLetterString();
}
```

The fact that the `BankAccount` class contains an abstract method means that the class itself is abstract (and must be marked as such). It is not possible to make an instance of an abstract class. If you think about it this is sensible. An instance of `BankAccount` would not know what to do if the `WarningLetterString` method was ever called.

An abstract class can be thought of as a kind of template. If you want to make an instance of a class based on an abstract parent you must provide implementations of all the abstract methods given in the parent.

Abstract classes and interfaces

You might decide that an abstract class looks a lot like an interface. This is true, in that an interface also provides a "shopping list" of methods which must be provided by a class. However, abstract classes are different in that they can contain fully implemented methods alongside the abstract ones. This can be useful because it means you don't have to repeatedly implement the same methods in each of the components that implement a particular interface.

A class can only inherit from one parent, so it can only pick up the behaviors of one class. Some languages support *multiple inheritance*, where a class can inherit from multiple parents. C# does not allow this.

References in class hierarchies

A reference to a base class in a class hierarchy can refer to an instance of any of the classes that inherits from that base class. In other words, a variable declared as a reference to `BankAccount` objects can refer to a `BabyAccount` instance.

However, the reverse is not true. A variable declared as a reference to a `BabyAccount` object cannot be made to refer to a `BankAccount` object. This is because the `BabyAccount` class may have added extra behaviors to the parent class (for example a method called `GetParentName`). A `BankAccount` instance will not have that method.

However, I much prefer it if you manage references to objects in terms of the interfaces than the type of the particular object. This is much more flexible, in that you're not restricted to a particular type of object when developing the code.

Create and implement classes based on the IComparable, IEnumerable, IDisposable, and IUnknown interfaces

The .NET framework provides services that make use of particular behaviors that may be provided by a given object. Each of these behaviors

In this section we are going to consider some of these interfaces, when they are useful, and how you can use them in our programs.

IComparable

The `IComparable` interface is used by .NET to determine the ordering of objects when they are sorted. Below is the definition of the method from the C# .NET library. You can see that the interface contains a single method, `CompareTo`, which compares this object with another. The `CompareTo` method returns an integer. If the value returned is less than 0 it indicates that this object should be placed before the one it is being compared with. If the value returned is zero, it indicates that this object should be placed at the same position as the one it is being compared with and if the value returned is greater than 0 it means that this object should be placed after the one it is being compared with.

Click here to view code image

```
public interface IComparable
{
    //
    // Summary:
    //     Compares the current instance with anoth
           returns
    //     an integer that indicates whether the cu
    //     occurs in the same position in the sort
    //
    // Parameters:
    //   obj:
    //     An object to compare with this instance.
    //
    // Returns:
    //     A value that indicates the relative orde
    //     return value has these meanings: Value M
           precedes
    //     obj in the sort order. Zero This instanc
    //     sort order as obj. Greater than zero Thi
           order.
    //
    // Exceptions:
    //   T:System.ArgumentException:
    //     obj is not the same type as this instanc
    int CompareTo(object obj);
}
```

You can allow your bank accounts to be sorted in order of balance by making the `BankAccount` class implement the `IComparable` interface and adding a `CompareTo` method to the `BankAccount` class.

Listing 2-39 shows how to do this. The `CompareTo` method is supplied with an object reference, which must be converted into an `IAccount` reference so that a `BankAccount` instance can be compared with any other object that implements the `IAccount` reference. The `CompareTo` method is simplified by using the `CompareTo` method of the decimal balance method to create the result.

LISTING 2-39 Comparing bank accounts

Click here to view code image

```
public class BankAccount : IAccount, IComparable
{
    public int CompareTo(object obj)
    {
        // if we are being compared with a null obj
        if (obj == null) return 1;

        // Convert the object reference into an acc
        IAccount account = obj as IAccount;

        // as generates null if the conversion fail
        if (account == null)
            throw new ArgumentException("Object is

        // use the balance value as the basis of th
        return this.balance.CompareTo(account.GetBa
    }

}
```

Once the `BankAccount` has been made to implement the `IComparable` interface, you can use the `Sort` behaviors provided by collection types. The code below creates a list of 20 accounts and sorts them using the `Sort` method provided by the `List` collection type.

Click here to view code image

```
// Create 20 accounts with random balances
List<IAccount> accounts = new List<IAccount>();
Random rand = new Random(1);
for(int i=0; i<20; i++)
{
    IAccount account = new BankAccount(rand.Next(0,
    accounts.Add(account);
```

```
accounts.Sort();

// Display the sorted accounts
foreach(IAccount account in accounts)
{
    Console.WriteLine(account.GetBalance());
}
```

Typed IComparable

The `IComparable` interface uses a `CompareTo` method that accepts an object reference as a parameter. This reference should refer to an object of the same type as the object that is doing the comparing, i.e. the `CompareTo` method in type `x` should be supplied with a reference to type `x` as a parameter.

As you can see in Listing 2-39 above, the `CompareTo` method for the `BankAccount` will throw an exception if a program attempts to compare a bank account with something that isn't a bank account. However, this error will be produced when the program runs, not when the program is compiled. The following two statements compile correctly even though they are clearly incorrect because they are trying to compare a bank account to a string.

Click here to view code image

```
BankAccount b = new BankAccount(100);
b.CompareTo("hello");
```

There is another version of the `IComparable` interface that accepts a type. This can be used to create a `CompareTo` that only accepts parameters of a specified type. Listing 2-40 shows how this works. The `CompareTo` method now accepts a parameter of type `BankAccount`, does not need to cast this to a `BankAccount` before performing the comparison and does not have to throw an exception if an invalid type is supplied.

LISTING 2-40 Typed IComparable

Click here to view code image

```
public interface IAccount : IComparable<IAccount>
{
    void PayInFunds(decimal amount);
    bool WithdrawFunds(decimal amount);
    decimal GetBalance();
}

public class BankAccount : IAccount, IComparable<Ba
{
    private decimal balance;

    public int CompareTo(IAccount account)
    {
        // if we are being compared with a null obj
        if (account == null) return 1;

        // use the balance value as the basis of th
        return this.balance.CompareTo(account.GetBa
    }
}
```

Note that for the use of typed `IComparable` to be made to work with objects managed by the `IAccount` interface I had to change the definition of the `IAccount` interface so that it extends the `IComparable` interface. This ensures that any objects that implement the `IAccount` interface also implement the `IComparable` interface. From this you can see that it is possible to create interface hierarchies as well as class hierarchies, but this is not something you will necessarily do a lot of when you write programs.

IEnumerable

Programs spend a lot of time consuming lists and other collections of items. This is called *iterating* or *enumerating*.

You might think of iteration as something that is performed on lists and arrays of values, with a program working through each element in turn. However, iteration is much more than this. Any C# object can implement the `IEnumerable` interface that allows other programs to get an enumerator from that object. The enumerator object can then be used to enumerate (or iterate) on the object.

The code in Listing 2-41 shows how this works. The string type supports enumeration, and so a program can call the `GetEnumerator` method on a string instance to get an enumerator. The enumerator exposes the method `MoveNext()`, which returns the value `true` if it was able to move onto another item in the enumeration. The enumerator also exposes a property called `Current`, which is a reference to the currently selected item in the enumerator.

LISTING 2-41 Get an enumerator

Click here to view code image

```
namespace LISTING_2_41_Get_an_enumerator
{
    class Program
    {
        static void Main(string[] args)
        {
            // Get an enumerator that can iterate t
            var stringEnumerator = "Hello world".Ge

            while(stringEnumerator.MoveNext())
            {
                Console.Write(stringEnumerator.Curr
            }

            Console.ReadKey();
        }
    }
}
```

The program in Listing 2-41 prints out the "Hello world" string one character at a time. The while construction in the Listing 2-41 uses the `MoveNext` and `Current` members of the enumerator to do this. You can consume all the iterators in this way, but C# makes life easier by providing the `foreach` construction, which automatically gets the enumerator from the object and the works through it. Listing 2-42 shows how we would iterate through the same string using a `foreach` construction.

LISTING 2-42 Using foreach

Click here to view code image

```
using System;

namespace LISTING_2_42_Using_foreach
{
    class Program
    {
        static void Main(string[] args)
        {
            foreach (char ch in "Hello world")
                Console.Write(ch);

            Console.ReadKey();
        }
    }
}
```

Making an object enumerable

The `IEnumerable` interface allows you to create objects that can be enumerated within your programs, for example by the `foreach` loop construction. Collection classes, and results returned by LINQ queries implement this interface.

Listing 2-43 shows how to create an enumerable object. It creates a class called `EnumeratorThing` that implements both the `IEnumerable` interface (meaning it can be enumerated) and the `IEenumerator<int>` interface (meaning that it contains a call of `GetEnumerator` to get an enumerator from it). An `EnumaratorThing` instance performs an iteration up to a limit that was set when it was created. Note that the `EmulatorThing` class contains the `Current` property and the `MoveNext` behavior that was used in Listing 2-41 when we wrote code that consumed an enumerator.

LISTING 2-43 Creating an enumerable type

Click here to view code image

```
class EnumeratorThing : IEnumerator<int>, IEnumerab
{
    int count;
    int limit;

    public int Current
    {
        get
        {
            return count;
        }
    }

    object IEnumerator.Current
    {
        get
        {
            return count;
        }
    }

    public void Dispose()
    {
    }

    public bool MoveNext()
    {
```

```
            return true;
        }

        public void Reset()
        {
            count = 0;
        }

        public IEnumerator GetEnumerator()
        {
            return this;
        }

        public EnumeratorThing(int limit)
        {
            count = 0;
            this.limit = limit;
        }
    }
```

You can use an `EnumeratorThing` instance in a `foreach` loop:

```
EnumeratorThing e = new EnumeratorThing(10);

foreach(int i in e)
    Console.WriteLine(i);
```

Using yield

You can create enumerators as we did in Listing 2-43, but this is quite complicated. To make it easier to create iterators C# includes the `yield` keyword. You can see it being used in Listing 2-44. The keyword `yield` is followed by the `return` keyword and precedes the value to be returned for the current iteration. The C# compiler generates all the `Current` and `MoveNext` behaviors that make the iteration work, and also records the state of the iterator method so that the iterator method resumes at the statement following the `yield` statement when the next iteration is requested. The `EnumeratorThing` class in Listing 2-44 provides exactly the same behavior as the one created in Listing 2-43, but it is much simpler.

LISTING 2-44 Using yield

Click here to view code image

```
class EnumeratorThing : IEnumerable<int>
{
    public IEnumerator<int> GetEnumerator()
    {
        for (int i = 1; i < 10; i++)
        yield return i;
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }

    private int limit;
    public EnumeratorThing(int limit)
    {
        this.limit = limit;
    }
}
```

The `yield` keyword does two things. It specifies the value to be returned for a given iteration, and it also returns control to the iterating method. You can express an iterator that returns the values 1, 2, 3, as follows.

Click here to view code image

```
public IEnumerator<int> GetEnumerator()
    {
            yield return 1;
            yield return 2;
            yield return 3;
    }
```

When the first `yield` is reached the enumerator returns the value 1. The next time that the enumerator is called (in other words the next time round the loop) the enumerator resumes at the statement following the first `yield`. This is another `yield` that returns 2. This continues, with the value 3 being returned by the third yield. When the enumerator method ends this has the effect of ending the loop.

IDisposable

C# provides memory management for applications, but it can't control how programs use other resources such as file handles, database connections and lock objects. While an object can get control when it is removed by the garbage collector (see Skill 2.6, "Manage the object lifecycle" for details on this) it is hard for a developer to know precisely when this happens. It may not even happen until the program ends.

The `IDisposable` interface provides a way that an object can indicate

exist in memory, but any attempts to use it will result in the `ObjectDisposedException` being thrown.

Below is the definition of the `IDisposable` interface. If an object implements the `IDisposable` interface it contains a `Dispose` method that can be called to tidy up the object.

Click here to view code image

```
//
// Summary:
//     Provides a mechanism for releasing unmana
public interface IDisposable
{
    //
    // Summary:
    //     Performs application-defined tasks a
    //     resetting
    //     unmanaged resources.
    void Dispose();
}
```

Note that the action of the `Dispose` method depends entirely on the needs of the application. Note also that the `Dispose` method is not called automatically when an object is deleted from memory. There are two ways to make sure that `Dispose` is called correctly; you can call the method yourself in your application, or you can make use of the C# `using` construction.

Listing 2-45 shows how we can create a `CrucialConnection` class that allocates resources that must be disposed of. The `CrucialConnection` class implements `IDisposable` and contains a `Dispose` method which, in this case, just prints a message. The `using` construction creates an instance of the `CrucialConnection` class and is then followed by the block of code that uses this instance. When the program exits, the `using` block the `Dispose` method is called on the `CrucialConnection` instance. When the program runs it will print the message "Dispose called."

LISTING 2-45 Using IDisposable

Click here to view code image

```
using System;

namespace LISTING_2_45_Using_IDisposable
{
    class CrucialConnection : IDisposable
    {
        public void Dispose()
        {
            Console.WriteLine("Dispose called");
        }

    }
    class Program
    {
        static void Main(string[] args)
        {
            using (CrucialConnection c = new Crucia
            {
                // do something with the crucial co
            }

            Console.ReadKey();
        }
    }
}
```

The C# `using` construction is a good way to ensure that `Dispose` is called correctly because it incorporates exception handling so that if an exception is thrown by the code using the object the `Dispose` method is automatically called on the object being used. If you have a number of objects that need to be disposed you can nest `using` blocks inside each other.

IUnknown

In Skill 2.1 we discussed dynamic types in the context of interaction with services provided via the Component Object Model (COM). You saw how a program can use the interop services provided by Microsoft Office to allow you to write a C# program that interacts with Excel. You saw that these interop services returned results that were dynamically typed, and that the program can then use these results simply by assigning them to correctly typed variables.

These interop services are implemented as C# libraries that *marshal* the data between the managed code world of C# and the unmanaged world of COM objects. In the COM world the `IUnknown` interface is the means by which one object describes the interfaces that it makes available for use by others. The `IUnknown` interface provides a means by which .NET applications can interoperate with COM objects at this level. You use it when connecting C# applications to COM objects. You can find more detail on how to do this here: *https://docs.microsoft.com/en-us/dotnet/framework/interop/*

SKILL 2.5: FIND, EXECUTE, AND CREATE TYPES AT RUNTIME BY USING REFLECTION

In the previous sections in this chapter the focus has been on features of C# that allow a program to be mapped onto an application domain. You have seen how to create types that implement behaviors in a problem domain, how to use these types, how to give the types integrity by using cohesion, how to use C# interfaces to turn objects into components and how to use class hierarchies that allow a collection of related component objects to share common behaviors.

In this section we're going to look at some C# features that are concerned with the management and manipulation of a code base. These features are not all necessarily concerned with creating an application that solves a problem; they are more concerned with easing the management and generation of software components. You will see how to add descriptive information to classes by the use of attribute objects, how to write programs that generate executable code as their output, and how to use the `System.Reflection` namespace to create programs that can analyze the content of software objects.

This section covers how to:

- Create and apply attributes
- Read attributes
- Generate code at runtime by using CodeDom and lambda Lambda expressions
- Use types from the System.Reflection namespace, including Assembly, PropertyInfo, MethodInfo and Type

Create and apply attributes

When describing a piece of data you can talk in terms of the attributes of that data. For example, a data file containing music can have attributes that gave the name of the recording artist. In this context the recording artist attribute can be described as a piece of *metadata*. Metadata is "data about data." In the case of the music; the data is the music itself, and the metadata is the name of the artist.

C# allows you to add metadata to an application in the form of *attributes* that are attached to classes and class members. An attribute is an instance of a class that extends the `Attribute` class.

The Serializable attribute

The first attribute that you encounter as a developer is usually the `Serializable` attribute. This attribute doesn't actually hold any data, it is the fact that a class has a `Serializable` attribute instance attached to it means that the class is may be opened and read by a serializer.

A serializer takes the entire contents of a class and sends it into a stream. There are possible security implications in serializing a class and so C# requires that a class should "opt in" to the serialization process. For more details on serialization, take a look at Skill 4.4, "Serialize and deserialize data."

Listing 2-46 shows a `Person` class that contains name and age information. The `Person` class has the `Serializable` attribute attached to it; this is expressed by giving the name of the attribute enclosed in square brackets, just before the declaration of the class. The `Person` class also contains a `NonSerialized` attribute that is applied to the `screenPosition` member variable. This member of the class is only used to manage the display of a `Person` object and should not be saved when it is serialized. This tells the serializer not to save the value of `screenPosition`.

LISTING 2-46 The serializable attribute

Click here to view code image

```
[Serializable]
public class Person
{
    public string Name;

    public int Age;

    [NonSerialized]
    // No need to save this
    private int screenPosition;

    public Person(string name, int age)
    {
        Name = name;
        Age = age;
        screenPosition = 0;
    }
}
```

Note that some serializers, notably the `XMLSerializer` and the `JSONSerializer`, don't need classes to be marked as serializable before they can work with them. You can find out more about serialization in Skill 4.4, "Serialize and deserialize data."

`System.Diagnostics` namespace. Listing 2-47 shows how it is used. The symbols `TERSE` and `VERBOSE` can be used to select the level of logging that is performed by a program. If the `TERSE` symbol is defined the body of the `terseReport` method will be obeyed when the method is called. If the `VERBOSE` symbol is defined the body of the `verboseReport` method will be obeyed. The body of the `reportHeader` method will be obeyed if either the TERSE or the VERBOSE symbols are defined because two attributes are combined before that method definition.

LISTING 2-47 The conditional attribute

Click here to view code image

```
//#define TERSE
#define VERBOSE

using System;
using System.Diagnostics;

namespace LISTING_2_47_The_conditional_attribute
{
    class Program
    {
        [Conditional("VERBOSE"), Conditional("TERSE
        static void reportHeader()
        {
            Console.WriteLine("This is the header f
        }

        [Conditional("VERBOSE")]
        static void verboseReport()
        {
            Console.WriteLine("This is output from
        }

        [Conditional("TERSE")]
        static void terseReport()
        {
            Console.WriteLine("This is output from
        }

        static void Main(string[] args)
        {
            reportHeader();
            terseReport();
            verboseReport();
            Console.ReadKey();
        }
    }
}
```

Note that the `Conditional` attribute controls whether or not the body of a given method is obeyed when the method is called, it does not control whether or not the method itself is passed to the compiler. The `Conditional` attribute does not perform the same function as conditional compilation in languages such as C and C++, it does not prevent code from being passed to the compiler, rather it controls whether code is executed when it runs. You can find out more about the use of pre-processor directives in Skill 3.4, "Debug an application."

Testing for attributes

A program can check that a given class has a particular attribute class attached to it by using the `IsDefined` method, which is a static member of the `Attribute` class. The `IsDefined` method accepts two parameters; the first is the type of the class being tested and the second type of the attribute class that the test is looking for.

Listing 2-48 shows how a program could check whether the `Person` class defined in Listing 2-46 is Serializable. Note that although the attribute is called `Serializable` when it is used in the source code, the name of the class that implements the attribute has the text `Attribute` appended to it, so that the attribute class we are looking for is called `SerializableAttribute`. The convention of adding the "Attribute" to the end of attribute classes is one that should be followed when creating our own attributes.

LISTING 2-48 Testing for an attribute

Click here to view code image

```
if (Attribute.IsDefined(typeof(Person), typeof(Seri
    Console.WriteLine("Person can be serialized");
```

Note that this test just tells us that a given class has an attribute of a particular type. Later you will discover how a program can read data which is stored in attribute instances.

Creating attribute classes

You can create your own attribute classes to help manage elements of your application. These classes can serve as markers in the same way as the `Serializable` attribute specifies that a class can be serialized, or you can store data in attribute instances to give information about the

program can change them as it runs, but these changes will be lost when the program ends.

Listing 2-49 shows an attribute class that can be used to tag items in our program with the name of the programmer that created it. We can add other details to the attribute, for example the name of the tester and the date the code was last updated.

LISTING 2-49 Creating an attribute class

Click here to view code image

```
class ProgrammerAttribute: Attribute
{
    private string programmerValue;

    public ProgrammerAttribute (string programmer)
    {
        programmerValue = programmer;
    }

    public string Programmer
    {
        get
        {
            return programmerValue;
        }
    }
}
```

Note that in Listing 2-49 the programmer name is stored as a read-only property of the attribute. We could have made the programmer name a writable property (added a `set` behavior to the `Programmer` property), but this is not sensible, because changes to the programmer name are not persisted when the program ends.

You can add the `Programmer` attribute to elements in your program in the same way as when adding the `Serializable` attribute, although in this case the attribute constructor must be called to set the programmer name.

The code here shows how this is done. Note that we've used named arguments in the call of the constructor to make it clear what is being set. When creating attributes, you should make good use of named and optional arguments.

Click here to view code image

```
 [ProgrammerAttribute(programmer:"Fred")]
class Person
{
    public string Name { get; set; }
}
```

Controlling the use of Attributes

The `ProgrammerAttribute` created earlier can be added to any item in my program, including member variables, methods, and properties. It can also be added to any type of object. When you create an attribute class, the proper practice is to add attribute usage information to the declaration of the attribute class, so that the compiler can make sure that the attribute is only used in meaningful situations. Perhaps you don't want to be able to assign `Programmer` attributes to the methods in a class. You only want to assign a Programmer attribute to the class itself.

This is done by adding an attribute to the declaration of the attribute class. The attribute is called `AttributeUseage` and this is set with a number of values that can control how the attribute is used.

The `AttributeUseage` settings in Listing 2-50 only allow the `Programmer` attribute to be applied to class declarations.

LISTING 2-50 Controlling attribute access

Click here to view code image

```
[AttributeUsage(AttributeTargets.Class)]
class ProgrammerAttribute: Attribute
{
…
}
```

If you try to use the `ProgrammerAttribute` on anything other than a class you will find that the compiler will generate errors. Note that this means that the compiler is performing reflection on our code as it compiles it, and you will find that Visual Studio does the same thing, in that invalid attempts to add attributes will be flagged as errors in the editor.

Click here to view code image

```
[ProgrammerAttribute(programmer:"Fred")]
class Person
{
    // This would cause a compilation error as we
    // are only allowed to apply this attribute to
```

You can set values of the `AttributeUsage` class to control whether children of a class can be given the attribute, specify which elements of your program can have the attribute assigned to them, identify specific types to be given the attribute and specify whether the given attribute can be applied multiple times to the same item. You can also use the or operator (`|`) to set multiple targets for a given attribute. The attribute class `FieldOrProp` below can be applied to properties or fields in a class.

Click here to view code image

```
 [AttributeUsage(AttributeTargets.Property | Attrib
class FieldOrProp: Attribute
{
…
}
```

Read attributes

You know how to add attributes to objects, so now you need to know how the program can read the attributes back. You saw that the `Attribute` class provides a static method called `IsDefined` that be used to determine if a class has a particular attribute defined. The `Attribute` class also provides a method called `GetCustomAttribute` to get an attribute from a particular type.

Listing 2-51 shows how this attribute is used. It has the same parameters as `IsDefined` (the type of the class and the type of the attribute class) and returns a reference to the attribute. If the attribute is not defined on the class, `GetCustomAttribute` returns `null`. In the code below the `Attribute` returned is cast to a `ProgrammerAttribute` and the name of the programmer is printed.

LISTING 2-51 Read an attribute

Click here to view code image

```
Attribute a = Attribute.GetCustomAttribute( typeof(
typeof(ProgrammerAttribute));

ProgrammerAttribute p = (ProgrammerAttribute)a;

Console.WriteLine("Programmer: {0}", p.Programmer);
```

Using reflection

Searching for and reading attribute values is an example of a program performing *reflection*. You can think of this as the program reflecting on its own contents, rather in the same way as people like to sit and reflect on their place in the universe.

Reflection is sometimes called *introspection*. Testing systems can use reflection to search for objects that need testing and identify the testing that is required by attributes that have been set on these objects.

You can use reflection to allow an application to automatically identify "plug-in" components, so that the application can be made up of discrete elements that are connected together when the program runs. If faults are found in one component, or new components need to be added, this can be performed without changes to the rest of the system.

You should regard elements discovered and loaded by the use of reflection as the natural endpoint of the process of creating code as components that implement interfaces and communicate by means of events which they publish and subscribe to.

Using type information from an object

You can start considering reflection by looking at the `GetType` method. All objects in a C# program expose this method, which will return a reference to the type that defines the object. Listing 2-52 shows how `GetType` can be used to get the type of a `Person` instance.

LISTING 2-52 The GetType method

Click here to view code image

```
System.Type type;

Person p = new Person();
type = p.GetType();
Console.WriteLine("Person type: {0}", type.ToString
```

The `Type` of an object contains all the fields of an object, along with all the metadata describing the object. You can use methods and objects in the `System.Reflection` namespace to work with `Type` objects. Listing 2-53 shows how to extract information about all the fields in the `Person` type. It prints all the members of the Person.

LISTING 2-53 Investigating a type

Click here to view code image

```
namespace LISTING_2_53_Investigating_a_type
{
    class Person
    {
        public string Name { get; set; }
    }

    class Program
    {
        static void Main(string[] args)
        {
            System.Type type;

            Person p = new Person();
            type = p.GetType();

            foreach(MemberInfo member in type.GetMe
            {
                Console.WriteLine(member.ToString()
            }

            Console.ReadKey();
        }
    }
}
```

When you run the program, it prints out the following information about the Person class:

Click here to view code image

```
System.String get_Name()
Void set_Name(System.String)
System.String ToString()
Boolean Equals(System.Object)
Int32 GetHashCode()
System.Type GetType()
Void .ctor()
System.String Name
```

Note that the `Name` property has been implemented by the compiler as a pair of `get` and `set` methods (`set_Name` and `get_Name`), and the class contains all the methods that are exposed by an object, including `ToString` and, of course, the `GetType` method.

Calling a method on an object by using reflection

You can use the information provided by a type to create a call to a method in that type. The program in Listing 2-54 will set the name of a person by using the `set_Name` behavior of the `Name` property in the `Person` class. It does this by finding the `MethodInfo` for this method and then calling the `Invoke` method on this reference. The Invoke method is supplied with a reference to the `Person` that is the target of the method invocation and an array of object references which will be used as the arguments to that method call.

LISTING 2-54 Reflection method call

Click here to view code image

```
System.Type type;

Person p = new Person();
type = p.GetType();

MethodInfo setMethod = type.GetMethod("set_Name");
setMethod.Invoke(p, new object [] { "Fred" });

Console.WriteLine(p.Name);
```

This code would, of course, be much slower than just setting the Name property to the value "Fred," but it illustrates the flexibility provided by reflection. A program can now obtain a reference to an object, find out what behaviors that object exposes, and then make use of the behaviors that it needs.

Finding components in assemblies

You have seen how a program can dynamically locate members of a class but to implement plugins you need to be able to search the classes in an assembly and find components that implement particular interfaces. This behavior is the basis of the Managed Extensibility Framework (MEF). You can find out more about MEF here at https://docs.microsoft.com/en-us/dotnet/framework/mef/.

Listing 2-55 shows how we can search an assembly for a particular type of component class. It can be used in a banking application to find all the classes that implement the `IAccount` interface. It searches the currently executing assembly for types that implement `IAccount`. The `Assembly` type provides a method that will get the currently executing assembly. It is then possible to iterate through all the types in this assembly looking for ones that are not interfaces and implement the `IAccount` interface. The code uses the `IsAssignableFrom` method to decide whether or not a given type implements the `IAccount` interface.

Click here to view code image

```
Assembly thisAssembly = Assembly.GetExecutingAssemb

List<Type> AccountTypes = new List<Type>();

foreach ( Type t in thisAssembly.GetTypes() )
{
    if (t.IsInterface)
        continue;

    if(typeof(IAccount).IsAssignableFrom(t))
    {
        AccountTypes.Add(t);
    }
}
```

If you run the sample program in Listing 2-55 it will print the names of the `BabyAccount` and `BankAccount` types, because these are defined in the same assembly as the sample program. You can simplify the identification of the types by using a LINQ query as shown in Listing 2-56.

LISTING 2-56 LINQ components

Click here to view code image

```
var AccountTypes = from type in thisAssembly.GetTyp
                   where typeof(IAccount).IsAssign
                   select type;
```

It is possible to load an assembly from a file by using the `Assembly.Load` method. The statement below would load all the types in a file called BankTypes.dll. This means that at its start an application could search a particular folder for assembly files, load them and then search for classes that can be used in the application.

Click here to view code image

```
Assembly bankTypes = Assembly.Load("BankTypes.dll")
```

Generate code at runtime by using CodeDOM and Lambda expressions

Now that you know that an entire application can be represented by objects such as `Assembly` and `Type`, the next thing to discover is how to programmatically add your own elements to these objects so that your applications can create code at runtime. You can use this technology to automate the production of code. It is used in situations where you have to create objects that will have to interact with services providing a particular data schema or when you're automatically generating code in a design tool.

In this section we are going to consider two techniques for generating code at runtime. They are the Code Document Object Model (CodeDOM) and Lambda expressions.

CodeDOM

A document object model is a way of representing the structure of a particular type of document. The object contains collections of other objects that represent the contents of the document. There are document object models for XML, JSON and HTML documents, and there is also one that is used to represent the structure of a class. This is called a CodeDOM object.

A CodeDOM object can be parsed to create a source file or an executable assembly. The constructions that are used in a CodeDOM object represent the logical structure of the code to be implemented and are independent of the syntax of the high-level language that is used to create the document. In other words, you can create either Visual Basic .NET or C# source files from a given CodeDOM object and you can create CodeDOM objects using either language.

Listing 2-57 shows how a CodeDOM object is created. The outer level `CodeCompileUnit` instance is created first. This serves as a container for `CodeNamespace` objects that can be added to it. A `CodeNameSpace` can contain a number of `CodeTypeDeclarations` and a class is a kind of `CodeTypeDeclaration`. The class can contain a number of fields. In Listing 2-57 a single data field is created, but there are also types to represent methods, statements within methods and expressions that the statements can work with.

LISTING 2-57 CodeDOM object

Click here to view code image

```
CodeCompileUnit compileUnit = new CodeCompileUnit()

// Create a namespace to hold the types we are goin
CodeNamespace personnelNameSpace = new CodeNamespac

// Import the system namespace
personnelNameSpace.Imports.Add(new CodeNamespaceImp
```

```
personClass.TypeAttributes = System.Reflection.Type

// Add the Person class to personnelNamespace
personnelNameSpace.Types.Add(personClass);

// Create a field to hold the name of a person
CodeMemberField nameField = new CodeMemberField("St
nameField.Attributes = MemberAttributes.Private;

// Add the name field to the Person class
personClass.Members.Add(nameField);

// Add the namespace to the document
compileUnit.Namespaces.Add(personnelNameSpace);
```

Once the CodeDOM object has been created you can create a CodeDomProvider to parse the code document and produce the program code from it. The code here shows how this works. It sends the program code to a string and then displays the string.

Click here to view code image

```
// Now we need to send our document somewhere
// Create a provider to parse the document
CodeDomProvider provider = CodeDomProvider.CreatePr
// Give the provider somewhere to send the parsed o
StringWriter s = new StringWriter();
// Set some options for the parse - we can use the
CodeGeneratorOptions options = new CodeGeneratorOpt

// Generate the C# source from the CodeDOM
provider.GenerateCodeFromCompileUnit(compileUnit, s
// Close the output stream
s.Close();

// Print the C# output
Console.WriteLine(s.ToString());
```

The output from this program can be seen here:

Click here to view code image

```
//------------------------------------------------------
// <auto-generated>
//     This code was generated by a tool.
//     Runtime Version:4.0.30319.42000
//
//     Changes to this file may cause incorrect beh
//     the code is regenerated.
// </auto-generated>
//------------------------------------------------------

namespace Personnel {
    using System;


    public class Person {

        private String name;
    }
}
```

There are a range of types that can be created and added to a document to allow you to programmatically create enumerated types expressions, method calls, properties and all the elements of a complete program. Note that you would normally create such a document model on the basis of some data structure that you were parsing.

Lambda expression trees

You saw lambda expressions in the context of event handlers in Skill 1.4, "Create and implement events and callbacks." A lambda expression is a way of expressing a data processing action (a value goes in and a result comes out). You can express a single action in a program by the use of a single lambda expression. More complex actions can be expressed in expression trees. If you think about it, the structure of a CodeDOM object is very like a tree, in that the root object contains elements that branch out from it. The elements in the root object can also contain other elements, leading to a tree like structure that describes the elements in the program that is being created. Expression trees are widely used in C#, particularly in the context of LINQ. The code that generates the result of a LINQ query will be created as an expression tree.

A lambda expression tree has as its base a lamba expression. Listing 2-58 shows how to create an expression tree that describes a lambda expression that evaluates the square of the incoming value. This code makes use of the `FUNC` delegate seen in Skill 1-4, "Create and implement events and callbacks." There are a number of built-in types for use with delegates, but the FUNC delegates allow us to define a delegate that accepts a number of inputs and returns a single result.

LISTING 2-58 Lambda expression tree

Click here to view code image

```
using System;
using System.Linq.Expressions;

namespace LISTING_2_58_Lambda_expression_tree
{
    class Program
    {
        static void Main(string[] args)
        {
            // build the expression tree:
            // Expression<Func<int,int>> square = n

            // The parameter for the expression is
            ParameterExpression numParam = Expressi

            // The opertion to be performed is to s
            BinaryExpression squareOperation = Expr

            // This creates an expression tree that
            Expression<Func<int, int>> square =
                Expression.Lambda<Func<int, int>>(
                    squareOperation,
                    new ParameterExpression[] { num

            // Compile the tree to make an executab
            Func<int, int> doSquare = square.Compil

            // Call the delegate
            Console.WriteLine("Square of 2: {0}", d

            Console.ReadKey();
        }
    }
}
```

The `System.Linq.Expressions` namespace contains a range of other types that can be used to represent other code elements in lambda expressions, including conditional operation, loops and collections.

Modifying an expression tree

An expression tree is immutable, which means that the elements in the expression cannot be changed once the expression has been created. To modify an expression tree you must make a copy of the tree which contains the modified behaviors.

Listing 2-59 shows how this is done using a class called `ExpressionVisitor`, which parses and copies each element of the expression tree. By extending the `ExpressionVisitor` base class and overriding different visit methods that it provides, you can create a class that will perform modification of the particular elements you are interested in. The `MultiplyToAdd` class will change any multiply operations in an expression to add.

LISTING 2-59 Modifying an expression tree

Click here to view code image

```
public class MultiplyToAdd : ExpressionVisitor
{
    public Expression Modify(Expression expression)
    {
        return Visit(expression);
    }

    protected override Expression VisitBinary(Binar
    {
        if (b.NodeType == ExpressionType.Multiply)
        {
            Expression left = this.Visit(b.Left);
            Expression right = this.Visit(b.Right);

            // Make this binary expression an Add r
            return Expression.Add(left, right);
        }

        return base.VisitBinary(b);
    }
}
```

You can use this class to modify the expression created in Listing 2-58 and create a new method that doubles any parameter given to it. The code here shows how this is done.

Click here to view code image

```
MultiplyToAdd m = new MultiplyToAdd();

Expression<Func<int, int>> addExpression = (Express
Func<int, int> doAdd = addExpression.Compile();

Console.WriteLine("Double of 4: {0}", doAdd(4));
```

The System.Reflection namespace contains a lot of useful types. You've seen some of these in this section, but here we're going to explicitly consider the function of some of them.

Assembly

An assembly is the output produced when a .NET project is compiled. The assembly type represents the contents of an assembly, which can be the currently executing assembly or one that is loaded from a file.

The Assembly class provides a way that programs can use reflection on the contents of the assembly that it represents. It provides methods and properties to establish and manage the version of the assembly, any dependencies that the assembly has on other files, and the definition of any types that are declared in the assembly. We used an assembly instance in Listing 2-55 "Finding components" when we loaded the current assembly and then searched through all of the types defined in that assembly to identify the ones that implement the `IAccount` interface. Listing 2-60 displays information about an assembly including the modules defined the assembly, the types in the modules and the content of each type.

LISTING 2-60 Assembly object

Click here to view code image

```
using System;
using System.Reflection;

namespace LISTING_2_60_Assembly_object
{
    class Program
    {
        static void Main(string[] args)
        {
            Assembly assembly = Assembly.GetExecuti

            Console.WriteLine("Full name: {0}", ass
            AssemblyName name = assembly.GetName();
            Console.WriteLine("Major Version: {0}",
            Console.WriteLine("Minor Version: {0}",
            Console.WriteLine("In global assembly c
            assembly.GlobalAssemblyCache);
            foreach (Module assemblyModule in assemb
            {
                Console.WriteLine("  Module: {0}",
                foreach (Type moduleType in assembly
                {
                    Console.WriteLine("     Type: {
                    foreach (MemberInfo member in m
                    {
                        Console.WriteLine("
                    }
                }
            }

            Console.ReadKey();
        }
    }
}
```

PropertyInfo

A C# property provides a quick way of providing get and set behaviors for a data variable in a type. The `PropertyInfo` class provides details of a property, including the `MethodInfo` information for the get and set behaviors if they are present. Listing 2-61 enumerates the properties in a type and prints information about each one.

LISTING 2-61 Property info

Click here to view code image

```
using System;
using System.Reflection;

namespace LISTING_2_61_Property_info
{
    public class Person
    {
        public String Name { get; set; }

        public String Age { get; }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Type type = typeof(Person);

            foreach (PropertyInfo p in type.GetProp
            {
                Console.WriteLine("Property name: {
                if(p.CanRead)
                {
                    Console.WriteLine("Can read");
                    Console.WriteLine("Set method:
```

```
                    Console.WriteLine("Can write");
                    Console.WriteLine("Set method:
                }
            }

            Console.ReadKey();
        }
    }
}
```

MethodInfo

The MethodInfo class holds data about a method in a type. This includes the signature of the method, the return type of the method, details of method parameters and even the byte code that forms the body of the method. The program in Listing 2-62 will work through a method and display this information from all the methods in a class. This code also makes use of the `Invoke` method to invoke a method from its method information and `MethodInvoke` to invoke a method from a class.

LISTING 2-62 Method reflection

Click here to view code image

```
using System;
using System.Reflection;

namespace LISTING_2_62_Method_reflection
{
    public class Calculator
    {
        public int AddInt(int v1, int v2)
        {
            return v1 + v2;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Get the type informa
            Type type = typeof(Calculator);

            Console.WriteLine("Get the method info
            MethodInfo AddIntMethodInfo = type.GetM

            Console.WriteLine("Get the IL instructi
            MethodBody AddIntMethodBody = AddIntMetl

            // Print the IL instructions.
            foreach (byte b in AddIntMethodBody.Get
            {
                Console.Write(" {0:X}", b);
            }
            Console.WriteLine();

            Console.WriteLine("Create Calculator in
            Calculator calc = new Calculator();

            Console.WriteLine("Create parameter arr
            object[] inputs = new object[] { 1, 2 }

            Console.WriteLine("Call Invoke on the m
            Console.WriteLine("Cast the result to a
            int result = (int) AddIntMethodInfo.Inv
            Console.WriteLine("Result of: {0}", res

            Console.WriteLine("Call InvokeMember on
            result = (int) type.InvokeMember("AddIn
                        BindingFlags.InvokeMethod |
                        BindingFlags.Instance | Bin
                        null,
                        calc,
                        inputs);
            Console.WriteLine("Result of: {0}", res

            Console.ReadKey();
        }
    }
}
```

Note that to invoke a method you must provide an array of parameters for the method to work on, and the invoke methods return an object reference that must be cast to the actual type of the method.

Type

You've used the `Type` type (as it were) in many places in your programs that use reflection. A type instance describes the contents of a C# type, including a collection holding all the methods, another containing all the class variables, another containing properties, and so on. It also contains a collection of attribute class instances associated with this type and details of the Base type from which the type is derived. The `GetType` method can be called on any instance to obtain a reference to the type for that object, and the `typeof` method can be used on any type to obtain the

in Listing 2-62, when it is used to obtain the type object for the `Calculator` type.

SKILL 2.6: MANAGE THE OBJECT LIFE CYCLE

In this section you are going to focus on the way that your applications will create and destroy objects as they run. The good news is that the .NET framework that underpins C# programs provides a *managed environment* for our programs that perform memory management in the form of a garbage collection process that will remove unwanted objects without us having to do anything.

When writing programs in unmanaged languages, such as C++, you need to include code to create and dispose of any objects that programs use. If this is not done properly one of two things will happen. A program will try to use a memory object that has been deleted which will cause the program to crash, or a program will fail to dispose of memory correctly, leading to a "memory leak," which will eventually cause the program to run out of available memory.

You might think that the fact that C# is based on the .NET framework, which provides automatic memory management, would make this a very short skill; but it turns out that our programs must also deal with "unmanaged" resources. For example, a program might create an object that contains a handle to a file or a database connection that connect it to a particular resource. The program must make sure that when this object is destroyed, any resources connected to the object must be released in a managed way. C# provides a finalization process that allows code in an object to get control as it is being removed from memory. C# also allows an object to implement an `IDisposable` interface, which you have already seen in Skill 2.4. In this section you will discover how to use these two features together to ensure that resources are always released correctly when objects are removed from memory.

This section covers how to:

- Manage unmanaged resources
- Implement IDisposable, including interaction with finalization
- Manage IDisposable by using the Using statement
- Manage finalization and garbage collection

Garbage collection in .NET

The precise way that memory is managed in a .NET application has a significant bearing on how to create objects that manage resources correctly. A full description of garbage collection technology would take the rest of this text, so this is a summary. Before we talk about garbage collection, let's first consider the situation in which it needs to happen.

Creating garbage

Consider the following two C# statements. The first statement declares a `Person` reference called `p` and makes the reference refer to a new `Person` instance. The second statement assigns the reference `p` to a new `Person` instance. These two statements will cause work for the garbage collector. The first `Person` object that was created can play no further part in the program as there is no way of accessing it.

Click here to view code image

```
Person p = new Person();
p = new Person();
```

You get a similar situation when a reference variable goes out of scope. This block of code that follows also creates work for the garbage collector. When the program exits from the block the `Person` object that was created it is no longer accessible because the reference `p` no longer exists. However, the person object will still be occupying memory.

Click here to view code image

```
{
    Person p = new Person();
}
```

A process has a particular amount allocated to it. Garbage collection only occurs when the amount of memory available for new objects falls below a threshold. Listing 2-63 is a program that deliberately creates a large number of inaccessible objects. I've adjusted the speed of the program and the size of memory allocated to produce a rate of memory use that causes the garbage collector to trigger on my machine.

LISTING 2-63 Garbage collection

Click here to view code image

```
namespace LISTING_2_63_Garbage_collection
{
    class Person
    {
        long[] personArray = new long[1000000];
    }
```

```
        static void Main(string[] args)
        {
            for ( long i=0;i<100000000000;i++)
            {
                Person p = new Person();
                System.Threading.Thread.Sleep(3);
            }
        }
    }
}
```

The Visual Studio runtime environment provides a display of memory use that will show the garbage collection process in action. Figure 2-8 below shows the display produced when the program runs.

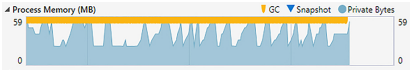

FIGURE 2-8 Memory usage and the garbage collector

If you look at the graph in Figure 2-8 you will see that the memory usage of the program varies dramatically over time. The rate at which the garbage is collected will change, depending on the loading on the host computer.

Value types and garbage collection

The storage graph in Figure 2-8 shows the size of the *heap*. The heap is the area of memory where an application stores objects that are referred to by reference. The contents of value types are stored on the stack. The stack automatically grows and contracts as programs run. Upon entry to a new block the .NET runtime will allocate space on the stack for values that are declared local to that block. When the program leaves the block the .NET runtime will automatically contract the stack space, which removes the memory allocated for those variables. The following block will not make any work for the garbage collector as the value 99 will be stored in the local stack frame.

```
{
    int i=99;
}
```

The garbage collector

When an application runs low on memory the garbage collector will search the heap for any objects that are no longer required and remove them. The .NET runtime contains an index of all the objects that have been created since the program started, the job of the garbage collector is to decide which of them are still in use, remove them from memory, and then compact the remaining objects so that the area of free memory is a single large area, rather than a number of smaller, free areas.

The first phase of garbage collection is marking all of the objects that are in use by the program. The garbage collector starts by clearing a flag on each object in the heap. It then searches through all the variables in use in the program and follows all the references in these variables to the objects they refer to and sets the flag on that object. After this "mark" phase the garbage collector can now move on to the "compact" phase of the collection, where it works through memory removing all the objects that have not had their flag set.

The final phase of the garbage collection is the "compaction" of the heap. The objects still in use must be moved down memory so that the available space on the stack is in one large block, rather than a large number of holes where unused objects have been removed.

All managed threads are suspended while the garbage collector is running. This means that your application will stop responding to inputs while the garbage collection is performed. It is possible to invoke the garbage collector manually if there are points in your application when you know a large number of objects have been released.

The garbage collector attempts to determine which objects are long lived, and which are short lived (ephemeral). It does this by adding a generation counter to each object on the heap. Objects start at generation 0. If they survive a garbage collection the counter is advanced to generation 1. Surviving a second garbage collection promotes an object to generation 2, the highest generation. The garbage collector will collect different generations, depending on circumstances. A "level 2" garbage collection will involve all objects. A "level 0" garbage collection will target newly created objects.

The garbage collector can run in "workstation" or "server" modes, depending on the role of the host system. There is also an option to run garbage collection concurrently on a separate thread. However, this increases the amount of memory used by the garbage collector and the loading on the host processor.

Reducing the need for garbage collection

Modern applications run in systems with large amounts of memory available. The garbage collection process has been optimized over the various generations of .NET and is now highly efficient and responsive. My strong advice on garbage collection is not to worry about it until you have a problem. There is no need to spend your time worrying about the memory usage of your application when your system will almost certainly have enough memory and processor performance. You can use the displays produced by Visual Studio to watch how your application uses

If you find that the garbage collection process is becoming intrusive you can force a garbage collection by calling the `Collect` method on the garbage collector as shown below.

```
GC.Collect();
```

The enforced garbage collection can be performed at points in your application where you know large objects have just been released, for example at the end of a transaction or upon exit from a large and complex user interface dialog. However, under normal circumstances I would strongly advise you to let the garbage collector look after itself. It is rather good at doing that.

Manage unmanaged resources

The .NET framework will take care of the creation and destruction of our objects, but we need to manage the resources that our objects use. For example, if an application creates a file handle and stores it in an object, when then object is destroyed the file handle will be lost. If the file connected to the handle is not closed properly before the object is destroyed, the file the handle is connected to will not be usable.

There are two mechanisms that we can use that allow us to get control at the point an object is being destroyed and tidy up any resources that the object may be using. They are finalization and disposable.

Object finalization using a finalizer method

The finalization of an object is triggered by the garbage collection process. An object can contain a finalizer method that is invoked by the garbage collector in advance of that object being removed from memory. This method gets control and can release any resources that are being used by that object. The finalizer method is given as a type less method with the name of the class pre-ceded by a tilde (~) character. Listing 2-64 shows how to give the `Person` class a finalizer method. The finalizer method just prints a message, but in an application you would use this method to release any resources that the `Person` object had allocated.

LISTING 2-64 The finalizer

Click here to view code image

```
public class Person
{
    long[] personArray = new long[1000000];

    ~Person()
    {
        // This is where the person would be finali
        Console.WriteLine("Finalizer called");
    }
}
```

Problems with finalization

When the garbage collector is about to remove an object, it checks to see if the object contains a finalizer method. If there is a finalizer method present, the garbage collector adds the object to a queue of objects waiting to be finalized. Once all of these objects have been identified, the garbage collector starts a thread to execute all the finalizer methods and waits for the thread to complete. Once the finalization methods are complete the garbage collector performs another garbage collection to remove the finalized objects. There are no guarantees as to when the finalizer thread will run. Objects waiting to be finalized will remain in memory until all of the finalizer methods have completed and the garbage collector has made another garbage collection pass to remove them.

A slow-running finalizer can seriously impair the garbage collection process. The destructor shown here contains a delay that will cause the program in Listing 2-64 to run out of memory.

Click here to view code image

```
~Person()
{
    // This is where the person would be fi
    Console.WriteLine("Finalizer called");

    // This will break the garbage collecti
    // as it slows it down so that it can't
    // faster than objects are being create
    System.Threading.Thread.Sleep(100);
}
```

Another problem with finalization is that there is no guarantee that the finalizer method will ever be called. If the program never runs short of memory, it might not need to initiate garbage collection. This means that an object waiting for deletion may remain in memory until the program completes.

You may have noticed that I'm not a fan of finalization. If you're a C++ programmer, who is used to writing destructors (the C++ equivalent of finalizers) to release memory when an object is deleted you might think that you should add finalizers to C# classes when you create them. However, I strongly suggest that you use the `Dispose` mechanism instead, as part of a planned approach to resource management, which is

Implement IDisposable, including interaction with finalization

We have already discussed the use of `IDisposable` and `Dispose` in the section describing `IDisposable` in Skill 2.4, "Create and implement a class hierarchy." An object can implement the `IDisposable` interface, which means it must provide a `Dispose` method that can be called within the application to request that the object to release any resources that it has allocated. Note that the `Dispose` method does not cause the object to be deleted from memory, nor does it mark the object for deletion by the garbage collector in any way. Only objects that have no references to them are deleted. Once `Dispose` has been called on an object, that object can no longer be used in an application.

Objects that implement `IDisposable` can be used in conjunction with the `using` statement, which will provide an automatic call of the `Dispose` method when execution leaves the block that follows the using statement.

The `Dispose` method is called by the application when an object is required to release all the resources that it is using. This is a significant improvement on a finalizer, in that your application can control exactly when this happens.

Using IDisposable and finalization on the same object

If you want to create an object that contains both a finalizer and a `Dispose` method you need to exercise some care to avoid the object trying to release the same resource more than once, and perhaps failing as a result of this. The `SuppressFinalize` method can be used to identify an object, which will not be finalized when the object is deleted. This should be used by the Dispose method in a class to prevent instances being disposed more than once. A *dispose pattern* can be used to allow an object to manage its disposal.

Listing 2-65 shows how an object can implement this pattern. It makes use of a helper method, `Dispose(bool disposing)`, which accepts a flag value that indicates whether the method is being called from a call of `Dispose` or from the finalizer. If the helper method is being called from a call of `Dispose` it will free off managed resources as well as unmanaged ones when it runs. The pattern also uses a class flag variable, `disposed`, which is set when dispose has been called and prevents multiple attempts to dispose items.

LISTING 2-65 The dispose pattern

Click here to view code image

```
using System;

namespace LISTING_2_65_The_dispose_pattern
{
    class ResourceHolder : IDisposable
    {

        // Flag to indicate when the object has bee
        // disposed
        bool disposed = false;

        public void Dispose()
        {
            // Call dispose and tell it that
            // it is being called from a Dispose ca
            Dispose(true);
            GC.SuppressFinalize(this);
        }

        public virtual void Dispose(bool disposing)
        {
            // Give up if already disposed
            if (disposed)
                return;

            if(disposing)
            {
                // free any managed objects here
            }

            // Free any unmanaged objects here
        }

        ~ResourceHolder()
        {
            // Dispose only of unmanaged objects
            Dispose(false);
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            ResourceHolder r = new ResourceHolder()

            r.Dispose();
        }
    }
}
```

us/dotnet/standard/garbage-collection/implementing-dispose.

Manage IDisposable by using the Using statement

As seen in the `IDisposable` item in Skill 2-4, "Create and implement a class hierarchy," the using statement allows you to ensure the calling of `Dispose` on objects that your program uses when they are no longer required. Listing 2-45 provides an example of how this works.

Note that the `using` statement ensures that `Dispose` is called on an object in the event of exceptions being thrown. If you don't use the `using` statement to manage calls of `Dispose` in your objects, make sure that your application calls `Dispose` appropriately. The dispose pattern above results in a disposal behavior that is tolerant of multiple calls of `Dispose`.

Manage finalization and garbage collection

In this section we can bring together some of the items we learned about garbage collection and consider how to manage the finalization and garbage collection process.

Invoking a garbage collection

You have seen that an application can force a garbage collection to take place by calling the Collect method. After the collection has been performed, a program can then be made to wait until all the finalizer methods have completed:

Click here to view code image

```
GC.Collect();
GC.WaitForPendingFinalizers();
```

Overloads of the `Collect` method allow you to specify which generation of objects to garbage collect and set other garbage collection options.

Managing finalization

You have seen that the garbage collector can be ordered to ignore the finalizer on an object. The statement below prevents finalization from being called on the object referred to by `p`.

Click here to view code image

```
GC.SuppressFinalize(p);
```

To later re-enable finalization you can use the `ReRegisterforFinalize` method:

Click here to view code image

```
GC.ReRegisterForFinalize(p);
```

SKILL 2.7: MANIPULATE STRINGS

As programmers we spend a lot of time manipulating text in our programs. The C# string type has a range of powerful features to make string manipulation easier. In this section we are going to explore these features and discover when they are useful.

This section covers how to:

- Manipulate strings by using the StringBuilder, StringWriter, and StringReader classes
- Search strings
- Enumerate string methods
- Format strings
- Use string interpolation

The string type

The C# type `string` (with a lower-case s) is mapped onto the .NET type `String` (with an upper-case S). The string type can hold text. In fact, it can hold a very large amount of text. It is perfectly feasible to hold a complete document as a single string. The theoretical limit on the maximum size of a string is around 2GB, but the practical one will be lower than that.

Immutable strings

Strings in C# are managed by reference, but string values are *immutable*, which means that the contents of a string can't be changed once the string has been created. If you want to change the contents of a string you have to make a new string. All string editing functions return a new string with the edited content. The string being edited is never changed. This ensures that strings are thread safe. There is no need to worry about code on another thread changing the contents of a string that a given thread is using.

The downside of this approach is that if you perform a lot of string editing you will end up creating a lot of string objects. However, the `StringBuilder` type, which we will see soon, provides a mutable string object that a program can use to assemble a string.

When a program is compiled the compiler uses a process called *string interning* to improve the efficiency of string storage. Consider the following C# statements. They set two string variables, s1 and s2, to the text "hello." The compiler will save program memory by making both s1 and s2 refer to the same string object with the content "hello."

LISTING 2-66 String intern

Click here to view code image

```
string s1 = "hello";
string s2 = "hello";
```

The process of string interning makes string comparison quicker. If two string references are equal it means that they both contain the same string, there is no need to compare the text in the two strings to see if it is the same. Note that if the string references are different; this doesn't mean that the strings are definitely not different. It means that the program must compare each of the characters in the strings to determine if they are the same.

String interning only happens when the program is compiled. The following statements would make a make a new string with the content "hello" when the program runs, however this string would be a different object from the string created for `s1` and `s2` above.

Click here to view code image

```
string h1 = "he";
string h2 = "llo";
string s3 = h1 + h2;
```

String interning doesn't happen when a program is running because every time a new string is created the program has to search through the list of interned strings to see if that string was already present. This slows the program down. However, you can force a given string to be interned at run time by using the `Intern` method provided by the string type. The statement here makes the string s3 refer to an interned version of the string.

Click here to view code image

```
s3 = string.Intern(s3);
```

Note that I have the same opinion of string interning as I have about many other issues relating to performance. You should only do this if you discover a need to speed up the program. Under normal circumstances the performance of string storage in a C# program is extremely good.

Manipulate strings by using the StringBuilder, StringWriter, and StringReader classes

The string type is accompanied in .NET by some other types that work with strings. Three of these types are `StringBuilder`, `StringWriter`, and `StringReader`. Let's consider each of these in turn.

StringBuilder

The `StringBuilder` type is very useful when we are writing programs that build up strings. It can improve the speed of a program while at the same time making less work for the garbage collector. Consider the C# statements:

Click here to view code image

```
string firstName = "Rob";
string secondName = "Miles";

string fullName = firstName + " " + secondName;
```

The string addition in the final statement will create the value of `fullName` by adding the three strings together, but it will create an intermediate string as it builds the result (`firstName + " "`). In the case of this program this has a negligible effect on performance, but if a program was performing a lot of string addition to build up a string of text this might result in a lot of work for the garbage collector.

The `StringBuilder` type provides methods that let a program treat a string as a mutable object. A `StringBuilder` is implemented by a character array. It provides methods that can be used to append strings to the `StringBuilder`, remove strings from the `StringBuilder` and a property, `Capacity`, which can be used to set the maximum number of characters that can be placed in a `StringBuilder` instance. Listing 2-67 shows how to use a `StringBuilder` to create a full name string without making any intermediate objects.

LISTING 2-67 Stringbuilder

Click here to view code image

```
StringBuilder fullNameBuilder = new StringBuilder()
fullNameBuilder.Append(firstName);
fullNameBuilder.Append(" ");
fullNameBuilder.Append(secondName);
Console.WriteLine(fullNameBuilder.ToString());
```

If you want to assemble strings out of other string you can also use format strings and string interpolation instead of string addition.

StringWriter

The `StringWriter` class is based on the `StringBuilder` class. It implements the `TextWriter` interface, so programs can send their output into a string. Listing 2-68 shows how to create a `StringWriter`, write some text to it, and then print the text out.

LISTING 2-68 StringWriter

Click here to view code image

```
using System;
using System.IO;

namespace LISTING_2_68_StringWriter
{
    class Program
    {
        static void Main(string[] args)
        {
            StringWriter writer = new StringWriter(

            writer.WriteLine("Hello World");

            writer.Close();

            Console.Write(writer.ToString());

            Console.ReadKey();
        }
    }
}
```

You have used the `StringWriter` class in a previous example; the program Listing 2-57 CodeDOM object, which creates some C# from a CodeDOM object, and sends the C# code to a `StringWriter` instance so that it can be printed on the screen.

StringReader

Instances of the `StringReader` class implement the `TextReader` interface. It is a convenient way of getting string input into a program that would like to read its input from a stream. The program in Listing 2-69 shows how `StringReader` is used. It creates a `StringReader` and then reads a string and an integer from it.

LISTING 2-69 StringReader

Click here to view code image

```
using System;
using System.IO;

namespace LISTING_2_69_StringReader
{
    class Program
    {
        static void Main(string[] args)
        {
            string dataString =
@"Rob Miles
21";
            StringReader dataStringReader = new Str

            string name = dataStringReader.ReadLine
            int age = int.Parse(dataStringReader.Re

            dataStringReader.Close();

            Console.WriteLine("Name: {0} Age: {1}",

            Console.ReadKey();
        }
    }
}
```

Search strings

The string type provides a range of methods that can be used to find the position of substrings inside a string. Note that, because strings are immutable it is not possible to change elements of the string once you have found the position of a substring (you can't replace the word "Rob" in a string with the word "Fred"), however, you can use the search methods to control the process of copying from one string to another. You can also use these methods to parse strings and search for particular words and character sequences. The sample code in Listing 2-70 contains all the demonstration code in this section.

Contains

The method `Contains` can be used to test of a string contains another string. It returns true if the source string contains the search string. The code here shows how `Contains` would be used. It prints that the string

Click here to view code image

```
string input = "      Rob Miles";

if(input.Contains("Rob"))
{
    Console.WriteLine("Input contains Rob");
}
```

StartsWith and EndsWith

The methods `StartsWith` and `EndsWith` can be used to test if a string starts or ends with a particular string. Note that if the string starts or ends with one or more *whitespace* characters these methods will not work. A whitespace character is a space, tab, linefeed, carriage-return, formfeed, vertical-tab or newline character. There are methods you can use to trim a string. `TrimStart` creates a new string with whitespace removed from the start, `TrimEnd` removes whitespace from the end of the string and `Trim` removes whitespace from both ends. The code below shows how this works. It will print that the name starts with Rob, because the input string was trimmed before the test.

Click here to view code image

```
string input = "      Rob Miles";

string trimmedString = input.TrimStart();

if(trimmedString.StartsWith("Rob"))
{
    Console.WriteLine("Starts with Rob");
}
```

IndexOf and SubString

The method `IndexOf` returns an integer which gives the position of the first occurrence of a character or string in a string. There is also a `LastIndexOf` method that will give the position of the last occurrence of a string. There are overloads for these methods that let you specify the start position for the search. You can use these position values with the `SubString` method to extract a particular substring from a string. The code below shows how this works. It extracts the name `Rob` from a string and prints it.

Click here to view code image

```
string input = "      Rob Miles";
int nameStart = input.IndexOf("Rob");
string name = input.Substring(nameStart, 3);
Console.Write(name);
```

Replace

The `Replace` method can be used to perform string editing, replacing an element of a source string. The code here shows how `Replace` can be used to convert an informal version of my name into a formal one.

Click here to view code image

```
string informalString = "Rob Miles";
string formalString = informalString.Replace("Rob",
Console.WriteLine(formalString);
```

Split

The `Split` method can be used to split a string into a number of substrings. The split action returns an array of strings, it is given one or more separator strings that will be used to split the string. The code here shows how this works.

Click here to view code image

```
string sentence = "The cat sat on the mat.";
string[] words = sentence.Split(' ');
foreach (string word in words)
{
    Console.WriteLine(word);
}
```

Note that if the source sentence held a number of consecutive spaces, each of these would be resolved into a separate line in the output.

String comparison and cultures

Strings in C# are made up of *Unicode* characters encoded into 16 bit values the using *UTF-16* encoding format. The Unicode standard provides a range of characters that can lead to culture specific behaviors when strings are being compared. For example, the character pair “ae” can be represented by the single character “æ” in a lot of American and European languages. This means that, from a language point of view, there are situations where the words “encyclopaedia” and “encyclopædia” should be regarded as equal.

You can request that the string comparison process takes this situation

You can request that strings be compared according to the culture of the currently executing thread (which is useful if we are creating a program for use in a particular region), compared according to the ordinal values of the character code of the characters (which is efficient but may cause ordering problems if the items contain characters such as “æ”), or compared according to an ordering scheme based on the English language, but not customized for a particular culture. The code here shows how these options are used. All of the conditions that are shown evaluate to true.

Click here to view code image

```
// Default comparison fails because the strings are
if (!"encyclopædia".Equals("encyclopaedia"))
    Console.WriteLine("Unicode encyclopaedias are n
// Set the curent culture for this thread to EN-US
Thread.CurrentThread.CurrentCulture = CultureInfo.C
// Using the current culture the strings are equal
if ("encyclopædia".Equals("encyclopaedia", StringCo
    Console.WriteLine("Culture comparison encyclopa
// We can use the IgnoreCase option to perform comp
if ("encyclopædia".Equals("ENCYCLOPAEDIA", StringCo
    Console.WriteLine("Case culture comparison ency
if (!"encyclopædia".Equals("ENCYCLOPAEDIA", StringC
    Console.WriteLine("Ordinal comparison encyclopa
```

Enumerate string methods

You can regard a string as an array of characters. A program can iterate through the characters in a string as it would any other collection. A string also provides a `Length` property that a program can use to determine the number of characters in a string. Listing 2-71 shows how this works.

LISTING 2-71 Enumerate string

Click here to view code image

```
foreach(char ch in "Hello world")
{
    Console.WriteLine(ch);
}
```

Format strings

You can use format strings to request that output be formatted in a particular way when it is printed. The sample code in Listing 2-72 shows how this works. It prints out an integer and a double precision value. The `WriteLine` method is given a string that contains placeholders which start and end with braces. Within the placeholder is first the number of the item in the `WriteLine` parameters to be printed at that placeholder, the number of characters the item should occupy (negative if the item is to be left justified), a colon (:) and then formatting information for the item. The formatting information for an integer string conversion can comprise the character ‘D’ meaning decimal string and ‘X’ meaning hexadecimal string. The formatting information for a floating-point number can comprise the character ‘N,’ which is followed by the number of decimal places to be printed.

LISTING 2-72 Format strings

Click here to view code image

```
int i = 99;
double pi = 3.141592654;

Console.WriteLine(" {0,-10:D}{0, -10:X}{1,5:N2}", i
```

This program prints the following output. Note that the value of pi is truncated to two decimal places, as requested in the format string, and that the value of i is printed twice, once in decimal and the second time in hexadecimal.

Click here to view code image

```
99         63          3.14
```

This works because the `int` and `double` types can accept formatting commands to specify the string they are to return. A program can give formatting commands to many .NET types. The `DateTime` structure provides a wide range of formatting commands. You can add behaviors to classes that you create to allow them to be given formatting commands in the same way. Any type that implements the `IFormattable` interface will contain a `ToString` method that can be used to request formatted conversion of values into a string.

Listing 2-73 shows how this is done. The `MusicTrack` class contains an implementation of a `ToString` method that overrides the `ToString` in the base class. However, the `MusicTrack` class also contains a `ToString` method that accepts two parameters. The first of these is a string that specifies a format for the conversion of the `MusicTrack` information into a string. The second parameter is format provided that the `MusicTrack` can use to determine the culture for the conversion.

description if the format is F. Note that there is also a G format option, this represents a common format that may be requested. The G format should also be used if the formatting information is missing. Note that if the format request is not recognized, the `ToString` method will throw an exception.

LISTING 2-73 Music track formatter

Click here to view code image

```
class MusicTrack : IFormattable
{
    string Artist { get; set; }
    string Title { get; set; }

    // ToString that implements the formatting beha
    public string ToString(string format, IFormatPr
    {
        // Select the default behavior if no format
        if (string.IsNullOrWhiteSpace(format))
            format = "G";

        switch (format)
        {
            case "A": return Artist;
            case "T": return Title;

            case "G": // default format
            case "F": return Artist + " " + Title;
            default:
                throw new FormatException("Format s
        }
    }

    // ToString that overrides the behavior in the
    public override string ToString()
    {
        return Artist + " " + Title;
    }

    public MusicTrack(string artist, string title)
    {
        Artist = artist;
        Title = title;
    }
}
```

This formatting behavior can now be used when the object is being printed as shown in the code here. The track information is printed three times, with different amounts of detail each time.

Click here to view code image

```
MusicTrack song = new MusicTrack(artist: "Rob Miles

Console.WriteLine("Track: {0:F}", song);
Console.WriteLine("Artist: {0:A}", song);
Console.WriteLine("Title: {0:T}", song);
```

The format provider in ToString

The second parameter to the `ToString` method in Listing 2-73 is an object that implements the `IFormatter` interface. This parameter can be used by the `ToString` method to determine any culture specific behaviors that may be required in the string conversion process. For example, you might add date of recording and price information to a music track, in which case the display of the date and price information could be customized for different regions.

By default (i.e. unless you specify otherwise) the `IFormatter` reference that is passed into the `ToString` call is the current culture. You saw the use of cultures earlier in the "String comparison and cultures" section.

You can create a culture description and explicitly pass it into a call of `ToString`. In the case of the `MusicTrack` implementation in Listing 2-73 this will have no effect, because the `ToString` method in `MusicTrack` class doesn't make use of the format provider, but the floating-point types implement a ToString method that does make use of culture information.

Listing 2-74 shows how this works. The bank balance value is stored in a double precision value and then displayed twice as a currency value using the format command 'C.' The first time it is displayed using the US culture information to control the output and the second time using the UK culture.

LISTING 2-74 Format provider

Click here to view code image

```
double bankBalance = 123.45;
CultureInfo usProvider = new CultureInfo("en-US");
Console.WriteLine("US balance: {0}", bankBalance.To
CultureInfo ukProvider = new CultureInfo("en-GB");
Console.WriteLine("UK balance: {0}", bankBalance.To
```

This program prints the following output. Note that the currency character is different.

Click here to view code image

```
US balance: $123.45
UK balance: £123.45
```

Use string interpolation

You have seen how to use format strings that incorporate placeholders in them. Each placeholder is enclosed in braces { and } and a placeholder relates to a particular value which is identified by its position in the arguments that follow the format string. The statements in Listing 2-75 show how this works. The `name` and `age` values are given after the format string.

LISTING 2-75 String interpolation

Click here to view code image

```
string name = "Rob";
int age = 21;
Console.WriteLine("Your name is {0} and your age is
```

String *interpolation* allows you to put the values to be converted directly into the string text. An interpolated string is identified by a leading dollar ($) sign at the start of the string literal. The statement below shows how this works. The WriteLine method now only has one parameter.

Click here to view code image

```
Console.WriteLine($"Your name is {name} and your ag
```

Note that when this program is compiled the compiler will convert the interpolated string into a format string, extract the values from the string, and build a call of `WriteLine` that looks exactly like the first call of `WriteLine`. If you use ildasm to look at the code produced by the compiler for the two calls of `WriteLine`, you will find that they are identical.

String interpolation can be used with `String.Format` method as shown:

Click here to view code image

```
Console.WriteLine(String.Format($"Your name is {nam
```

Using a FormattedString with string interpolation

The act of assigning a string interpolation literal in a program produces a result of type `FormattedString`. This provides a `ToString` method that accepts a `FormatProvider`. This is useful because it allows you to use interpolated strings with culture providers. The code below uses string interpolation to produce a bank balance value, which is formatted using the "en-US" (English United States) culture info.

Click here to view code image

```
double bankBalance = 123.45;
FormattableString balanceMessage = $"US balance : {
CultureInfo usProvider = new CultureInfo("en-US");
Console.WriteLine(balanceMessage.ToString(usProvide
```

THOUGHT EXPERIMENTS

In these thought experiments, demonstrate your skills and knowledge of the topics covered in this chapter. You can find the answers to these thought experiments in the next section.

1 Creating types

There comes a point in application construction where the data storage for the application must be designed. At this point you need to map data items onto C# data types. Each data item will be mapped onto an object that may be managed by value or reference. State information can be mapped onto enumerated types for which you can set a range of allowed values.

Here are some questions to consider:

1. You want to store information about the employees that work in my company. Is this a job for a value type or a reference type?
2. Can an object be referred to by more than one reference?
3. What happens when an object no longer has any references to it?
4. If you use a value type rather than a reference type, will your program still work correctly?
5. How can you make a data type immutable?

8. Do you need to add a constructor method for every object that you create?
9. Is there ever any point in creating objects that cannot be constructed?
10. What does it mean when a member of a type is made static?
11. Should you provide default values for all of the parameters to a method?
12. Does using a lot of overridden methods slow a program down?

2 Consuming types

Once you have established the types that your program will use, you will have to write code to manipulate the types. This frequently means that the program will have to convert values from one type to another, and sometimes a program will have to interact with external services which do not have type information that can be used by the C# compiler to perform static analysis to establish the correctness of an interaction.

Here are some questions to consider:

1. If every variable in a program was managed by reference, would you ever need to box a value?
2. Do you need to use a cast when your program is widening a type?
3. Does casting between values slow a program down?
4. Can an object have multiple type conversion methods?
5. Could you write a program that used entirely dynamic types?
6. Why do you need to interact with Component Object Model (COM) objects?

3 Enforce encapsulation

Encapsulation is a powerful technique. It allows developers to hide the implementation of an object and control precisely how code outside the object will interact with it. It also means that developers can change the internal design of an object without affecting any of the users of that object.

Here are some questions to consider:

1. Is it better to use a property rather than get and set methods for private data members in your classes?
2. Can structures contain private data members and properties?
3. Does using properties slow down the execution of my program?
4. Which provides a greater amount of access to a class member, protected or internal?
5. Are there any reasons not to use explicit implementation of methods in an object?

4 Create and implement a class hierarchy

The starting point for a class hierarchy should not be the design of the base class of the hierarchy, but the creation of an interface that expresses the behaviors of this base class. Using an interface from the start of your design process allows a great deal of flexibility in that objects can be regarded in terms of their abilities, not their particular type. You can still a class hierarchy to reduce the amount of code that you have to write, by creating child classes that inherit methods from their parents and only override the one that need to provide behaviors specific to the child.

Here are some questions to consider:

1. Should you design your interfaces after you have designed your application?
2. Can a C# interface contain anything other than method signatures?
3. Can a C# interface be extended?
4. Are all the methods defined in an interface public?
5. Does a class have to implement all the methods in an interface that it implements?
6. Can a C# interface contain a constructor method?
7. Is it sensible for an object to implement more than one interface?
8. Can only the base class of a hierarchy implement an interface?
9. Does making methods virtual slow a program down?
10. Does making methods virtual reduce the security of a program?
11. What is the difference between a C# interface and an abstract C# class?
12. Should every class that you create be part of a class hierarchy (either as a base class or a child class)?
13. Can a method be both overridden and overloaded?
14. Can you override a method in a structure type?
15. Can a structure implement an interface?

18. Does the yield keyword cause my program to stop?
19. Does calling the Dispose method on an object remove it from memory?
20. If you use IDispose do you have to call the Dispose method myself to dispose of objects.
21. If you use the IDispose interface to tidy up your objects, does this mean that they will be deleted more quickly by the garbage collector?

5 Find, execute, and create types at runtime by using reflection

This skill involves the use of code to work with code. You can improve your development process by adding attribute information to objects and behaviors in the program. You can create programs that dynamically configure their components by searching for code elements that have a particular set of behaviors as defined in an interface. You can also use code to create more code, automatically producing complete classes, which can be converted into program source or a compiled assembly. You can also create expression trees that can be converted into executable methods.

Here are some questions to consider:

1. Where are the values of attributes actually stored?
2. What happens if you change the value of a member of an attribute when the program is running?
3. Can an attribute class have attributes?
4. Where is the type information for a class stored?
5. What is the difference between typeof and GetType?
6. Can the GetType method be called on value types?
7. Can you change the contents of a type description returned by GetType?
8. Can you use reflection to obtain details of private items in objects?
9. Does code created by using a CodeDOM run slower than “normally” written code?
10. Is it possible to read code details from an assembly file?
11. Can you use CodeDOM to create a class that implements a particular interface?
12. Can you use reflection on the compiled output of a CodeDOM object?

6 Manage the object life cycle

The automatic garbage collection enjoyed by C# developers is built on carefully written and highly developed foundations. However, the non-deterministic nature of the garbage collection process (you can’t always be sure precisely when an object will be removed from memory) means that programmers must exercise care when working with objects that contain references to resources that are to be shared with other objects. The IDisposable interface provides a way that programmers can enforce a more managed approach to resource management.

Here are some questions to consider:

1. If you use value types rather than reference types, will this stop the garbage collector from being called?
2. Does creating and destroying arrays of value types make work for the garbage collector?
3. Does the garbage collector also collect garbage from the stack?
4. Can a finalizer be called before the Dispose method in an object?
5. Can the finalizer method in a class be overloaded?
6. Does a finalizer in a child class have to call the finalizer in a base class when it runs?
7. Is it possible for the Dispose method in an instance to be called more than once?
8. What happens if an object finalizer takes a long time to run?
9. What happens if an object finalizer creates a new reference to the object being finalized?
10. Should you design your applications from the start to use the minimum possible memory?

7 Manipulate strings

Successive versions of C# have added more and string manipulation abilities to the language and its libraries. It is very unlikely that you will find yourself in a situation where you have to write some code to perform string processing. I strongly advise you to check out the abilities of the string type and the libraries that work with it before you start to write custom code to work with strings.

Here are some questions to consider:

1. Can you hold the text of an entire book in a single string instance?

3. Are there any disadvantages to interning strings in a C# program?
4. Is the StringBuilder type based on the string type?
5. Will using StringBuilder always make your programs go faster?
6. Can you store binary data in a StringWriter?
7. Does the Trim method change the contents of a string?
8. What does it mean if an object implements the IFormattable interface?
9. What does a CultureInfo object actually contain?
10. Can you use string interpolation with strings of text that are created as the program runs?

THOUGHT EXPERIMENT ANSWERS

This section provides the solutions for the tasks included in the thought experiments.

1 Creating types

1. Essentially this boils down to the question of whether you want to store your employee records in a structure (value type) or a class (reference type). If you want to handle different types of employee such as part time, shift worker, and manager then you might find that a class hierarchy is useful, in which case a reference type is needed, because structures cannot be used as the basis of class hierarchies. Using reference types will make it much easier to index and sort your employee records. For example, you can have two employee lists, one ordered by employee name and another ordered by pay grade. Each list will contain references to employee objects. Structures are best suited to small value types, which you may want to make immutable. You might create a data type that holds the record of when an employee started with the company. This could contain their age at starting, the day they started and other information. This could be held in a structure as it is a small amount of data that you just want to store somewhere. It might also be worth making this item of data immutable, since there is no need to edit the information once it has been created. Using a structure (value type) to hold the data would be a good idea if the number of data items is very large.
2. Yes, a single object can have multiple references referring to it.
3. It might be that as a program runs, and references are assigned different values, then an object in memory no longer has any references to it from the program. At this point the object can be destroyed, since it can never be used again. The C# runtime environment contains a "garbage collection" process that checks for objects that have no references to them. These objects are removed when they are detected. For more details take a look at Skill 2-6, "Manage the object lifecycle."
4. It is possible to create any program entirely using reference types or value types. The behavior of assignments and the effects of changes to objects means that if your program treats a value type as a reference type (or vice versa), you will discover that it will not behave as you might expect.
5. A data type can be made immutable by providing no means of changing the variables held inside the type. The values of the data type should be set by the constructor and the type should only provide methods and properties that can be used to read these variables. The variables in the type must all be made private to the type.
6. Value types can be regarded as slightly more efficient to access than reference types. To access a reference type a program must follow the reference to the position in the heap where the object is stored. A value type can be accessed directly and an array of value types is stored in a single block of storage that contains all of the values. The performance difference will not be noticeable except in extreme circumstances where huge numbers of objects are being processed.
7. Space on the stack is allocated as a program runs. Each time a block is entered, an area of stack space is reserved for value type variables that are created when the method runs. When the block exits, all of the stack space is recovered and the value type variables deleted. Space on the heap is allocated when reference types are created. A reference type may have a lifetime that is longer than the block in which it is declared. For example, a method that creates a new Employee record that is a reference type will return the reference. If the Employee is created on the stack it will disappear when the method finishes. The heap has no particular structure, so there is no guarantee that successively created objects will be in adjacent locations on the heap. Given that C# programs run in a managed environment, they are not able to determine the precise position of objects in memory.
8. Most types will require at least one constructor, so that your application can ensure that the object has a valid initial state and can feed information into the type via the constructor to set it up. The construction of the objects in your system is something that should be planned at the time the system is designed.
9. A class that only contains static members does not need to be constructed. You find static classes like this that contain service methods.

smallest) and default values (default age for a customer) as these only need to be stored for an entire class, not for each object.

11. A program can provide default values for method parameters. This can be useful, because it can simplify the use of the methods. It also, however, provides a form of information hiding, in that users of the method may not know what the default values are. I restrict default values to parameters that specify "optional" rather than "core" behaviors.

12. A method that is marked as virtual can be overridden in a class hierarchy. When an overridden method is called on a reference the runtime system must search up the class hierarchy for the version of the method to use. The deeper the class hierarchy the longer it will take for this search to find the method. This means that extensive use of overridden methods in a class hierarchy may have an impact on program performance, but it is unlikely to be noticeable unless the class hierarchy is very deep. I usually try to avoid a hierarchy more than five or so levels deep, but this is as much in respect of organizational complexity as it is program performance.

2 Consuming types

1. Boxing is the name given to the action of converting a value type (which is most likely stored on a local stack) into a reference type (which will be stored on the heap). It is performed when we need to work with a value as an object, perhaps to allow the value to be inserted in a data structure that is managed by reference. Some programming languages have no value types at all and manage everything by reference. This simplifies some aspects of the way that programs will execute, but it does this at the expense of performance. A program will spend a lot of its time just manipulating values, and the use of reference types will significantly slow down this process. You could write a C# program that uses nothing but the interface types that provide the object implementations of value types, but this would run a lot more slowly than one that used value types.

2. Widening is the situation when a value is being moved from one type to another with a wide range of possible values, for example from int to double. In this situation there is no possibility of data being lost, and so casting is not required. However, you can add it for clarity if you wish.

3. Casting between value types may be something that the Central Processor Unit (cpu) of the computer can perform very quickly. However, if the casting process involves calling a type conversion method this will take longer.

4. A type conversion method is added to an object to generate a value to be used when an object is cast into a particular type. Each type conversion method is assigned a "target" type, and so it would be possible for an object to have a type conversion that produces an integer result and another that produces a floating-point result. However, I would suggest that in these situations it may be clearer to make use of properties to return a value from a type, rather than using casting and type conversion methods, which are not something that all programmers will have come across.

5. You can write a program that uses entirely dynamic types. It should always compile, since the compiler will not perform any static error checking. However, it would probably fail quite frequently too, as mistakes that would normally produce compilation errors would not be spotted and would manifest themselves as run time errors. Using dynamic types also makes development more difficult in another way. Dynamic types acquire their types at runtime and so the Intellisense feature of Visual Studio cannot provide any useful information about a variable of dynamic type as it doesn't know what that type of information the variable is holding.

6. Many applications, for example the Microsoft Office Suite, provide automation features that are accessed via COM interfaces. This is also true of some of the lower level elements of the Windows operating system. It is unlikely that you will have to interact with lots of COM objects during your programming career, but it is important to be able to work with them, and the dynamic type removes a lot of effort from this.

3 Enforce encapsulation

1. There are two ways that an object can encapsulate data values. It can provide properties, or it can provide getter and setter methods (sometimes called accessors and mutators respectively). Properties are very easy to use from the point of view of code that has to interact with the encapsulated values, they be assigned and read in the same way as public data values. From a performance perspective there is no difference between properties and get/set methods. We have seen that the compiler actually generates get and set methods when properties are compiled. However, get and set methods are slightly more flexible, in that a set method can perform validation on an input value and return a result that indicates whether an input value has been set correctly. The set behavior of a property could also perform validation, but the set behavior cannot return a status value.

2. There is no reason why a structure should not contain private members and properties. Although most people consider encapsulation as something that is performed on classes, structures can be encapsulated objects. An extreme form of encapsulation is the creation of immutable objects. These contain data that is set when they are created and cannot be modified. An immutable

3. Using properties will slightly slow down a program, in that a property behavior will take longer to complete than just getting a value directly from an object. However, the property access code will be *inlined* in that rather than calling a function to perform the property behavior the compiler will just output the code required. This makes a program slightly larger (there may be multiple copies of the code that uses the property) but it also means that accessing properties in objects is almost as fast as accessing the values directly.

4. A member of a class with the access modifier protected is visible in that class and in any classes that are children of that class. A member of a class with the access modifier internal is visible to any class that is declared in the same assembly. This means that an internal item has the potential to be seen by more classes in an application.

5. The only reason that an object would not use an explicit implementation of interface methods is if the programmer wants to "share" some methods between several interfaces. This is not the kind of thing that sensible programmers should do.

4 Create and implement a class hierarchy

1. The behaviors that are required by an object are best expressed in the form of an interface. You might create a proof of concept application and then create interfaces based on the behaviors of the objects in that application, but you should make sure that the interfaces are established right at the start of the development process.

2. A C# interface can also contain the definition of properties that should be created any class that implements the interface.

3. Yes. A C# interface can be extended. We actually did this when we extended the IComparable interface. Take a look at Listing 2-40 for how this was done.

4. Yes. All the methods in a C# interface must be made public. However, an interface can be made internal, which means that it is only visible within a given assembly.

5. No. A class must contain implementations of all the methods in an interface if we want to create instances of that class. A class in a class hierarchy could implement some methods in an interface and a child method of the class could implement the rest. In this situation it would not be possible to create an instance of the base class as this does not contain implementations of all the interfaces. Note that this design approach is not something I would encourage.

6. No. Interfaces do not contain constructors. If you want to go down the route of having interface methods that construct instances you will have to put factory methods (methods which return references to instances that they have created) in the interface.

7. It can be. For example an `Employee` record that must be printed could expose an `IEmployee` interface and an `IPrintable` interface.

8. No. Interfaces can be implemented at any level of a class hierarchy. If the base class implements the interface, this has the effect of forcing any children of the base class to implement the interfaces too, but child classes can implement further interfaces. However, this is not something that I would encourage.

9. Yes. Very slightly. A non-virtual method can be called directly, since there will only ever be one copy of that method. When a virtual method is called the .NET runtime must search for the most appropriate method to call, starting with the type of the object and then looking up the class hierarchy to find the method in base classes.

10. Yes. A method that is virtual can be overridden and replaced with one that has a different behavior in the child. You should only do this for methods for which you have a definite need for the ability to override their behavior.

11. An interface specifies a set of behaviors that an object must provide so that it can be referred to by a reference of the interface type. This provides a great deal of flexibility, for example a list of interface references could contain any type of object, as long as that type implemented the interface. By contrast, an abstract type is more restrictive. It contains a set of behaviors that an object must provide, but the object can only then be regarded in terms of the abstract type, not in any other way.

12. No. If there is no need for an object to be extended then it does not need to be part of a hierarchy.

13. Yes it can. However, if a method is going to be overridden (replaced in a child class) then I would be wary of providing too many overloaded forms of the method in the base class, as the child class will have to provide overriding implementations of all the overloaded versions, otherwise the behavior of the child objects may not be correct.

14. No. Structures are value types and cannot be used in class hierarchies.

15. Yes. Although structures cannot be used in class hierarchies they can be declared as implementing interfaces. A structure value will be boxed (turned from a value to a reference type – see Skill 2-2 for details) if it is to be accessed via an interface reference, but this will

16. A class that has been sealed cannot be used as the base of a child class that extends it.

17. Yes. Any object can implement `IEnumerable`. However, it is only meaning for items that contain or deal with enumerable collections to implement this interface.

18. No. It causes an enumerator to return the value specified by the yield return. When the next enumeration is requested from the enumerator the enumerator will continue from the statement following the yield.

19. No. The object will still exist in memory after it has been disposed. However, any attempt to use the object will result in an exception being thrown.

20. Yes. The C# system will not call `Dispose` automatically. However, if you use the `using` construction this will call `Dispose` for you.

21. Not necessarily. The use of `using` means that the system can be absolutely clear when an object is no longer required, but this will not invoke a garbage collection action to remove that object.

5 Find, execute, and create types at runtime by using reflection

1. The values of attributes are stored in the assembly file that is created when a program is compiled.

2. When a class is loaded the attribute class instances are created and the values in them are initialized. A program can change a value in an attribute, but this will not be persisted when the program ends. This is because programs are not allowed to change the contents of the assembly files from which they are loaded.

3. Yes it can, in fact the `AttributeUseage` class is an example of an attribute that is applied to attributes.

4. The type information for a class is stored in the assembly file where the code for that class is stored.

5. You can use `GetType` to get the type of any object in your program. `GetType` is provided by all objects. The `typeof` method is used to get the type object that represents a particular type. It is used if you don't have (or need) an instance of the object but you want to use the type data in your program.

6. The `GetType` method will return the reference to the type of any object; whether the object contents are managed by value or reference. You can even call `GetType` on a literal value in a program, in which case the value will be boxed (converted to a reference type) before the `GetType` runs.

7. No. You can create brand new types by using CodeDOM or expression trees, but you can't modify the behavior of a class by changing elements in its type description. This would be very dangerous if it was possible.

8. Yes. The `GetProperty` method provided by a `Type` object can be given `Binding` flags that specify `NonPublic` items. But you really shouldn't do this.

9. There is no difference between the speed of code produced "normally" and that produced by a provider that parses a CodeDOM document and generates a program that way.

10. Yes. A programmer given the assemblies for an application could extract all the class names and member names from the assembly. They can also view the actual code of the methods and there are even tools that will reconstruct a C# source file from an .exe or .dll file. If you are planning to release your code into the wild you should investigate technology which "obfuscates" your assembly files, changing the names of items and adding extra code that makes it much harder to understand what your program does.

11. Yes you could. This would be an interesting application of reflection (looking at code to find out what it does) and code generation. A program can add elements to a class being built in CodeDOM document that match those that it has discovered when parsing an interface. Visual Studio does something similar to this when you use the built in support that implements an interface for you in the code editor.

12. Yes you can. The compiled output of a CodeDOM object is an assembly like any other and will contain all the metadata.

6 Manage the object life cycle

1. Value types are normally stored on the stack (unless the type is part of a closure – see the Closures section in Skill 1.4, "Create and implement events and callbacks"). When a program exits from a block the variables on the stack are automatically removed. This means creating and destroying types such as integer, float, and double will not make any work for the garbage collector. However, if the value type is a struct, and the struct contains reference types, it will create work for the garbage collector, as the objects inside the structure will need to be managed.

2. Arrays of value types are implemented as objects. For example, an array of 100 integers will be managed as an object that contains 100 integers and an array reference to that object. This means that such arrays will be implemented on the heap and their removal from memory will involve the garbage collector.

4. Yes. It is the application developer's responsibility to make sure that the Dispose method is called to request that the object release any resources that it is using, but if this does not happen, and the object is deleted from memory, the finalize method will be called first. The dispose pattern has been designed to ensure that an object would release resources in a sensible way in this situation.

5. No. It would not be meaningful for the finalizer method in a class does to accept any parameters, so overloading the method (providing versions with different numbers of parameters) is not possible.

6. No. There is no need for a finalizer in a child class to make any calls to the finalizer in a parent class. Each class in a class hierarchy can have its own finalizer and each of them will be called in turn when the object is finalized.

7. When the Dispose method in an instance is called is down to the application itself. It may call Dispose multiple times. The dispose pattern uses a flag to ensure that this does not cause problems.

8. A long running finalizer will seriously impact on the performance of the garbage collection process. In an extreme situation this may result in the application running out of memory as the garbage collector cannot recover memory faster than it is being used.

9. It is bad practice for a finalizer to create a new reference to itself. This results in the object not being deleted from memory. Remember that the garbage collector makes another pass through the heap after the finalizers have been called. Any object that has a reference to it will be retained.

10. Security should be designed into an application very carefully from the start. However, in regard to performance and memory use, I advise that you don't do much more than avoid obviously stupid solutions that impact on these factors, and then just focus on creating an application that is easy to understand and test. If you subsequently discover that you have performance or memory issues, that is the point that you use your diagnostics tools to identify resource hungry elements of your program and then optimize those. A lot of developers (including me) have spent a lot of time making programs smaller and quicker when there is no need to do this, particularly with modern hardware.

7 Manipulate strings

1. The novel "War and Peace" by Tolstoy is generally accepted to be a long book. It contains around half a million words. However, even if each word was 20 characters long, this would only equate to around ten million characters. Strings are indexed using a 32-bit integer, which makes the theoretical limit on string size at around two thousand million characters. So you can say that all novels will fit into a single string.

2. The contents of a string object cannot be changed. However; a string variable can be made to refer to a different string instance. In other words, don't change the contents of a string variable called `name` from "Rob" to "Robert". Instead, make the string variable `name` refer to a new string instance that contains "Robert."

3. Interning of strings takes place when the compiler notices the same string literal is assigned to multiple variables in the program source. The compiler makes a single "interned" variable that contains the string literal and all the variables assigned that literal are made to refer to this single value. At run time this has the advantage that the program may be able to compare strings more quickly because two string variables will be equal if they both refer to the same interned object. The only disadvantage of interning is that it slows down the compilation process.

4. The `StringBuilder` type is not based on string. The `StringBuilder` will use one or more arrays of characters to build a string.

5. If a program does a lot of string addition to assemble an output string it may be that using `StringBuilder` to assemble the string will improve performance and perhaps reduce the number of intermediate objects that will be created.

6. You can't store binary data in a `StringWriter` because a `StringWriter` implements the `TextWriter` interface, which can only accept strings of text.

7. The `Trim` method is used to remove whitespace from an input string and return a new string with the whitespace removed. It does not change the contents of a string value because strings are immutable.

8. An object that implements the `iFormattable` interface contains a `ToString` method that accepts a format string and a format provider that can be used to provide culture specific information about the culture for which the string is being formatted. Objects that implement `iFormattable` can be used with formatted output which contains formatting instructions that are given the `ToString` methods.

9. A `CultureInfo` instance contains information describing the formatting of output for use by a particular culture. The `CultureInfo` instance contains references to the `NumberFormatInfo` and `DateTimeFormatInfo` objects for the culture. These specify such things as the ordering of dates when printed, the character to be used to mean currency and the

those parts of a `CultureInfo` instances as it needs to produce output that reflects different regions of the world.

10. String interpolation takes place when the compiler works on literal string values in the program source. A string that is identified as an interpolated string by having a dollar character ($) added before the start of the string will be parsed and the names of variables to be printed extracted. This all happens at compile time, so it is not possible to use interpolation on strings that are generated when the program runs.

CHAPTER SUMMARY

- Objects in a solution can be managed by value or by reference. A struct is a value type, which means that during assignment the contents of the struct (the value) is copied from one variable to another. A class is a reference type, which means that during assignment the destination of the assignment is made to refer to the same object as the source of the assignment.
- An immutable object is one that cannot be changed. Immutable objects, for example `DateTime`, provide methods that can be used to provide new, mutated copies of the original. For example, the `DateTime` structure provides a method that can be used to return a new `DateTime` structure that represents a given number of days into the future.
- A generic type can be used as a "placeholder" in a type design, so that the type of object that a type works on can be established dynamically. A good example of a situation where generic types are used is in the creation of lists and dictionaries.
- Types can be given a constructor method that is called when a new instance of the type is created. A constructor can be given parameters that can be used to initialize the object. If the constructor detects that the initialization data is invalid it must interrupt the construction process by throwing an exception, as the object will be created when the constructor completes. Constructors can be overloaded to provide different ways in which a given type can be instantiated. By use of the "this" syntax one constructor in type can call another constructor, to allow the creation of a master constructor and avoid code duplication in objects. Classes can contain a static constructor, which is called once when the first instance of the class is created.
- Types can contain methods that allow an object to perform behaviors. A method accepts a number of parameters and has a particular type. A method of type void does not return a value, whereas a method with any other type must return a result of that type.
- An extension method can be added to a class to add to the functionality of the class. The extension method does not have access to any of the private data members of the class it is added to.
- Parameters to a method can be identified by their names when called. This improves the readability of programs and also reduces the likelihood of errors which may be caused by parameters being entered in the wrong sequence.
- Parameters to a method can be made optional by providing a default value for the parameter. If the parameter is left out of the method call the default value is used instead.
- Objects can provide indexed properties which can be used to provide an index value (usually an integer) to identify the item to be returned.
- Methods in a single type can be overloaded. This means that the type will contain multiple versions of the same method, each of which has a different signature. Method overloading is used in situations where there are multiple ways of performing a given task from different data inputs.
- Methods in a class hierarchy can be overridden. To be overridden a method must be marked as virtual. When a virtual method is called on an instance of a class in the hierarchy the system will search up the class hierarchy for an implementation of the method, stating at the class of the object. The first implementation that is found is called. Method overriding is used in situations where objects in a class hierarchy need to be able to perform a given action in their own particular way. The base keyword allows a method to call the method it has overridden.
- Every item in an application is of a particular type. The C# type system will automatically convert between reference and value types and automatically convert (widen) values when it is sensible to do so and no data will be lost.
- If a type conversion will result in data loss (narrowing) the programmer must use casting to explicitly request the conversion of the values.
- Dynamic types are provided so that a C# application can interact with languages and services which are not statically typed. They allow the compiler to be told to ignore the type of an object and build the program even if the typing information indicates errors.
- When a dynamic type is used the type of the variable that is created is inferred from the context of the use. This means that what mistakes that would have been detected as compilation errors are now changed into run-time errors.

automatically mapped onto the correct types from the context of their use.

- The C# access modifiers allow members of a class to be hidden from code outside the class. Unless you specify otherwise, class members are private to a class and not visible outside it. Making a data member public will allow uncontrolled access to that data member in the class, which may not be a good idea.
- C# properties allow get and set behaviors to get control when an external code wishes to interact with a member of a class. The property can control access to private member of the class that holds a *backing value* for the data that is being managed by the class. Properties can provide only get behaviors for read only properties, and only set behaviors for write-only properties. The get and set behaviors can have different access modifiers so that the get behavior for a property can be made public and the set behavior private.
- Generally speaking, methods that a class exposes for other classes to use should be made public, and data contained within the class should be made private.
- The protected access modifier allows a class member to be made visible within classes in a class hierarchy and the internal access modifier allows a class member to be made visible to code in the same assembly as the class member.
- Methods in a class that implement the behaviors of an interface can be identified as explicitly implementing this interface. This improves encapsulation as it means that the only context in which those methods can be used is via a reference to the interface, not a reference to the object. Explicit implementation of behaviors also removes the potential for confusion if a class implements multiple interfaces and the same method appears in some of the interfaces.
- A C# class can implement an interface that contains a set of method signatures. Implementing an interface involves providing a method that matches each of the methods described in the interface. A class that implements an interface can be referred to by references of the interface type.
- A C# class can serve as the base class for a child class that extends it. A child class inherits all of the members of the base class and use C# the override mechanism to provide overridden versions of methods in the base class which have a behavior more specific to child class. For a method in a base class to be overridden it must be declared as virtual. An overriding method can use the base keyword to call overridden methods in the base class.
- A programmer can create replacement methods for those in a base class. This is not to be encouraged.
- A class can be declared as sealed to prevent it from being used as the basis of child classes. Methods in a class that extend those in a base class can also be declared as sealed, preventing them from being overridden.
- To construct a child class a program needs to create an instance of the base class. If a base class has a constructor the child class must call this and supply any required parameters. The base keyword is used in the constructor for this purpose.
- A class can be declared as abstract, in which case it will contain signatures of methods which are to be implemented in child classes. Abstract classes can serve as templates for classes.
- A reference to a base class can refer to any objects that are created from classes that extend the base class. However, the reverse is not true. A reference to a child class is not able to be made to refer to an instance of the parent.
- A class can implement the `IComparable` interface. This means that it contains a method called `CompareTo` which can be used to determine the ordering of objects of that type. The `IComparable` behavior can be used by methods such as the `Sort` method provided by the `List` type, which can place objects in order.
- A class can implement the `IEnumerable` interface, which means that it can provides a method `GetEnumerator` which will supply an enumerator which can be used by a consumer of the enumeration (for example a `foreach` construction) to work through the class.
- The C# `yield` keyword provides an easy way for a programmer to create enumerators. It retains the state of the enumerator method until the next items is requested from the enumeration by the consumer of the enumeration.
- A class can implement the `IDisposable` interface. The class will contain a `Dispose` method that can be called to instruct an instance of the class to release critical resources that it is using. The `Dispose` method is not called automatically by the garbage collection process, but it can be called automatically if the instance of the object is created and used within a C# `using` construction.
- The `IUnknown` interface can be used when creating .NET code that must interact directly with Component Object Model (COM) objects.
- Metadata is data about data. In the context of a C# program, the metadata for the code would be expressed in the form of one or

program is built and loaded into attribute class instances when the assembly is loaded.

- Attribute classes extend the Attribute class and have names that end with the text “Attribute”. An attribute class can be empty, in which case it just indicates that a particular attribute is applied to an item, or an attribute can contain data elements which can be accessed programmatically. Attributes can be applied to many different elements in a program; an `AttributeUseage` attribute can be added to an attribute declaration to specify objects and classes that can have the attribute applied.
- Reflection allows a program to investigate the contents of a type and programmatically interact with it.
- Reflection can be used on an assembly to allow a program determine the characteristics of types in the assembly. This can be used to allow a component-based system to automatically select and load the components that it needs.
- Programs code can be created programmatically by using the CodeDOM document model which allows namespaces, classes, properties and class members to be specified. A CodeDom object can be output as a binary assembly file or as a source file in C# or Visual Basic.
- Another way to programmatically create code behaviors is to create lambda expression trees which contain lambda expressions that give the actions to be performed, along with a range of expression node types that specify other program actions. Expression trees are used to express programmatic behavior for such things as LINQ queries and the implementation of dynamic languages.
- Garbage collection of objects makes program creation much easier. The garbage collection system in C# runs automatically and will delete unused objects (those that are not referred to by any reference).
- Application threads are paused while the garbage collection process runs.
- The garbage collector recognizes persistent objects (those which are present after a garbage collection) and omits those from the garbage collection process.
- The garbage collection process can be initiated manually, although this is not recommended.
- Unused objects that contain references to resources that need to be released when the object is deleted can contain a finalizer method that is executed during the garbage collection process. The finalizer method can release resources allocated to the object.
- In preference to a finalizer method, an object can implement the `IDisposable` interface and contain a `Dispose` method that can be used by the application to instruct the object to release any resources. The using statement can be used to ensure that the `Dispose` method is called on an object which is used in only one part of a program.
- By using the dispose pattern, an object can combine the use of finalizer and `Dispose` to ensure that resources held by the object are released correctly.
- A string is collection of characters of arbitrary length.
- String variables are immutable (they cannot be changed) but they are managed by reference. This combination means that they can be treated as value types.
- During compilation multiple copies of string literals for a particular string are mapped onto a single string instance in memory. In other words, if a program assigns the string literal “cheese” to several different string variables this would result in a single string value containing the text “cheese” being created to which all the variables would be made to refer.
- Building up a large string by adding many strings together will result in the creation of a large number of substrings. The `StringBuilder` type provides a string implementation which is mutable. A program can modify the contents in a `StringBuilder` instance. `StringBuilders` provide a more efficient means of assembling large strings.
- The `StringReader` and `StringWriter` classes allow strings to be used by programs that work with `TextReader` and `TextWriter` streams.
- The string type provides a range of methods for working with strings. It is possible to establish the location of substrings within the string, extract them and match the start and the end of a string.
- The contents of a string cannot be changed (strings are immutable) but the `Replace` string method can be used to create a new string instance with updated contents.
- String matching can be performed using the character codes or according to particular culture where multiple spellings of the same word may be matched.
- Strings can be enumerated, for example by using `foreach` construction.

- String interpolation allows a program to use string literals (preceded by a `$` character) that combine formatting information and the names of variables to be formatted and incorporated into the string. Interpolated strings are processed at compile time to produce composite format strings.

PREV
Chapter 1 Manage program flow

NEXT
Chapter 3 Debug applications and implement security