Chapter: 3

**C# Only : Operator Overloading, Indexers, and Properties**

Operator overloading fundamentals, Overload binary/unary/relational operators, Indexers,

**C#\_3.1 The General Forms of an Operator Method**

When an operator is overloaded, none of its original meaning is lost. It is simply that a new operation, relative to a specific class, is added. Therefore, overloading the + to handle a linked list, for example, does not cause its meaning relative to integers (that is, addition) to be changed.

* Operator overloading is closely related to method overloading. To ***overload an operator***, use the operator keyword to define an operator method, which defines the *action* of the *operator*. There are two forms of operator methods: one for unary operators and one for binary operators. General forms are:

***public static ret-type operator op(param-type operand){*** */\* General form for overloading a* ***unary operator****.\*/* ***}***

***public static ret-type operator op(param-type1 operand1, param-type2 operand2){***

*/\* General form for overloading a* ***binary operator****.\*/* ***}***

* op is the overloaded operator, such as + or /. The ret-type is the type of value returned by the specified operation. Return value is often of the same type as the class for which the operator is being overloaded.
* For unary operators, the operand is passed in operand. And the operand must be of the same type as the class for which the operator is being defined.
* For binary operators, the operands are passed in operand1 and operand2. And at least one of the operands must be of the same type as the class.
* Operator parameters must not use the ref or out modifier. And you cannot overload any C# operators for objects that you have not created.

|  |  |
| --- | --- |
| * Overloading Binary (arithmetic +) Operator: | |
| **using System**;  **class** ThreeD { **int** x, y, z; *// 3-D coordinates*  **public** ThreeD() { x = y = z = 0; }  **public** ThreeD(**int** i, **int** j, **int** k) { x = i; y = j; z = k; }  **public** **static** ThreeD **operator** **+**(**ThreeD** op1, **ThreeD** op2) { *// Overload binary +.*  **ThreeD** result = **new** ThreeD();  */\* This adds together the coordinates of the two points and returns the result. \*/*  result.x = op1.x + op2.x; *// These are integer additions*  result.y = op1.y + op2.y; *// and the + retains its original*  result.z = op1.z + op2.z; *// meaning relative to them.*  **return** result; }  **public** **void** Show(){ **Console.WriteLine**(x + ", " + y + ", " + z); } } | **class** ThreeDDemo{ **static void Main(){**  **ThreeD** a = **new** ThreeD(1, 2, 3);  **ThreeD** b = **new** ThreeD(10, 10, 10);  **ThreeD** c = **new** ThreeD();  c = a + b; *// add a and b together*  **Console.Write("**Result of a + b: ");  c.**Show**();  **Console.WriteLine**(); }}   * Notice that ***operator+()*** returns an object of type ThreeD. Although the method could have returned any valid C# type. |

* Overloading Unary Operators: Unary operators are overloaded just like the binary operators. The main difference, of course, is that there is only one operand.

**public** **static** ThreeD **operator -**(**ThreeD** op){**ThreeD** result = **new** ThreeD(); result.x = -op.x; result.y = -op.y; result.z = -op.z; **return** result; }

* Here, a new object is created that contains the negated fields of the operand. This object is then returned. Notice that the operand is unchanged. Again, this is in keeping with the usual meaning of the unary minus. For example, in an expression such as **a = -b** a receives the negation of b, but b is not changed.
* In C#, overloading **++** and **–** **–** is quite easy; simply return the incremented or decremented value, but don’t change the invoking object. C# will automatically handle that for you, taking into account the difference between the prefix and postfix forms.

**public static** ThreeD **operator** **++**(**ThreeD** op) { **ThreeD** result = **new** ThreeD(); result.x = op.x + 1; result.y = op.y + 1; result.z = op.z + 1; **return** result; }

* Flexibility of Operator overloading: In ThreeD example we overloaded + for two ThreeD types. We can do it for " ThreeD + ThreeD, ThreeD + int, and int + ThreeD ". This called flexibility. Following demonstrate this process:

**class** ThreeD { **int** x, y, z; *// 3-D coordinates*

**public** ThreeD() { x = y = z = 0; }

**public** ThreeD(**int** i, **int** j, **int** k) { x = i; y = j; z = k; }

*// Overload binary + for ThreeD + ThreeD.*

**public static ThreeD operator +**(**ThreeD** op1, **ThreeD** op2) { **ThreeD** result = **new** ThreeD();

result.x = op1.x + op2.x; result.y = op1.y + op2.y; result.z = op1.z + op2.z; **return** result; }

*// Overload binary + for ThreeD + int.*

**public static ThreeD operator +**(**ThreeD** op1, **int** op2) { **ThreeD** result = **new** ThreeD();

result.x = op1.x + op2; result.y = op1.y + op2; result.z = op1.z + op2; **return** result; }

*// Overload binary + for int + ThreeD.*

**public static ThreeD operator +**(**int** op1, **ThreeD** op2) { **ThreeD** result = **new** ThreeD();

result.x = op2.x + op1; result.y = op2.y + op1; result.z = op2.z + op1; **return** result; }

*// Show X, Y, Z coordinates.*

**public void** Show() { **Console.WriteLine**(x + ", " + y + ", " + z); } }

* Overloading the Relational Operators: Usually, an overloaded relational operator returns a true or false value. Consider following example:

**public static bool operator <**(**ThreeD** op1, **ThreeD** op2) {

**if**(**Math.Sqrt**(op1.x \* op1.x + op1.y \* op1.y + op1.z \* op1.z) < **Math.Sqrt**(op2.x \* op2.x + op2.y \* op2.y + op2.z \* op2.z)) **return** **true**;

**else return false**; }

**public static bool operator >**(**ThreeD** op1, **ThreeD** op2) {

**if**(**Math.Sqrt**(op1.x \* op1.x + op1.y \* op1.y + op1.z \* op1.z) > **Math.Sqrt**(op2.x \* op2.x + op2.y \* op2.y + op2.z \* op2.z)) **return** **true**;

**else return false**; }

* Restrictions: You must overload them in pairs. For example, if you overload **<**, you must also overload **>**, and vice versa. The operator pairs are:

( **= =** , **!=** ), ( **<** , **>** ), ( **<=** , **>=** ) I.e., if overload **<=**, you must also overload **>=**, & if overload **= =**, must also overload **!=**.

* If you overload the = = and != operators, you will usually need to override ***Object.Equals( )*** and ***Object.GetHashCode( )***.
* An overloaded operator should reflect, when possible, the spirit of the operator’s original use. For example, the ***+*** relative to ***ThreeD*** is conceptually similar to the ***+*** relative to ***integer*** types. While you can give an overloaded operator any meaning you like, for clarity, it is best when its new meaning is related to its original meaning.
* You cannot alter the precedence of any operator.
* You cannot alter the number of operands required by the operator, although your operator method could choose to ignore an operand.
* You cannot overload any assignment operator, including the compound assignments, such as **+=**. Following operators cannot be overloaded:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| && | ( ) | . | ? | ?? | [ ] | || | = | => |
| −> | as | checked | default | is | new | sizeof | typeof | unchecked |

* The keywords true and false can also be used as unary operators for the purposes of overloading. They are overloaded relative to a class to determine whether an object is “true” or “false.” Once these are overloaded for a class, you can use objects of that class to control an if statement, for example.
* Compound assignment with overloaded operational part of that assignment: If you have defined an operator, when that operator is used in a compound assignment, your overloaded operator method is invoked. Thus, ***+=*** automatically uses your version of operator ***+()***. Eg: Consider the ThreeD class:

**ThreeD** a = **new** ThreeD(1, 2, 3); **ThreeD** b = **new** ThreeD(10, 10, 10); b += a; *// add a and b together*

* ThreeD’s operator **+()** is automatically invoked, and b will contain the coordinates **11, 12, 13**.

**C#\_3.2 Indexers: The [ ] operator**

***Array indexing*** is performed using the ***[ ] operator***. It is possible to overload the [ ] operator for classes that you create, but you don’t use an operator method. Instead, you create an indexer. An indexer allows an object to be indexed like an array. The main use of indexers is to support the creation of specialized arrays that are subject to one or more constraints. However, you can use an indexer for any purpose for which an array-like syntax is beneficial. Indexers can have one or more dimensions.

* One-dimensional indexers: One-dimensional indexers have this general form:

**element-type this[int index]{ get {** */\* return the value specified by index \*/* **}** */\* The get accessor.\*/*

**set {** */\* set the value specified by index \*/* **}** */\* The set accessor.\*/*  **}**

* Here, element-type is the element type of the indexer. It corresponds to the ***element type of an array***.
* The parameter index receives the ***index of the element*** being accessed. indexdoes not have to be int, but for array indexing, an int type is customary.
* get and set accessors: An accessor is similar to a method, except that it does not declare a return type or parameters. The accessors are ***automatically called*** when the indexer is used, and both accessors receive index as a parameter.
* If the indexer is being assigned, such as when it’s on the left side of an ***assignment statement***, then the set accessor is called and the ***element specified*** by ***index*** must be set. Otherwise, the get accessor is called and the value associated with index must be returned. The set method also receives a value called value, which contains the ***value being assigned*** to the ***specified index***.
* One of the benefits of an indexer is that you can control precisely how an array is accessed, heading off improper accesses. Following uses an indexer, thus allowing the array to be accessed using the normal array notation. The indexer prevents the array boundaries from being overrun.

|  |  |
| --- | --- |
| using System ;  class FailSoftArray { int[] a; *// reference to array*  public int Length; *// Length is public*  public bool ErrFlag; *// outcome of last operation*  *// Construct array given its size.*  public FailSoftArray(int size) { a = new int[size]; Length = size; }  *// This is the indexer for FailSoftArray.*  public int this[int index] {  get { if(ok(index)) { ErrFlag = false; return a[index]; }  else { ErrFlag = true; return 0; } }  set { if(ok(index)) { a[index] = value; ErrFlag = false; }  else ErrFlag = true; } }  *// Return true if index is within bounds.*  private bool ok(int index) { if(index >= 0 & index < Length) return true;  return false; } } | class ImprovedFSDemo { static void Main() {  int x;  FailSoftArray fs = new FailSoftArray(5);  Console.WriteLine("Fail quietly."); *// Show quiet failures.*  for(int i=0; i < (fs.Length \* 2); i++) fs[i] = i\*10; */\* Invoke the indexer’s set \*/*  for(int i=0; i < (fs.Length \* 2); i++) { x = fs[i]; */\* Invoke the indexer’s get \*/*  if(x != -1) Console.Write(x + " "); }  Console.WriteLine();  *// Now, generate failures.*  Console.WriteLine("\nFail with error reports.");  for(int i=0; i < (fs.Length \* 2); i++) { fs[i] = i\*10;  if(fs.ErrFlag) Console.WriteLine("fs[" + i + "] out-of-bounds"); }  for(int i=0; i < (fs.Length \* 2); i++) { x = fs[i];  if(!fs.ErrFlag) Console.Write(x + " ");  else Console.WriteLine("fs[" + i + "] out-of-bounds"); }  }} |

* **public int this[int index] {** Declares an indexer that operates on int elements. The index is passed in index. The indexer is public, allowing it to be used by code outside of its class.
* The get accessor prevents array boundary errors. If the specified index is within bounds, the element corresponding to the index is returned. If it is out of bounds, no operation takes place and no overrun occurs.
* A variable called ErrFlag contains the outcome of each operation. This field can be examined after each operation to assess the success or failure of the operation.
* set too, prevents a boundary error. If index is within bounds, the value passed in value is assigned to the corresponding element. Otherwise, ErrFlag is set to true. Recall that in an ***accessor method***, value is an automatic parameter that contains the value being assigned. You do not need to (nor can you) declare it.
* It is not necessary for an indexer to provide both get and set. You can create a read-only indexer by implementing only the get accessor. You can create a write-only indexer by implementing only set.
* There is no requirement that an indexer actually operate on an array. It simply must provide functionality that appears “array-like” to the user of the indexer. Eg:

**class** PwrOfTwo { /\* . . . \*/

**public** **int this**[**int** index] { *// Compute and return power of 2.*

**get** { **if**((index >= 0) && (index < 16)) **return** Pwr(index); **else** **return -1**; } */\* There is no set accessor. \*/* }

**int** Pwr(**int** p) { **int** result = 1; **for**(**int** i=0; i<p; i++) result \*= 2; **return** result; }

}

* There no set accessor, i.e. the indexer is read-only. Thus, a object can be used on the right side of an assignment statement, but not on the left. For example, attempting to add this statement to the program won’t work: **PwrOfTwo pwr = new PwrOfTwo(); pwr[0] = 11;** *// won't compile*

This statement will cause a compilation error because there is no set accessor defined for the indexer.

* Multidimensional Indexers: You can create indexers for multidimensional arrays, too. For example, here is a two-dimensional fail-soft array.

|  |  |
| --- | --- |
| using System;  class FailSoftArray2D { int[,] a; *// reference to 2D array*  int rows, cols; *// dimensions*  public int Length; *// Length is public*  public bool ErrFlag; *// outcome of last operation*  *// Construct array given its dimensions.*  public FailSoftArray2D(int r, int c) {  rows = r; cols = c;  a = new int[rows, cols];  Length = rows \* cols; } | public int this[int index1, int index2] {  get { if(ok(index1, index2)) { ErrFlag = false; return a[index1, index2]; }  else { ErrFlag = true; return 0; } }  set { if(ok(index1, index2)) { a[index1, index2] = value; ErrFlag = false; }  else ErrFlag = true; } } *//indexer ends*  *// Return true if indexes are within bounds.*  private bool ok(int index1, int index2) {  if(index1 >= 0 & index1 < rows & index2 >= 0 & index2 < cols) return true;  return false;} } *//class ends* |

* Indexers can be overloaded: The version executed will be the one that has the closest type-match between its parameter(s) and the argument(s) used as an index.
* Restrictions to using indexers: Because an indexer does not define a storage location, a value produced by an indexer cannot be passed as a ref or out parameter to a method. Second, an indexer cannot be declared static.

**C#\_3.3 Properties**

A property combines a field with the methods that access it. Used to create a field that is available to users of an object, but maintain control over what operations are allowed on that field. For instance, you might want to limit the range of values that can be assigned to that field.

* Properties are similar to indexers. A property consists of a name, along with get and set accessors. The accessors are used to get and set the value of a variable. The key benefit of a property is that its name can be used in *expressions and assignments* like a normal variable, but in actuality, the get and set accessors are automatically invoked. This is similar to the way that an indexer’s get and set accessors are automatically used. The general form of a property is:

|  |  |
| --- | --- |
| **type name { get {** */\* get accessor code \*/* **}**  **set {** */\* set accessor code \*/* **} }** | * type specifies the type of the property, such as int, and name is the name of the property. After definition, any use of name results in a call to its appropriate accessor. The set receives a parameter called value that contains the ***value being assigned*** to the property. |

* Properties do not define storage locations. Instead, a property typically manages access to a field defined elsewhere. The property itself does not provide this field. Thus, a field must be specified independently of the property. (The exception is the auto-implemented property added by C# 3.0)
* Following defines a property called MyProp, which is used to access the field prop. In this case, the property allows only positive values to be assigned.

|  |  |
| --- | --- |
| using System;  class SimpProp { int prop; *// field being managed by MyProp*  public SimpProp() { prop = 0; }  */\* This is the property that supports access to the private*  *instance variable prop. It allows only positive values. \*/*  public int MyProp { get { return prop; }  set { if(value >= 0) prop = value; } }  } | class PropertyDemo { static void Main() { SimpProp ob = new SimpProp();  Console.WriteLine("Original value of ob.MyProp: " + ob.MyProp);  */\* assign value \*/* ob.MyProp = 100; Console.WriteLine("Value of ob.MyProp: " + ob.MyProp);  *// Can't assign negative value to prop.*  Console.WriteLine("Attempting to assign -10 to ob.MyProp");  ob.MyProp = -10; Console.WriteLine("Value of ob.MyProp: " + ob.MyProp); }} |

* prop is a private field, and a property called MyProp manages access to prop. Because prop is private, it can be accessed ***only through*** MyProp.
* The *property* MyProp is specified as public so that it can be *accessed by code outside* of its class. The get accessor simply returns the *value of prop*. The set accessor sets the value of prop if and only if that value is *positive*.
* The type of property defined by MyProp is called a read-write property because it allows its underlying field to be read and written. It is possible, however, to create read-only and write-only properties. To create a read-only property, define only a get accessor. To define a write-only property, define only a set accessor.
* Auto-Implemented Properties: With C# 3.0, it is possible to implement very simple properties without having to explicitly define the variable managed by the property. Instead, you can let the compiler automatically supply the underlying variable. This is called an auto-implemented property. General form:

***type name { get; set; }***

* Here, type specifies the *type of the property* and name specifies the *name*. Notice that get and set are *immediately followed* by a semicolon. The accessors for an *auto-implemented property* have *no bodies*. This syntax tells the compiler to automatically create a storage location (sometimes referred to as a *backing field* ) that holds the value. This variable is *not named* and is not directly available to you. Instead, it can only be accessed through the property.
* Here is how a property called UserCount is declared using an ***auto-implemented property***: ***public int UserCount { get; set; }***

|  |  |
| --- | --- |
| * Property Restrictions: | * Because a property does not define a storage location, it cannot be passed as a ref or out parameter to a method. * You cannot overload a property. (You can have two different properties that both access the same underlying variable, but it is unusual). * Finally, a property should not *alter the state of the underlying variable* when the get accessor is called. Although this rule is not enforced by the compiler, such an alteration is semantically wrong. A get operation should not create side effects. |

**C#\_3.4 Use an Access Modifier with an Accessor**

By default, the set and get accessors have the same accessibility as the indexer or property of which they are a part. For example, if the property is declared public, then, by default, the get and set accessors are also public.

* It is possible, however, to give set or get its own access modifier, such as private. In all cases, the access modifier for an accessor must be *more restrictive* than the access specification of its property or indexer. For example, here is a property called Max that has its set accessor specified as private:

**class** MyClass { **int** maximum;

**public** **int** Max { **get** { **return** maximum; }

**private** **set** { **if**(value < 0) maximum = -value; **else** maximum = value; } */\* the set accessor is private \*/* }

}

* Now, only code inside MyClass can set the value of Max, but any code can obtain its value.
* Most important use of restricting an accessor’s is found when working with auto-implemented properties. Since, it is not possible to create a read-only or write-only, auto-implemented property because both the get and set accessors must be specified when the auto-implemented property is declared. However, you can gain much the same effect by declaring either get or set as private. For example, this declares what is effectively a *read-only*, *auto-implemented* Length *property* for the FailSoftArray class shown earlier: **public int Length { get; private set; }**
* Because set is private, Length can be ***set only by code within*** its class. Outside its class, an attempt to change Length is illegal. Thus, outside its class, Length is effectively read-only.
* To try the auto-implemented version of Length with FailSoftArray, first remove the len variable. Then, replace each use of len inside FailSoftArray with Length. Here is the updated version of FailSoftArray, along with a Main( ) to demonstrate it:

|  |  |
| --- | --- |
| using System;  class FailSoftArray {  int[] a; *// reference to array*  public bool ErrFlag; *// indicate outcome of last operation*  public int Length { get; private set; } *// auto-implemented, read-only Length property.*  *// Construct array given its size.*  */\* Assignment to Length OK inside FailSoftArray. \*/*  public FailSoftArray(int size) { a = new int[size]; Length = size; }  *// This is the indexer for FailSoftArray.*  public int this[int index] {  get { if(ok(index)) { ErrFlag = false; return a[index]; } else { ErrFlag = true; return 0; } }  set { if(ok(index)) { a[index] = value; ErrFlag = false; } else ErrFlag = true; }  }  *// Return true if index is within bounds.*  private bool ok(int index) { if(index >= 0 & index < Length) return true; return false; }  } | class AutoImpPropertyFSDemo { static void Main() {  FailSoftArray fs = new FailSoftArray(5);  int x;  *// Can read Length.*  for(int i=0; i < (fs.Length); i++) fs[i] = i\*10;  for(int i=0; i < (fs.Length); i++) { x = fs[i];  if(x != -1) Console.Write(x + " "); }  Console.WriteLine();  */\* Assignment to Length outside FailSoftArray is illegal. \*/*  *//* fs.Length = 10; *// Error! Length's set accessor is private.*  }} |

* This version of FailSoftArray works in the same way as the previous version, but it does not contain an explicitly declared *backing field*.