**Indexer In C#**

* In a C# program, indexers allow instances of a class or struct to be indexed like arrays.
* Indexers are syntactically similar to properties, but unlike properties, the accessors of indexers accept one or more parameters.
* Indexers allow our object to be used just like an array, or we can say we can index the object using [ ] brackets just like arrays.
* We can say indexers are special type of properties which adds logic that how can array or list store the values.
* Syntax of indexer resembles to properties.
* We can use all access modifiers with indexers and also have return types.
* Indexers are the regular members of a class.
* Indexer is created with the help of this keyword.

**In C# introduce new concept is Indexer. This is very useful for some situation. Let as discuss something about Indexer.**

* Indexer Concept is object act as an array.
* Indexer an object to be indexed in the same way as an array.
* Indexer modifier can be private, public, protected or internal.
* Indexer modifier can be private, public, protected or internal.
* Indexers in C# must have at least one parameter. Else the compiler will generate a compilation error.

**Indexers**

* Indexers are data members that allow you to access data within objects in a way that is similar to accessing arrays.
* Indexers provide faster access to the data within an object as they help in indexing the data.
* Indexers allows you to use the index of an object to access the values within the object.
* Indexers are also known as smart arrays in C#.
* In arrays, you use the index position of an object to access its value.
* The implementation of indexers is similar to properties, except that the declaration of an indexer can contain parameters.
* Indexers allow you to index a class, struct, or an interface.

**An indexer can be defined by specifying the following:**

* An access modifier, which decides the scope of the indexer.
* The return type of the indexer, which specifies the type of value an indexer will return.
* The this keyword, which refers to the current instance of the current class.
* The bracket notation ([]), which consists of the data type and the identifier of the index.
* The open and close curly braces, which contain the declaration of the set and get accessors.

**The following syntax creates an indexer.**

<access\_modifier><return\_type> this [<parameter>]

{

get { // return value }

set { // assign value }

}

**The following code demonstrates the use of indexers.**

class EmployeeDetails {

public string[] empName = new string[2];

public string this[int index] {

get { return empName[index]; }

set { empName[index] = value; }

}

static void Main(string[] args) {

EmployeeDetails objEmp = new EmployeeDetails();

objEmp[0] = “Jack Anderson”;

objEmp[1] = “Kate Jones”;

Console.WriteLine(“Employee Names: “);

for (int i=0; i<2; i++) {

Console.Write(objEmp[i] + “\t”);

}

}

}

**In Above code,**

* The class EmployeeDetails creates an indexer that takes a parameter of type int.
* The instance of the class, objEmp, is assigned values at each index position.
* The set accessor is invoked for each index position.
* The for loop iterates for two times and displays values assigned at each index position using the get accessor.

**Output**

Employee Names : Jack Anderson Kate Jones

**Parameters With Indexers**

* Indexers must have at least one parameter.
* The parameter denotes the index position, using which the stored value at that position is set or accessed.
* Indexers can also have multiple parameters. Such indexers can be accessed like a multi-dimensional array.
* When accessing arrays, you need to mention the object name followed by the array name.
* The value can be accessed by specifying the index position.
* Indexers can be accessed directly by specifying the index number along with the instance of the class.

**Source Code Of Indexers**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace INDEXERS

{

class Employee

{

private int[] Age = new int[3];

public int this[int index]

{

set

{

if(index >= 0 && index < Age.Length)

{

if(value > 0)

{

Age[index] = value;

}

else

{

Console.WriteLine("Value is invalid !!");

}

}

else

{

Console.WriteLine("Invalid index !!");

}

}

get

{

return Age[index];

}

}

public int this[int index, int i]

{

set

{

Age[index] = value + i;

}

get

{

return Age[index];

}

}

}

class Program

{

static void Main(string[] args)

{

Employee emp = new Employee();

emp[0, 1] = 5;

//emp[2] = 5;

Console.WriteLine(emp[0]);

Console.ReadLine();

}

}

}

**Implementing Inheritance With Indexers**

* Indexers can be inherited like other members of the class.
* It means the base class indexers can be inherited by the derived class.

**The following code demonstrates the implementation of inheritance with indexers.**

using System;

class Numbers

{

private int[] num = new int[3];

public int this[int index]

{

get { return num [index];

}

set { num [index] = value; }

}

}

class EvenNumbers : Numbers {

public static void Main() {

EvenNumbers objEven = new EvenNumbers();

objEven[0] = 0;

objEven[1] = 2;

objEven[2] = 4;

for(int i=0; i<3; i++) {

Console.WriteLine(objEven[i]);

}

}

}

**In Above Code,**

* The class Numbers creates an indexer that takes a parameter of type int. The class EvenNumbers inherits the class Numbers.
* The Main() method creates an instance of the derived class EvenNumbers.
* When this instance is assigned values at each index position, the set accessor of the indexer defined in the base class Numbers is invoked for each index position.
* The for loop iterates three times and displays values assigned at each index position using the accessor.
* By inheriting, an indexer in the base class can be reused in the derived class.

**Output**

0

2

4

**Implementing Polymorphism Using Indexers**

* Indexers can implement polymorphism by overriding the base class indexers or by overloading indexers.
* A particular class can include more than one indexer having different signatures. This feature of polymorphism is called overloading.
* Thus, polymorphism allows the indexer to function with different data types of C# and generate customized output.

**The following code demonstrates the implementation of polymorphism with indexers by overriding the base class indexers.**

class EvenNumbers : Numbers

using System;

class Student {

string[] studName = new string[2];

public virtual string this[int index] {

get { return studName[index]; }

set { studName[index] = value; }

}

}

class Result : Student {

string[] result = new string[2];

public override string this[int index] {

get { return base[index]; }

set { base[index] = value; }

}

static void Main(string[] args) {

Result objResult = new Result();

objResult[0] = "First";

objResult[1] = "Pass";

Student objStudent = new Student();

objStudent[0] = "Peter";

objStudent[1] = "Patrick";

for (int i = 0; i< 2; i++) {

Console.WriteLine(objStudent[i] + "\t\t" + objResult[i] + " class");

}

}

}

**In Above Code,**

* The class Student declares an array variable and a virtual indexer.
* The class Result inherits the class Student and overrides the virtual indexer.
* The Main() method declares an instance of the base class Student and the derived class Result.
* When the instance of the class Student is assigned values at each index position, the set accessor of the class Student is invoked.
* When the instance of the class Result is assigned values at each index position, the set accessor of the class Result is invoked.
* This overrides the base class indexer.
* The set accessor of the derived class Result invokes the base class indexer by using the base keyword.
* The for loop displays values at each index position by invoking the get accessors of the appropriate classes.

**Output**

Peter First class  
Patrick Pass class

**Multiple Parameters in Indexers**

* Indexers must be declared with at least one parameter within the square bracket notation [ ].
* Indexers can include multiple parameters.
* An indexer with multiple parameters can be accessed like a multi-dimensional array.
* A parameterized indexer can be used to hold a set of related values.
* For example, it can be used to store and change values in multi-dimensional arrays.

**The following code demonstrates how multiple parameters can be passed to an indexer.**

using System;

class Account {

string[,] accountDetails = new string[4, 2];

public string this[int pos, int column] {

get { return (accountDetails[pos, column]); }

set { accountDetails[pos, column] = value; }

}

static void Main(string[] args)

{

Account objAccount = new Account();

string[] id = new string[3] { "1001", "1002", "1003" };

string[] name = new string[3] { "John", "Peter", "Patrick" };

int counter = 0;

for (int i = 0; i< 3; i++)

{

for (int j = 0; j < 1; j++)

{

objAccount[i, j] = id[counter];

objAccount[i, j+1] = name[counter++];

}

}

Console.WriteLine("ID Name");

Console.WriteLine();

for (int i = 0; i< 4; i++)

{

for (int j = 0; j < 2; j++)

{

Console.Write(objAccount[i, j]+ " ");

}

Console.WriteLine();

}

}

}

**In Above Code,**

* The class Account creates an array variable accountDetails having four rows and two columns.
* A parameterized indexer is defined to enter values in the array accountDetails.
* The indexer takes two parameters, which defines the positions of the values that will be stored in an array.
* The Main() method creates an instance of the Account class.
* This instance is used to enter values in the accountDetails array using a for loop.
* This invokes the set accessor of the indexer which assigns value in the array.
* A for loop displays the customer ID and name that is stored in an array which invokes the get accessor.

**Output**

ID Name  
1001 John  
1002 Peter  
1003 Patrick

**Indexers in Interfaces**

* Indexers can also be declared in interfaces.
* The accessors of indexers declared in interfaces differ from the indexers declared within a class.
* The set and get accessors declared within an interface do not use access modifiers and do not contain a body.
* An indexer declared in the interface must be implemented in the class implementing the interface.
* This enforces reusability and provides the flexibility to customize indexers.

**The following code demonstrates the implementation of interface indexers.**

using System;

public interface Idetails {

string this[int index]

{ get; set; }

}

class Students :Idetails {

string [] studentName = new string[3];

int[] studentID = new int[3];

public string this[int index] {

get { return studentName[index]; }

set { studentName[index] = value; }

}

static void Main(string[] args) {

Students objStudent = new Students();

objStudent[0] = "James";

objStudent[1] = "Wilson";

objStudent[2] = "Patrick";

Console.WriteLine("Student Names");

Console.WriteLine();

for (int i = 0; i< 3; i++) {

Console.WriteLine(objStudent[i]);

}

}

}

**In Above Code,**

* The interface Idetails declares a read-write indexer.
* The Students class implements the Idetails interface and implements the indexer defined in the interface.
* The Main() method creates an instance of the Students class and assigns values at different index positions.
* This invokes the set accessor.
* The for loop displays the output by invoking the get accessor of the indexer.