**Lecture 4 – Object Oriented Programming Using C#**

The key concepts, visibility, user types. Inheritance. Operator overload. Indexers.

C# is a general-purpose, type-safe, object-oriented programming language. The goal of the language is programmer productivity. To this end, the language balances simplicity, expressiveness, and performance. The chief architect of the language since its first version is Anders Hejlsberg (creator of Turbo Pascal and architect of Delphi). The C# language is platform-neutral, but it was written to work well with the Microsoft .NET Framework.

Object Orientation

C# is a rich implementation of the object-orientation paradigm, which includes *encapsulation*, *inheritance*, and *polymorphism*. Encapsulation means creating a boundary around an *object*, to separate its external (public) behavior from its internal (private) implementation details. The distinctive features of C# from an object-oriented perspective are:

*Unified type system*

The fundamental building block in C# is an encapsulated unit of data and functions called a *type*. C# has a *unified type system*, where all types ultimately share a common base type. This means that all types, whether they represent business objects or are primitive types such as numbers, share the same basic set of functionality. For example, an instance of any type can be converted to a string by calling its ToString method.

*Classes and interfaces*

In a traditional object-oriented paradigm, the only kind of type is a class. In C#, there are several other kinds of types, one of which is an *interface*. An interface is like a class, except that it only *describes* members. The implementation for those members comes from types that *implement* the interface. Interfaces are particularly useful in scenarios where multiple inheritance is required (unlike languages such as C++ and Eiffel, C# does not support multiple inheritance of classes).

*Properties, methods, and events*

In the pure object-oriented paradigm, all functions are *methods* (this is the case in Smalltalk). In C#, methods are only one kind of *function member*, which also includes *properties* and *events* (there are others, too). Properties are function members that encapsulate a piece of an object’s state, such as a button’s color or a label’s text. Events are function members that simplify acting on object state changes.

Type Safety

C# is primarily a *type-safe* language, meaning that instances of types can interact only through protocols they define, thereby ensuring each type’s internal consistency. For instance, C# prevents you from interacting with a *string* type as though it were an *integer* type. More specifically, C# supports *static typing*, meaning that the language enforces type safety at *compile time*. This is in addition to type safety being enforced at *runtime*. Static typing eliminates a large class of errors before a program is even run. It shifts the burden away from runtime unit tests onto the compiler to verify that all the types in a program fit together correctly. This makes large programs much easier to manage, more predictable, and more robust. Furthermore, static typing allows tools such as IntelliSense in Visual Studio to help you write a program, since it knows for a given variable what type it is, and hence what methods you can call on that variable. C# is also called a *strongly typed language* because its type rules (whether enforced statically or at runtime) are very strict. For instance, you cannot call a function that’s designed to accept an integer with a floating-point number, unless you first *explicitly* convert the floating-point number to an integer. This helps prevent mistakes. Strong typing also plays a role in enabling C# code to run in a sandbox—an environment where every aspect of security is controlled by the host. In a sandbox, it is important that you cannot arbitrarily corrupt the state of an object by bypassing its type rules.

Memory Management

C# relies on the runtime to perform automatic memory management. The Common Language Runtime has a garbage collector that executes as part of your program, reclaiming memory for objects that are no longer referenced. This frees programmers from explicitly deallocating the memory for an object, eliminating the problem of incorrect pointers encountered in languages such as C++. C# does not eliminate pointers: it merely makes them unnecessary for most programming tasks. For performance-critical hotspots and interoperability, pointers

may be used, but they are permitted only in blocks that are explicitly marked unsafe.

Platform Support

C# is typically used for writing code that runs on Windows platforms. Although Microsoft standardized the C# language through ECMA, the total amount of resources (both inside and outside of Microsoft) dedicated to supporting C# on non-Windows platforms is relatively small. This means that languages such as Java are sensible choices when multiplatform support is of primary concern. Having said this, C# can be used to write cross-platform code in the following scenarios:

• C# code may run on the server and dish up HTML that can run on any platform. This is precisely the case for ASP.NET.

• C# code may run on a runtime other than the Microsoft Common Language Runtime. The most notable example is the Mono project, which has its own C# compiler and runtime, running on Linux, Solaris, Mac OS X, and Windows.

• C# code may run on a host that supports Microsoft Silverlight (supported for Windows and Mac OS X). This technology is analogous to Adobe’s Flash Player.

**Private, Public, Protected, Internal**

All types and type members have an accessibility level, which controls whether they can be used from other code in your assembly or other assemblies. You can use the following access modifiers to specify the accessibility of a type or member when you declare it:

[public](http://msdn.microsoft.com/en-us/library/yzh058ae.aspx)

The type or member can be accessed by any other code in the same assembly or another assembly that references it.

[private](http://msdn.microsoft.com/en-us/library/st6sy9xe.aspx)

The type or member can be accessed only by code in the same class or struct.

[protected](http://msdn.microsoft.com/en-us/library/bcd5672a.aspx)

The type or member can be accessed only by code in the same class or struct, or in a class that is derived from that class.

[internal](http://msdn.microsoft.com/en-us/library/7c5ka91b.aspx)

The type or member can be accessed by any code in the same assembly, but not from another assembly.

**protected internal**

The type or member can be accessed by any code in the assembly in which it is declared, or from within a derived class in another assembly. Access from another assembly must take place within a class declaration that derives from the class in which the protected internal element is declared, and it must take place through an instance of the derived class type.

The following examples demonstrate how to specify access modifiers on a type and member:

C#

public class Bicycle

{

public void Pedal() { }

}

Not all access modifiers can be used by all types or members in all contexts, and in some cases the accessibility of a type member is constrained by the accessibility of its containing type. The following sections provide more details about accessibility.

[Class and Struct Accessibility](javascript:void(0))

Classes and structs that are declared directly within a namespace (in other words, that are not nested within other classes or structs) can be either public or internal. Internal is the default if no access modifier is specified.

Struct members, including nested classes and structs, can be declared as public, internal, or private. Class members, including nested classes and structs, can be public, protected internal, protected, internal, or private. The access level for class members and struct members, including nested classes and structs, is private by default. Private nested types are not accessible from outside the containing type.

Derived classes cannot have greater accessibility than their base types. In other words, you cannot have a public class B that derives from an internal class A. If this were allowed, it would have the effect of making Apublic, because all protected or internal members of A are accessible from the derived class.

You can enable specific other assemblies to access your internal types by using the InternalsVisibleToAttribute. For more information, see [Friend Assemblies (C# and Visual Basic)](http://msdn.microsoft.com/en-us/library/0tke9fxk.aspx).

[Class and Struct Member Accessibility](javascript:void(0))

Class members (including nested classes and structs) can be declared with any of the five types of access. Struct members cannot be declared as protected because structs do not support inheritance.

Normally, the accessibility of a member is not greater than the accessibility of the type that contains it. However, a public member of an internal class might be accessible from outside the assembly if the member implements interface methods or overrides virtual methods that are defined in a public base class.

The type of any member that is a field, property, or event must be at least as accessible as the member itself. Similarly, the return type and the parameter types of any member that is a method, indexer, or delegate must be at least as accessible as the member itself. For example, you cannot have a public method M that returns a class C unless C is also public. Likewise, you cannot have a protected property of type A if A is declared as private.

User-defined operators must always be declared as public. For more information, see [operator (C# Reference)](http://msdn.microsoft.com/en-us/library/s53ehcz3.aspx).

Destructors cannot have accessibility modifiers.

To set the access level for a class or struct member, add the appropriate keyword to the member declaration, as shown in the following example.

C#

// public class:

public class Tricycle

{

// protected method:

protected void Pedal() { }

// private field:

private int wheels = 3;

// protected internal property:

protected internal int Wheels

{

get { return wheels; }

}

}

**Inheritance in C#**

Classes can inherit from another class. This is accomplished by putting a colon after the class name when declaring the class, and naming the class to inherit from—the base class—after the colon, as follows:

C#

public class A

{

public A() { }

}

public class B : A

{

public B() { }

}

The new class—the derived class—then gains all the non-private data and behavior of the base class in addition to any other data or behaviors it defines for itself. The new class then has two effective types: the type of the new class and the type of the class it inherits.

In the example above, class B is effectively both B and A. When you access a B object, you can use the cast operation to convert it to an A object. The B object is not changed by the cast, but your view of the B object becomes restricted to A's data and behaviors. After casting a B to an A, that A can be cast back to a B. Not all instances of A can be cast to B—just those that are actually instances of B. If you access class B as a B type, you get both the class A and class B data and behaviors. The ability for an object to represent more than one type is called polymorphism. For more information, see [Polymorphism (C# Programming Guide)](http://msdn.microsoft.com/en-us/library/ms173152(v=vs.80).aspx). For more information on casting, see [Casting (C# Programming Guide)](http://msdn.microsoft.com/en-us/library/ms173105(v=vs.80).aspx).

Structs cannot inherit from other structs or classes. Both classes and structs can inherit from one or more interfaces. For more information, see [Interfaces (C# Programming Guide)](http://msdn.microsoft.com/en-us/library/ms173156(v=vs.80).aspx)

The [abstract](http://msdn.microsoft.com/en-us/library/sf985hc5(v=vs.80).aspx) keyword enables you to create classes and [class](http://msdn.microsoft.com/en-us/library/0b0thckt(v=vs.80).aspx) members solely for the purpose of inheritance—to define features of derived, non-abstract classes. The [sealed](http://msdn.microsoft.com/en-us/library/88c54tsw(v=vs.80).aspx) keyword enables you to prevent the inheritance of a class or certain class members that were previously marked [virtual](http://msdn.microsoft.com/en-us/library/9fkccyh4(v=vs.80).aspx). For more information, see [How to: Define Abstract Properties (C# Programming Guide)](http://msdn.microsoft.com/en-us/library/yd3z1377(v=vs.80).aspx).

[Abstract Classes and Class Members](javascript:void(0))

Classes can be declared as abstract. This is accomplished by putting the keyword **abstract** before the keyword **class** in the class definition. For example:

C#

public abstract class A

{

// Class members here.

}

An abstract class cannot be instantiated. The purpose of an abstract class is to provide a common definition of a base class that multiple derived classes can share. For example, a class library may define an abstract class that is used as a parameter to many of its functions, and require programmers using that library to provide their own implementation of the class by creating a derived class.

Abstract classes may also define abstract methods. This is accomplished by adding the keyword **abstract** before the return type of the method. For example:

C#

public abstract class A

{

public abstract void DoWork(int i);

}

Abstract methods have no implementation, so the method definition is followed by a semicolon instead of a normal method block. Derived classes of the abstract class must implement all abstract methods. When an abstract class inherits a virtual method from a base class, the abstract class can override the virtual method with an abstract method. For example:

C#

// compile with: /target:library

public class D

{

public virtual void DoWork(int i)

{

// Original implementation.

}

}

public abstract class E : D

{

public abstract override void DoWork(int i);

}

public class F : E

{

public override void DoWork(int i)

{

// New implementation.

}

}

If a virtual method is declared abstract, it is still virtual to any class inheriting from the abstract class. A class inheriting an abstract method cannot access the original implementation of the method—in the previous example, DoWork on class F cannot call DoWork on class D. In this way, an abstract class can force derived classes to provide new method implementations for virtual methods.

[Sealed Classes and Class Members](javascript:void(0))

Classes can be declared as sealed. This is accomplished by putting the keyword **sealed** before the keyword **class** in the class definition. For example:

C#

public sealed class D

{

// Class members here.

}

A sealed class cannot be used as a base class. For this reason, it cannot also be an abstract class. Sealed classes are primarily used to prevent derivation. Because they can never be used as a base class, some run-time optimizations can make calling sealed class members slightly faster.

A class member, method, field, property, or event, on a derived class that is overriding a virtual member of the base class can declare that member as sealed. This negates the virtual aspect of the member for any further derived class. This is accomplished by putting the **sealed** keyword before the [override](http://msdn.microsoft.com/en-us/library/ebca9ah3(v=vs.80).aspx) keyword in the class member declaration. For example:

C#

public class D : C

{

public sealed override void DoWork() { }

}

Through inheritance, a [class](http://msdn.microsoft.com/en-us/library/0b0thckt(v=vs.80).aspx) can be used as more than one type; it can be used as its own type, any base types, or any [interface](http://msdn.microsoft.com/en-us/library/87d83y5b(v=vs.80).aspx) type if it implements interfaces. This is called polymorphism. In C#, every type is polymorphic. Types can be used as their own type or as a [Object](http://msdn.microsoft.com/en-us/library/system.object(v=vs.80).aspx) instance, because any type automatically treats **Object** as a base type.

Polymorphism is important not only to the derived classes, but to the base classes as well. Anyone using the [base](http://msdn.microsoft.com/en-us/library/hfw7t1ce(v=vs.80).aspx) class could, in fact, be using an object of the derived class that has been cast to the base class type. Designers of a base class can anticipate the aspects of their base class that are likely to change for a derived type. For example, a base class for cars might contain behavior that is subject to change when the car in question is a minivan or a convertible. A base class can mark those class members as virtual, allowing derived classes representing convertibles and minivans to override that behavior.

For more information, see [Inheritance](http://msdn.microsoft.com/en-us/library/ms173149(v=vs.80).aspx).

# **Polymorphism Overview**

When a derived class inherits from a base class, it gains all the methods, fields, properties and events of the base class. To change the data and behavior of a base class, you have two choices: you can replace the base member with a new derived member, or you can override a virtual base member.

Replacing a member of a base class with a new derived member requires the [new](http://msdn.microsoft.com/en-us/library/51y09td4(v=vs.80).aspx) keyword. If a base class defines a method, field, or property, the **new** keyword is used to create a new definition of that method, field, or property on a derived class. The **new** keyword is placed before the return type of a class member that is being replaced. For example:

C#

public class BaseClass

{

public void DoWork() { }

public int WorkField;

public int WorkProperty

{

get { return 0; }

}

}

public class DerivedClass : BaseClass

{

public new void DoWork() { }

public new int WorkField;

public new int WorkProperty

{

get { return 0; }

}

}

When the **new** keyword is used, the new class members are called instead of the base class members that have been replaced. Those base class members are called hidden members. Hidden class members can still be called if an instance of the derived class is cast to an instance of the base class. For example:

C#

DerivedClass B = new DerivedClass();

B.DoWork(); // Calls the new method.

BaseClass A = (BaseClass)B;

A.DoWork(); // Calls the old method.

In order for an instance of a derived class to completely take over a class member from a base class, the base class has to declare that member as virtual. This is accomplished by adding the [virtual](http://msdn.microsoft.com/en-us/library/9fkccyh4(v=vs.80).aspx) keyword before the return type of the member. A derived class then has the option of using the [override](http://msdn.microsoft.com/en-us/library/ebca9ah3(v=vs.80).aspx) keyword, instead of **new**, to replace the base class implementation with its own. For example:

C#

public class BaseClass

{

public virtual void DoWork() { }

public virtual int WorkProperty

{

get { return 0; }

}

}

public class DerivedClass : BaseClass

{

public override void DoWork() { }

public override int WorkProperty

{

get { return 0; }

}

}

Fields cannot be virtual; only methods, properties, events and indexers can be virtual. When a derived class overrides a virtual member, that member is called even when an instance of that class is being accessed as an instance of the base class. For example:

C#

DerivedClass B = new DerivedClass();

B.DoWork(); // Calls the new method.

BaseClass A = (BaseClass)B;

A.DoWork(); // Also calls the new method.

Virtual methods and properties allow you to plan ahead for future expansion. Because a virtual member is called regardless of which type the caller is using, it gives derived classes the option to completely change the apparent behavior of the base class.

Virtual members remain virtual indefinitely, no matter how many classes have been declared between the class that originally declared the virtual member. If class A declares a virtual member, and class B derives from A, and class C derives from B, class C inherits the virtual member, and has the option to override it, regardless of whether class B declared an override for that member. For example:

C#

public class A

{

public virtual void DoWork() { }

}

public class B : A

{

public override void DoWork() { }

}

C#

public class C : B

{

public override void DoWork() { }

}

A derived class can stop virtual inheritance by declaring an override as sealed. This requires putting the [sealed](http://msdn.microsoft.com/en-us/library/88c54tsw(v=vs.80).aspx) keyword before the **override** keyword in the class member declaration. For example:

C#

public class C : B

{

public sealed override void DoWork() { }

}

In the previous example, the method DoWork is no longer virtual to any class derived from C. It is still virtual for instances of C, even if they are cast to type B or type A. Sealed methods can be replaced by derived classes using the **new** keyword, as the following example shows:

C#

public class D : C

{

public new void DoWork() { }

}

In this case, if DoWork is called on D using a variable of type D, the new DoWork is called. If a variable of type C, B, or A is used to access an instance of D, a call to DoWork will follow the rules of virtual inheritance, routing those calls to the implementation of DoWork on class C.

A derived class that has replaced or overridden a method or property can still access the method or property on the base class using the base keyword. For example:

C#

public class A

{

public virtual void DoWork() { }

}

public class B : A

{

public override void DoWork() { }

}

C#

public class C : B

{

public override void DoWork()

{

// Call DoWork on B to get B's behavior:

base.DoWork();

// DoWork behavior specific to C goes here:

// ...

}

}

**C# Overloading operators**

C# allows user-defined types to overload operators by defining static member functions using the [operator](http://msdn.microsoft.com/en-us/library/s53ehcz3.aspx) keyword. Not all operators can be overloaded, however, and others have restrictions, as listed in this table:

|  |  |
| --- | --- |
| **Operators** | **Overloadability** |
| [+](http://msdn.microsoft.com/en-us/library/k1a63xkz.aspx), [-](http://msdn.microsoft.com/en-us/library/wch5w409.aspx), [!](http://msdn.microsoft.com/en-us/library/f2kd6eb2.aspx), [~](http://msdn.microsoft.com/en-us/library/d2bd4x66.aspx), [++](http://msdn.microsoft.com/en-us/library/36x43w8w.aspx), [--](http://msdn.microsoft.com/en-us/library/wc3z3k8c.aspx), [true](http://msdn.microsoft.com/en-us/library/eahhcxk2.aspx), [false](http://msdn.microsoft.com/en-us/library/67bxt5ee.aspx) | These unary operators can be overloaded. |
| [+](http://msdn.microsoft.com/en-us/library/k1a63xkz.aspx), [-](http://msdn.microsoft.com/en-us/library/wch5w409.aspx), [\*](http://msdn.microsoft.com/en-us/library/z19tbbca.aspx), [/](http://msdn.microsoft.com/en-us/library/3b1ff23f.aspx), [%](http://msdn.microsoft.com/en-us/library/0w4e0fzs.aspx), [&](http://msdn.microsoft.com/en-us/library/sbf85k1c.aspx), [|](http://msdn.microsoft.com/en-us/library/kxszd0kx.aspx), [^](http://msdn.microsoft.com/en-us/library/zkacc7k1.aspx), [<<](http://msdn.microsoft.com/en-us/library/a1sway8w.aspx), [>>](http://msdn.microsoft.com/en-us/library/xt18et0d.aspx) | These binary operators can be overloaded. |
| [==](http://msdn.microsoft.com/en-us/library/53k8ybth.aspx), [!=](http://msdn.microsoft.com/en-us/library/3tz250sf.aspx), [<](http://msdn.microsoft.com/en-us/library/z5wecxwa.aspx), [>](http://msdn.microsoft.com/en-us/library/yxk8751b.aspx), [<=](http://msdn.microsoft.com/en-us/library/hx063734.aspx), [>=](http://msdn.microsoft.com/en-us/library/a59bsyk4.aspx) | The comparison operators can be overloaded (but see the note that follows this table). |
| [&&](http://msdn.microsoft.com/en-us/library/2a723cdk.aspx), [||](http://msdn.microsoft.com/en-us/library/6373h346.aspx) | The conditional logical operators cannot be overloaded, but they are evaluated using **&** and **|**, which can be overloaded. |
| [[]](http://msdn.microsoft.com/en-us/library/a3hd7ste.aspx) | The array indexing operator cannot be overloaded, but you can define indexers. |
| [(T)x](http://msdn.microsoft.com/en-us/library/0z4503sa.aspx) | The cast operator cannot be overloaded, but you can define new conversion operators (see [explicit](http://msdn.microsoft.com/en-us/library/xhbhezf4.aspx) and [implicit](http://msdn.microsoft.com/en-us/library/z5z9kes2.aspx)). |
| [+=](http://msdn.microsoft.com/en-us/library/sa7629ew.aspx), [-=](http://msdn.microsoft.com/en-us/library/2y9zhhx1.aspx), [\*=](http://msdn.microsoft.com/en-us/library/s2bkaksf.aspx), [/=](http://msdn.microsoft.com/en-us/library/d31sybc9.aspx), [%=](http://msdn.microsoft.com/en-us/library/ydwa9zh0.aspx), [&=](http://msdn.microsoft.com/en-us/library/e669ax02.aspx), [|=](http://msdn.microsoft.com/en-us/library/h5f1zzaw.aspx), [^=](http://msdn.microsoft.com/en-us/library/0zbsw2z6.aspx), [<<=](http://msdn.microsoft.com/en-us/library/ayt2kcfb.aspx), [>>=](http://msdn.microsoft.com/en-us/library/23as4533.aspx) | Assignment operators cannot be overloaded, but **+=**, for example, is evaluated using **+**, which can be overloaded. |
| [=](http://msdn.microsoft.com/en-us/library/sbkb459w.aspx), [.](http://msdn.microsoft.com/en-us/library/6zhxzbds.aspx), [?:](http://msdn.microsoft.com/en-us/library/ty67wk28.aspx), [??](http://msdn.microsoft.com/en-us/library/ms173224.aspx), [->](http://msdn.microsoft.com/en-us/library/s8bz4d5h.aspx), [=>](http://msdn.microsoft.com/en-us/library/bb311046.aspx), [f(x)](http://msdn.microsoft.com/en-us/library/0z4503sa.aspx), [as](http://msdn.microsoft.com/en-us/library/cscsdfbt.aspx), [checked](http://msdn.microsoft.com/en-us/library/74b4xzyw.aspx), [unchecked](http://msdn.microsoft.com/en-us/library/a569z7k8.aspx), [default](http://msdn.microsoft.com/en-us/library/xwth0h0d.aspx), [delegate](http://msdn.microsoft.com/en-us/library/0yw3tz5k.aspx), [is](http://msdn.microsoft.com/en-us/library/scekt9xw.aspx), [new](http://msdn.microsoft.com/en-us/library/51y09td4.aspx), [sizeof](http://msdn.microsoft.com/en-us/library/eahchzkf.aspx), [typeof](http://msdn.microsoft.com/en-us/library/58918ffs.aspx) | These operators cannot be overloaded. |
| **Note Note** | |
| The comparison operators, if overloaded, must be overloaded in pairs; that is, if **==** is overloaded, **!=** must also be overloaded. The reverse is also true, and similar for **<** and **>**, and for **<=** and **>=**. | |

To overload an operator on a custom class requires creating a method on the class with the correct signature. The method must be named "operator X" where X is the name or symbol of the operator being overloaded. Unary operators have one parameter, and binary operators have two parameters. In each case, one parameter must be the same type as the class or struct that declares the operator, as demonstrated in the following example:

public static Complex operator +(Complex c1, Complex c2)

**Properties and Indexers**

A property is a member that provides a flexible mechanism to read, write, or compute the value of a private field. Properties can be used as if they are public data members, but they are actually special methods called*accessors*. This enables data to be accessed easily and still helps promote the safety and flexibility of methods.

In this example, the TimePeriod class stores a time period. Internally the class stores the time in seconds, but a property named Hours enables a client to specify a time in hours. The accessors for the Hours property perform the conversion between hours and seconds.

## [Example](javascript:void(0))

C#

class TimePeriod

{

private double seconds;

public double Hours

{

get { return seconds / 3600; }

set { seconds = value \* 3600; }

}

}

class Program

{

static void Main()

{

TimePeriod t = new TimePeriod();

// Assigning the Hours property causes the 'set' accessor to be called.

t.Hours = 24;

// Evaluating the Hours property causes the 'get' accessor to be called.

System.Console.WriteLine("Time in hours: " + t.Hours);

}

}

// Output: Time in hours: 24

## [Properties Overview](javascript:void(0))

* Properties enable a class to expose a public way of getting and setting values, while hiding implementation or verification code.
* A [get](http://msdn.microsoft.com/en-us/library/ms228503.aspx) property accessor is used to return the property value, and a [set](http://msdn.microsoft.com/en-us/library/ms228368.aspx) accessor is used to assign a new value. These accessors can have different access levels. For more information, see [Restricting Accessor Accessibility (C# Programming Guide)](http://msdn.microsoft.com/en-us/library/75e8y5dd.aspx).
* The [value](http://msdn.microsoft.com/en-us/library/a1khb4f8.aspx) keyword is used to define the value being assigned by the **set** accessor.
* Properties that do not implement a **set** accessor are read only.
* For simple properties that require no custom accessor code, consider the option of using auto-implemented properties.

Properties combine aspects of both fields and methods. To the user of an object, a property appears to be a field, accessing the property requires the same syntax. To the implementer of a class, a property is one or two code blocks, representing a [get](http://msdn.microsoft.com/en-us/library/ms228503.aspx) accessor and/or a [set](http://msdn.microsoft.com/en-us/library/ms228368.aspx) accessor. The code block for the **get** accessor is executed when the property is read; the code block for the **set** accessor is executed when the property is assigned a new value. A property without a **set** accessor is considered read-only. A property without a **get** accessor is considered write-only. A property that has both accessors is read-write.

Unlike fields, properties are not classified as variables. Therefore, you cannot pass a property as a [ref (C# Reference)](http://msdn.microsoft.com/en-us/library/14akc2c7.aspx) or [out (C# Reference)](http://msdn.microsoft.com/en-us/library/t3c3bfhx.aspx) parameter.

Properties have many uses: they can validate data before allowing a change; they can transparently expose data on a class where that data is actually retrieved from some other source, such as a database; they can take an action when data is changed, such as raising an event, or changing the value of other fields.

Properties are declared in the class block by specifying the access level of the field, followed by the type of the property, followed by the name of the property, and followed by a code block that declares a **get**-accessor and/or a **set** accessor. For example:

C#

public class Date

{

private int month = 7; // Backing store

public int Month

{

get

{

return month;

}

set

{

if ((value > 0) && (value < 13))

{

month = value;

}

}

}

}

In this example, Month is declared as a property so that the **set** accessor can make sure that the Month value is set between 1 and 12. The Month property uses a private field to track the actual value. The real location of a property's data is often referred to as the property's "backing store." It is common for properties to use private fields as a backing store. The field is marked private in order to make sure that it can only be changed by calling the property. For more information about public and private access restrictions, see [Access Modifiers (C# Programming Guide)](http://msdn.microsoft.com/en-us/library/ms173121.aspx).

Auto-implemented properties provide simplified syntax for simple property declarations. For more information, see [Auto-Implemented Properties (C# Programming Guide)](http://msdn.microsoft.com/en-us/library/bb384054.aspx).

## [The get Accessor](javascript:void(0))

The body of the **get** accessor resembles that of a method. It must return a value of the property type. The execution of the **get** accessor is equivalent to reading the value of the field. For example, when you are returning the private variable from the **get** accessor and optimizations are enabled, the call to the **get** accessor method is inlined by the compiler so there is no method-call overhead. However, a virtual **get** accessor method cannot be inlined because the compiler does not know at compile-time which method may actually be called at run time. The following is a **get** accessor that returns the value of a private field name:

C#

class Person

{

private string name; // the name field

public string Name // the Name property

{

get

{

return name;

}

}

}

When you reference the property, except as the target of an assignment, the **get** accessor is invoked to read the value of the property. For example:

C#

Person person = new Person();

//...

System.Console.Write(person.Name); // the get accessor is invoked here

The **get** accessor must end in a [return](http://msdn.microsoft.com/en-us/library/1h3swy84.aspx) or [throw](http://msdn.microsoft.com/en-us/library/1ah5wsex.aspx) statement, and control cannot flow off the accessor body.

It is a bad programming style to change the state of the object by using the **get** accessor. For example, the following accessor produces the side effect of changing the state of the object every time that the numberfield is accessed.

C#

private int number;

public int Number

{

get

{

return number++; // Don't do this

}

}

The **get** accessor can be used to return the field value or to compute it and return it. For example:

C#

class Employee

{

private string name;

public string Name

{

get

{

return name != null ? name : "NA";

}

}

}

In the previous code segment, if you do not assign a value to the Name property, it will return the value NA.

## [The set Accessor](javascript:void(0))

The **set** accessor resembles a method whose return type is [void](http://msdn.microsoft.com/en-us/library/yah0tteb.aspx). It uses an implicit parameter called *value*, whose type is the type of the property. In the following example, a **set** accessor is added to the Name property:

C#

class Person

{

private string name; // the name field

public string Name // the Name property

{

get

{

return name;

}

set

{

name = value;

}

}

}

When you assign a value to the property, the **set** accessor is invoked by using an argument that provides the new value. For example:

C#

Person person = new Person();

person.Name = "Joe"; // the set accessor is invoked here

System.Console.Write(person.Name); // the get accessor is invoked here

It is an error to use the implicit parameter name, *value*, for a local variable declaration in a **set** accessor.

## [Remarks](javascript:void(0))

Properties can be marked as **public**, **private**, **protected**, **internal**, or **protected internal**. These access modifiers define how users of the class can access the property. The **get** and **set** accessors for the same property may have different access modifiers. For example, the **get** may be **public** to allow read-only access from outside the type, and the **set** may be **private** or **protected**. For more information, see [Access Modifiers (C# Programming Guide)](http://msdn.microsoft.com/en-us/library/ms173121.aspx).

A property may be declared as a static property by using the **static** keyword. This makes the property available to callers at any time, even if no instance of the class exists. For more information, see [Static Classes and Static Class Members (C# Programming Guide)](http://msdn.microsoft.com/en-us/library/79b3xss3.aspx).

A property may be marked as a virtual property by using the [virtual](http://msdn.microsoft.com/en-us/library/9fkccyh4.aspx) keyword. This enables derived classes to override the property behavior by using the [override](http://msdn.microsoft.com/en-us/library/ebca9ah3.aspx) keyword. For more information about these options, see [Inheritance (C# Programming Guide)](http://msdn.microsoft.com/en-us/library/ms173149.aspx).

A property overriding a virtual property can also be [sealed](http://msdn.microsoft.com/en-us/library/88c54tsw.aspx), specifying that for derived classes it is no longer virtual. Lastly, a property can be declared [abstract](http://msdn.microsoft.com/en-us/library/sf985hc5.aspx). This means that there is no implementation in the class, and derived classes must write their own implementation.

**Indexers**

Indexers allow instances of a class or struct to be indexed just like arrays. Indexers resemble [properties](http://msdn.microsoft.com/en-us/library/x9fsa0sw.aspx) except that their accessors take parameters.

In the following example, a generic class is defined and provided with simple [get](http://msdn.microsoft.com/en-us/library/ms228503.aspx) and [set](http://msdn.microsoft.com/en-us/library/ms228368.aspx) accessor methods as a means of assigning and retrieving values. The Program class creates an instance of this class for storing strings.

C#

class SampleCollection<T>

{

// Declare an array to store the data elements.

private T[] arr = new T[100];

// Define the indexer, which will allow client code

// to use [] notation on the class instance itself.

// (See line 2 of code in Main below.)

public T this[int i]

{

get

{

// This indexer is very simple, and just returns or sets

// the corresponding element from the internal array.

return arr[i];

}

set

{

arr[i] = value;

}

}

}

// This class shows how client code uses the indexer.

class Program

{

static void Main(string[] args)

{

// Declare an instance of the SampleCollection type.

SampleCollection<string> stringCollection = new SampleCollection<string>();

// Use [] notation on the type.

stringCollection[0] = "Hello, World";

System.Console.WriteLine(stringCollection[0]);

}

}

// Output:

// Hello, World.

## [Indexers Overview](javascript:void(0))

* Indexers enable objects to be indexed in a similar manner to arrays.
* A **get** accessor returns a value. A **set** accessor assigns a value.
* The [this](http://msdn.microsoft.com/en-us/library/dk1507sz.aspx) keyword is used to define the indexers.
* The [value](http://msdn.microsoft.com/en-us/library/a1khb4f8.aspx) keyword is used to define the value being assigned by the **set** indexer.
* Indexers do not have to be indexed by an integer value; it is up to you how to define the specific look-up mechanism.
* Indexers can be overloaded.
* Indexers can have more than one formal parameter, for example, when accessing a two-dimensional array.

Indexers are a syntactic convenience that enable you to create a [class](http://msdn.microsoft.com/en-us/library/0b0thckt.aspx), [struct](http://msdn.microsoft.com/en-us/library/ah19swz4.aspx), or [interface](http://msdn.microsoft.com/en-us/library/87d83y5b.aspx) that client applications can access just as an array. Indexers are most frequently implemented in types whose primary purpose is to encapsulate an internal collection or array. For example, suppose you have a class named TempRecord that represents the temperature in Farenheit as recorded at 10 different times during a 24 hour period. The class contains an array named "temps" of type float to represent the temperatures, and a [DateTime](http://msdn.microsoft.com/en-us/library/system.datetime.aspx) that represents the date the temperatures were recorded. By implementing an indexer in this class, clients can access the temperatures in a TempRecord instance as float temp = tr[4] instead of as float temp = tr.temps[4]. The indexer notation not only simplifies the syntax for client applications; it also makes the class and its purpose more intuitive for other developers to understand.

To declare an indexer on a class or struct, use the [this](http://msdn.microsoft.com/en-us/library/dk1507sz.aspx) keyword, as in this example:

public int this[int index] // Indexer declaration

{

// get and set accessors

}

## [Remarks](javascript:void(0))

The type of an indexer and the type of its parameters must be at least as accessible as the indexer itself. For more information about accessibility levels, see [Access Modifiers](http://msdn.microsoft.com/en-us/library/wxh6fsc7.aspx).

For more information about how to use indexers with an interface, see [Interface Indexers](http://msdn.microsoft.com/en-us/library/tkyhsw31.aspx).

The signature of an indexer consists of the number and types of its formal parameters. It does not include the indexer type or the names of the formal parameters. If you declare more than one indexer in the same class, they must have different signatures.

An indexer value is not classified as a variable; therefore, you cannot pass an indexer value as a [ref](http://msdn.microsoft.com/en-us/library/14akc2c7.aspx) or [out](http://msdn.microsoft.com/en-us/library/t3c3bfhx.aspx) parameter.

To provide the indexer with a name that other languages can use, use a name attribute in the declaration. For example:

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

public int this [int index] // Indexer declaration

{

}

This indexer will have the name TheItem. Not providing the name attribute would make Item the default name.

## [Example 1](javascript:void(0))

### [**Description**](javascript:void(0))

The following example shows how to declare a private array field, temps, and an indexer. The indexer enables direct access to the instance tempRecord[i]. The alternative to using the indexer is to declare the array as a [public](http://msdn.microsoft.com/en-us/library/yzh058ae.aspx) member and access its members, tempRecord.temps[i], directly.

Notice that when an indexer's access is evaluated, for example, in a **Console.Write** statement, the [get](http://msdn.microsoft.com/en-us/library/ms228503.aspx) accessor is invoked. Therefore, if no **get** accessor exists, a compile-time error occurs.

### [**Code**](javascript:void(0))

C#

class TempRecord

{

// Array of temperature values

private float[] temps = new float[10] { 56.2F, 56.7F, 56.5F, 56.9F, 58.8F,

61.3F, 65.9F, 62.1F, 59.2F, 57.5F };

// To enable client code to validate input

// when accessing your indexer.

public int Length

{

get { return temps.Length; }

}

// Indexer declaration.

// If index is out of range, the temps array will throw the exception.

public float this[int index]

{

get

{

return temps[index];

}

set

{

temps[index] = value;

}

}

}

class MainClass

{

static void Main()

{

TempRecord tempRecord = new TempRecord();

// Use the indexer's set accessor

tempRecord[3] = 58.3F;

tempRecord[5] = 60.1F;

// Use the indexer's get accessor

for (int i = 0; i < 10; i++)

{

System.Console.WriteLine("Element #{0} = {1}", i, tempRecord[i]);

}

// Keep the console window open in debug mode.

System.Console.WriteLine("Press any key to exit.");

System.Console.ReadKey();

}

}

/\* Output:

Element #0 = 56.2

Element #1 = 56.7

Element #2 = 56.5

Element #3 = 58.3

Element #4 = 58.8

Element #5 = 60.1

Element #6 = 65.9

Element #7 = 62.1

Element #8 = 59.2

Element #9 = 57.5

\*/

## [Indexing Using Other Values](javascript:void(0))

C# does not limit the index type to integer. For example, it may be useful to use a string with an indexer. Such an indexer might be implemented by searching for the string in the collection, and returning the appropriate value. As accessors can be overloaded, the string and integer versions can co-exist.

## [Example 2](javascript:void(0))

### [**Description**](javascript:void(0))

In this example, a class is declared that stores the days of the week. A **get** accessor is declared that takes a string, the name of a day, and returns the corresponding integer. For example, Sunday will return 0, Monday will return 1, and so on.

### [**Code**](javascript:void(0))

C#

// Using a string as an indexer value

class DayCollection

{

string[] days = { "Sun", "Mon", "Tues", "Wed", "Thurs", "Fri", "Sat" };

// This method finds the day or returns -1

private int GetDay(string testDay)

{

for (int j = 0; j < days.Length; j++)

{

if (days[j] == testDay)

{

return j;

}

}

throw new System.ArgumentOutOfRangeException(testDay, "testDay must be in the form \"Sun\", \"Mon\", etc");

}

// The get accessor returns an integer for a given string

public int this[string day]

{

get

{

return (GetDay(day));

}

}

}

class Program

{

static void Main(string[] args)

{

DayCollection week = new DayCollection();

System.Console.WriteLine(week["Fri"]);

// Raises ArgumentOutOfRangeException

System.Console.WriteLine(week["Made-up Day"]);

// Keep the console window open in debug mode.

System.Console.WriteLine("Press any key to exit.");

System.Console.ReadKey();

}

}

// Output: 5

## [Robust Programming](javascript:void(0))

There are two main ways in which the security and reliability of indexers can be improved:

* Be sure to incorporate some type of error-handling strategy to handle the chance of client code passing in an invalid index value. In the first example earlier in this topic, the TempRecord class provides a Length property that enables the client code to verify the input before passing it to the indexer. You can also put the error handling code inside the indexer itself. Be sure to document for users any exceptions that you throw inside an indexer accessor.
* Set the accessibility of the **get** and [set](http://msdn.microsoft.com/en-us/library/ms228368.aspx) accessors to be as restrictive as is reasonable. This is important for the **set** accessor in particular.

Questions

1. Ad hoc polymorphism and Classic Polymorphism?
2. Encapsulation methods and Properties accessors ?
3. Main class properties?
4. Constructors. base keyword.
5. In which cases can we use base keyword?
6. Delegation model implementation.
7. Methods override approaches.