**Indexers**

* As you know, array indexing is performed using the **[ ]** operator.
* It is possible to define 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.

**Creating One-Dimensional Indexers:**

A one-dimensional indexer has this general form:

*element-type* this[int *index*]

{

// The get accessor

get

{

// return the value specifi ed by *index*

}

// The set accessor

set

{

// set the value specifi ed by *index*

}

}

* Here, *element-type* is the element type of the indexer.
* Thus, each element accessed by the indexer will be of type *element-type.* This type corresponds to the element type of an array.
* The parameter *index* receives the index of the element being accessed. Technically, this parameter does not have to be of type **int**, but since indexers are typically used to provide

array indexing, an integer type is customary.

* Inside the body of the indexer two *accessors* are defined that are called **get** and **set**.
* 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 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 an implicit parameter 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 access.
* Here is an example. In the following program, the **FailSoftArray** class implements an array that traps boundary errors, thus preventing runtime exceptions ifthe array is indexed out-of-bounds.
* This is accomplished by encapsulating the array as aprivate member of a class, allowing access to the array only through the indexer.
* With thisapproach, any attempt to access the array beyond its boundaries can be prevented, with such

an attempt failing gracefully (resulting in a “soft landing” rather than a “crash”).

* Since **FailSoftArray** uses an indexer, the array can be accessed using the normal array notation.

// Use an indexer to create a fail-soft array.

using System;

class FailSoftArray

{

int[] a; // reference to underlying array

public int Length; // Length is public

public bool ErrFlag; // indicates 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]

{

// This is the get accessor.

get

{

if(ok(index))

{

ErrFlag = false;

return a[index];

}

else

{

ErrFlag = true;

return 0;

}

}

// This is the set accessor.

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;

}

}

// Demonstrate the fail-soft array.

class FSDemo

{

public static void Main()

{

FailSoftArray fs = new FailSoftArray(5);

int x;

// Show quiet failures.

Console.WriteLine("Fail quietly.");

for(int i=0; i < (fs.Length \* 2); i++)

fs[i] = i\*10;

for(int i=0; i < (fs.Length \* 2); i++)

{

x = fs[i];

if(x != -1)

Console.Write(x + " ");

}

Console.WriteLine();

// Now, display 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");

}

}

}

**The output from the program is shown here:**

Fail quietly.

0 10 20 30 40 0 0 0 0 0

Fail with error reports.

fs[5] out-of-bounds

fs[6] out-of-bounds

fs[7] out-of-bounds

fs[8] out-of-bounds

fs[9] out-of-bounds

0 10 20 30 40 fs[5] out-of-bounds

fs[6] out-of-bounds

fs[7] out-of-bounds

fs[8] out-of-bounds

fs[9] out-of-bounds

* The indexer prevents the array boundaries from being overrun.
* Let’s look closely at each part of the indexer.
* It begins with this line:

public int this[int index] {

* This 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 is shown here:

get

{

if(ok(index))

{

ErrFlag = false;

return a[index];

}

else

{

ErrFlag = true;

return 0;

}

}

* The **get** accessor prevents array boundary errors by first confirming that the index is not out-of-bounds. This range check is performed by the **ok( )** method, which returns true if the index is valid and false otherwise.
* 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. In this version of **FailSoftArray**, 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
* The **set** accessor is shown here. It too prevents a boundary error.

set

{

if(ok(index))

{

a[index] = value;

ErrFlag = false;

}

else

ErrFlag = true;

}

* Here, 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 implicit 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 support 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**.

**Multidimensional Indexers:**

* You can create indexers for multidimensional arrays, too.
* For example, here is a twodimensional fail-soft array.
* Pay close attention to the way that the indexer is declared.

// A two-dimensional fail-soft array.

using System;

class FailSoftArray2D

{

int[,] a; // reference to underlying 2D array

int rows, cols; // dimensions

public int Length; // Length is public

public bool ErrFlag; // indicates 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;

}

// This is the indexer for FailSoftArray2D.

public int this[int index1, int index2]

{

// This is the get accessor.

get

{

if(ok(index1, index2))

{

ErrFlag = false;

return a[index1, index2];

}

else

{

ErrFlag = true;

return 0;

}

}

// This is the set accessor.

set

{

if(ok(index1, index2))

{

a[index1, index2] = value;

ErrFlag = false;

}

else

ErrFlag = true;

}

}

// 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;

}

}

// Demonstrate a 2D indexer.

class TwoDIndexerDemo

{

public static void Main()

{

FailSoftArray2D fs = new FailSoftArray2D(3, 5);

int x;

// Show quiet failures.

Console.WriteLine("Fail quietly.");

for(int i=0; i < 6; i++)

fs[i, i] = i\*10;

for(int i=0; i < 6; i++)

{

x = fs[i,i];

if(x != -1)

Console.Write(x + " ");

}

Console.WriteLine();

// Now, display failures.

Console.WriteLine("\nFail with error reports.");

for(int i=0; i < 6; i++)

{

fs[i,i] = i\*10;

if(fs.ErrFlag)

Console.WriteLine("fs[" + i + ", " + i + "] out-of-bounds");

}

for(int i=0; i < 6; i++)

{

x = fs[i,i];

if(!fs.ErrFlag)

Console.Write(x + " ");

else

Console.WriteLine("fs[" + i + ", " + i + "] out-of-bounds");

}

}

}

**The output from this program is shown here:**

Fail quietly.

0 10 20 0 0 0

Fail with error reports.

fs[3, 3] out-of-bounds

fs[4, 4] out-of-bounds

fs[5, 5] out-of-bounds

0 10 20 fs[3, 3] out-of-bounds

fs[4, 4] out-of-bounds

fs[5, 5] out-of-bounds