Session: 9

Properties and Indexers

Objectives

- Define properties in C#
- Explain properties, fields, and methods
- Explain indexers

Properties in C#

- Access modifiers such as `public`, `private`, `protected`, and `internal` control accessibility of fields and methods in C#.
  - `public` fields: accessible by other classes
  - `private` fields: accessible only by the class in which they are declared

- **Properties** in C# allow you to set and retrieve values of fields declared with any access modifier in a secured manner.

Example

- Consider fields that store names and IDs of employees.
- You can create properties for these fields to ensure accuracy and validity of values stored in them.

Applications of Properties

- Properties:
  - allow to protect a field in the class by reading and writing to the field through a property declaration.
  - allow to access private fields, which would otherwise be inaccessible.
  - can validate values before allowing you to change them and also perform specified actions on those changes.
  - ensure security of private data.
  - support abstraction and encapsulation by exposing only necessary actions and hiding their implementation.
- The following syntax is used to declare a property in C#.

Syntax

```
<access_modifier><return_type><PropertyName>
{
    //body of the property
}
```

where,

- `access_modifier`: Defines the scope of access for the property, which can be `private`, `public`, `protected`, or `internal`.
- `return_type`: Determines the type of data the property will return.
- `PropertyName`: Is the name of the property.

get and set Accessors 1-3

- Property accessors allow you to read and assign a value to a field by implementing get and set accessors as follows:

The get accessor

- The get accessor is used to read a value and is executed when the property name is referred.
- It does not take any parameter and returns a value that is of the return type of the property.

The set accessor

- The set accessor is used to assign a value and is executed when the property is assigned a new value using the equal to (=) operator.
- This value is stored in the private field by an implicit parameter called value (keyword in C#) used in the set accessor.

Syntax

```
<access_modifier><return_type>PropertyName>
{
     get
     {
        // return value
     }
     set
     {
        // assign value
     }
}
```

get and set Accessors 2-3

- The following code demonstrates the use of the `get` and `set` accessors.

Snippet

```
using System;
class SalaryDetails
{
      private string _empName;
      public string EmployeeName
      {
      get
      {
      return _empName;
      }
      set
      {
      _empName = value;
      }
      }
      static void Main (string [] args)
      {
          SalaryDetails objSal = new SalaryDetails();
          objSal.EmployeeName = "Patrick Johnson";
          Console.WriteLine("Employee Name: " + objSal.EmployeeName);
      }
}
```

- In the code:
  - The class **SalaryDetails** creates a private variable **_empName** and declares a property called **EmployeeName**.
  - The instance of the **SalaryDetails** class, **objSal**, invokes the property **EmployeeName** using the dot (**.**) operator to initialize the value of employee name. This invokes the `set` accessor, where the value keyword assigns the value to **_empName**.
  - The code displays the employee name by invoking the property name.
  - This invokes the `get` accessor, which returns the assigned employee name. This invokes the `set` accessor, where the `value` keyword assigns the value to **_empName**.

Output

```
Employee Name: Patrick Johnson
```

Categories of Properties 1-10

- Properties are broadly divided into three categories:

- **Read-Only Property**:
  - The read-only property allows you to retrieve the value of a private field. To create a read-only property, you should define the `get` accessor.
  - The following syntax creates a read-only property.

Syntax

```
<access_modifier><return_type><PropertyName>{
get
{
// return value
}
}
```

- The following code demonstrates how to create a read-only property.

Snippet

```
using System;
class Books {
      string _bookName;
      long _bookID;
      public Books(string name, int value){
      _bookName = name;
      _bookID = value;
      }
      public string BookName {
      get{ return _bookName; }
      }
      public long BookID {
      get { return _bookID; }
      }
}
class BookStore {
       static void Main(string[] args) {
       Books objBook = new Books("Learn C# in 21 Days", 10015);
       Console.WriteLine("Book Name: " + objBook.BookName);
       Console.WriteLine("Book ID: " + objBook.BookID);
      }
}
```

- In the code:
  - The **Books** class creates two read-only properties that returns the name and ID of the book.
  - The class **BookStore** defines a `Main()` method that creates an instance of the class **Books** by passing the parameter values that refer to the name and ID of the book.
  - The output displays the name and ID of the book by invoking the `get` accessor of the appropriate read-only properties.

Output

```
Book Name: Learn C# in 21 Days
Book ID: 10015
```

- **Write-Only Property**:
  - The write-only property allows you to change the value of a private field.
  - To create a write-only property, you should define the `set` accessor.
  - The following syntax creates a write-only property.

Syntax

```
<access_modifer><return_type><PropertyName>
{
    set
    {
    // assign value
    }
}
```

- The following code demonstrates how to create a write-only property.

Snippet

```
using System;
class Department {
      string _deptName;
      int _deptID;
      public string DeptName{
      set { _deptName = value; }
      }
      public intDeptID {
      set { _deptID = value;}
      }
      public void Display() {
      Console.WriteLine("Department Name: " + _deptName);
      Console.WriteLine("Department ID: " + _deptID);
      }
}
class Company {
       static void Main(string[] args) {
      Department objDepartment = new Department();
       objDepartment.DeptID = 201;
       objDepartment.DeptName = "Sales";
       objDepartment.Display();
       }
}
```

- In the code:
  - The **`Department`** class consists of two write-only properties.
  - The **`Main()`** method of the class **`Company`** instantiates the class **`Department`** and this instance invokes the `set` accessor of the appropriate write-only properties to assign the department name and its ID.
  - The **`Display()`** method of the class **`Department`** displays the name and ID of the department.

Output

```
Department Name: Sales
Department ID: 201
```

- **Read-Write Property**:
  - The read-write property allows you to set and retrieve the value of a private field. To create a read-write property, you should define the `set` and `get` accessors.
  - The following syntax creates a read-write property.

Syntax

```
<access_modifer><return type><PropertyName>
{
      get
      {
      // return value
      }
      set
      {
      // assign value
      }
}
```

- The following code demonstrates how to create a read-write property.

Snippet

```
using System;
class Product {
string _productName;
int _productID;
float _price;
public Product(string name, intval) {
_productName = name;
_productID = val;
}
public float Price {
get { return _price;}
set { if (value < 0) { _price = 0; }
else { _price = value; }
}
}
public void Display() {
Console.WriteLine("Product Name: " + _productName);
Console.WriteLine("Product ID: " + _productID);
Console.WriteLine("Price: " + _price + "$");
}
}
class Goods {
static void Main(string[] args) {
      Product objProduct = new Product("Hard Disk", 101);
      objProduct.Price = 345.25F;
      objProduct.Display();
}
}
```

- In the code:
  - The class **`Product`** creates a read-write property **`Price`** that assigns and retrieves the price of the product based on the if statement.
  - The **`Goods`** class defines the **`Main()`** method that creates an instance of the class Product by passing the values as parameters that are name and ID of the product.
  - The **`Display()`** method of the class **`Product`** is invoked that displays the name, ID, and price of the product.

Output

```
Product Name: Hard Disk
Product ID: 101
Price: 345.25$
```

- Properties can be further classified as static, abstract, and boolean properties.

Static Properties 1-3

- The static property is:
  - declared by using the `static` keyword.
  - accessed using the class name and thus, belongs to the class rather than just an instance of the class.
  - by a programmer without creating an instance of the class.
  - used to access and manipulate static fields of a class in a safe manner.
- The following code demonstrates a class with a static property.

Snippet

```
using System;
class University
{
      private static string _department;
      private static string _universityName;
      public static string Department
      {
      get
      {
      return _department;
      }
      set
      {
      _department = value;
      }
```

Snippet

```
        }
        public static string UniversityName
        {
        get { return _universityName; }
        set { _universityName = value; }
        }
}
class Physics
{
        static void Main(string[] args)
        {
        University.UniversityName = "University of Maryland";
        University.Department = "Physics";
        Console.WriteLine("University Name: " + University.UniversityName);
        Console.WriteLine("Department name: " + University.Department);
        }
}
```

In the code:

- The class **`University`** defines two static properties **`UniversityName`** and **`DepartmentName`**.
- The `Main()` method of the class `Physics` invokes the static properties **`UniversityName`** and **`DepartmentName`** of the class **`University`** by using the dot (.) operator.
- This initializes the static fields of the class by invoking the `set` accessor of the appropriate properties.
- The code displays the name of the university and the department by invoking the `get` accessor of the appropriate properties.

Output

```
University Name: University of Maryland
Department name: Physics
```

Abstract properties:

- Declared by using the `abstract` keyword.
- Contain only the declaration of the property without the body of the `get` and `set` accessors (which do not contain any statements and can be implemented in the derived class).
- Are only allowed in an abstract class.
- Are used:
  - when it is required to secure data within multiple fields of the derived class of the abstract class.
  - to avoid redefining properties by reusing the existing properties.
- The following code demonstrates a class that uses an abstract property.

Snippet

```
using System;
public abstract class Figure {
     public abstract float DimensionOne {
          set;
     }
     public abstract float DimensionTwo {
     set; }
}
class Rectangle : Figure {
     float _dimensionOne;
```

Snippet

```
        float _dimensionTwo;
        public override float DimensionOne {
        set {
        if (value <= 0){
        _dimensionOne = 0;
        }
        else {
        _dimensionOne = value;
        }
      }
    }
    public override float DimensionTwo {
        set {
        if (value <= 0)
        {
        _dimensionTwo = 0;
        }
        else {
        _dimensionTwo = value;
        }
        }
    }
    float Area() {
    return _dimensionOne * _dimensionTwo;
    }
    static void Main(string[] args) {
```

Snippet

```
Rectangle objRectangle = new Rectangle();
objRectangle.DimensionOne = 20;
objRectangle.DimensionTwo = 4.233F;
Console.WriteLine("Area of Rectangle: " + objRectangle.Area());
}
}
```

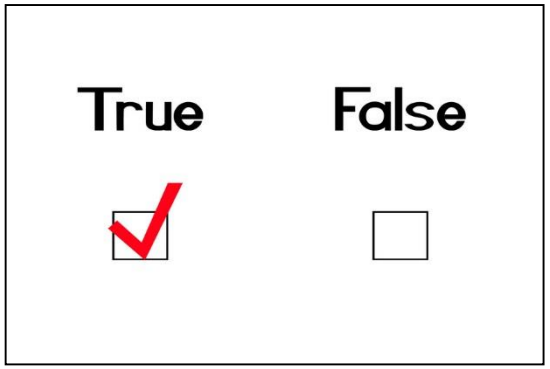
- In the code:
  - The abstract class **`Figure`** declares two write-only abstract properties, **`DimensionOne`** and **`DimensionTwo`**.
  - The class **`Rectangle`** inherits the abstract class **`Figure`** and overrides the two abstract properties **`DimensionOne`** and **`DimensionTwo`** by setting appropriate dimension values for the rectangle.
  - The `Area()` method calculates the area of the rectangle.
  - The `Main()` method creates an instance of the derived class **`Rectangle`**.
  - This instance invokes the properties, **`DimensionOne`** and **`DimensionTwo`**, which, in turn, invokes the `set` accessor of appropriate properties to assign appropriate dimension values.
  - The code displays the area of the rectangle by invoking the `Area()` method of the **`Rectangle`** class.

Output

```
Area of Rectangle: 84.66
```

Boolean Properties

- A boolean property is declared by specifying the data type of the property as `bool`.
- Unlike other properties, the boolean property produces only true or false values.
- While working with boolean property, a programmer needs to be sure that the `get` accessor returns the boolean value.



Implementing Inheritance 1-3

- Properties can be inherited just like other members of the class.
- The base class properties are inherited by the derived class.
- The following code demonstrates how properties can be inherited.

Snippet

```
using System;
class Employee {
      string _empName;
      int _empID;
      float _salary;
      public string EmpName {
      get { return _empName; }
      set { _empName = value; }
      }
      public int EmpID {
      get { return _empID; }
      set { _empID = value; }
      }
      public float Salary {
      get { return _salary; }
      set {
      if (value < 0) {
      _salary = 0;
```

Snippet

```
            }
            else
            {
            _salary = value;
            }
        }
    }
}
class SalaryDetails : Employee {
        static void Main(string[] args){
        SalaryDetails objSalary = new SalaryDetails();
        objSalary.EmpName = "Frank";
        objSalary.EmpID = 10;
        objSalary.Salary = 1000.25F;
        Console.WriteLine("Name: " + objSalary.EmpName);
        Console.WriteLine("ID: " + objSalary.EmpID);
        Console.WriteLine("Salary: " + objSalary.Salary + "$");
        }
}
```

- In the code:
  - The class **Employee** creates three properties to set and retrieve the employee name, ID, and salary respectively.
  - The class **SalaryDetails** is derived from the **Employee** class and inherits its public members.
  - The instance of the **SalaryDetails** class initializes the value of the **_empName**, **_empID**, and **_salary** using the respective properties **EmpName**, **EmpID**, and **Salary** of the base class **Employee**.
  - This invokes the set accessors of the respective properties.
  - The code displays the name, ID, and salary of an employee by invoking the `get` accessor of the respective properties.
  - By implementing inheritance, the code implemented in the base class for defining property can be reused in the derived class.

Output

```
Name: Frank
ID: 10
Salary: 1000.25$
```

Auto-Implemented Properties 1-3

- C# provides an alternative syntax to declare properties where a programmer can specify a property in a class without explicitly providing the `get` and `set` accessors.
- Such properties are called auto-implemented properties and results in more concise and easy-to-understand programs.
- For an auto-implemented property, the compiler automatically creates:
  - a private field to store the property variable.
  - the corresponding `get` and `set` accessors.
- The following syntax creates an auto-implemented property.

Syntax

```
public string Name { get; set; }
```

- The following code uses auto-implemented properties.

Snippet

```
class Employee
{
     public string Name { get; set; }
     public int Age { get; set; }
```

Auto-Implemented Properties 2-3

Snippet

```
        public string Designation { get; set; }
        static void Main (string [] args)
        {
        Employee emp = new Employee();
        emp.Name = "John Doe";
        emp.Age = 24;
        emp.Designation = "Sales Person";
        Console.WriteLine("Name: {0}, Age: {1}, Designation: {2}",
        emp.Name, emp.Age, emp.Designation);
        }
}
```

The code declares three auto-implemented properties: **`Name`**, **`Age`**, and **`Designation`**. The `Main()` method first sets the values of the properties and then, retrieves the values and writes to the console.

Output

```
Name: John Doe, Age: 24, Designation: Sales Person
```

Auto-Implemented Properties 3-3

- Like normal properties, auto-implemented properties can be declared to be read-only and write-only.
- The following code declares read-only and write-only properties.

Snippet

```
public float Age { get; private set; }
public int Salary { private get; set; }
```

In the code:

- The `private` keyword before the `set` keyword declares the **`Age`** property as read-only.
- In the second property declaration, the `private` keyword before the `get` keyword declares the **`Salary`** property as write-only.

Object Initializers 1-2

- In C#, programmers can use object initializers to initialize an object with values without explicitly calling the constructor.
- The declarative form of object initializers makes the initialization of objects more readable in a program.
- When object initializers are used in a program, the compiler first accesses the default instance constructor of the class to create the object and then performs the initialization.

Object Initializers 2-2

- The following code uses object initializers to initialize an **Employee** object.

Snippet

```
class Employee {
    public string Name { get; set; }
    public int Age { get; set; }
    public string Designation { get; set; }
    static void Main (string [] args) {
    Employee emp1 = new Employee {
        Name = "John Doe",
        Age = 24,
        Designation = "Sales Person"
    };
Console.WriteLine("Name: {0}, Age: {1}, Designation: {2}", emp1.Name,
emp1.Age, emp1.Designation);
}
}
```

- The code creates three auto-implemented properties in an **Employee** class.
- The **Main()** method uses an object initializer to create an **Employee** object initialized with values of its properties.

Output

- `Name: John Doe, Age: 24, Designation: Sales Person`

Implementing Polymorphism 1-2

- Properties can implement polymorphism by overriding the base class properties in the derived class.
- However, properties cannot be overloaded.
- The following code demonstrates the implementation of polymorphism by overriding the base class properties.

Snippet

```
using System;
class Car {
string _carType;
public virtual string CarType {
get { return _carType; }
set { _carType = value; }
}
}
class Ferrari : Car {
string _carType;
public override string CarType{
get { return base.CarType; }
set {
base.CarType = value;
_carType = value;
}
}
static void Main(string[] args)
{
Car objCar = new Car();
objCar.CarType = "Utility Vehicle";
```

Implementing Polymorphism 2-2

Snippet

```
Ferrari objFerrari = new Ferrari();
objFerrari.CarType = "Sports Car";
Console.WriteLine("Car Type: " + objCar.CarType);
Console.WriteLine("Ferrari Car Type: " + objFerrari.CarType);
}
}
```

- The class **Car** declares a virtual property **CarType**.
- The class **Ferrari** inherits the base class **Car** and overrides the property **CarType**.
- The **Main()** method of the class **Ferrari** declares an instance of the base class **Car**.
- When the **Main()** method creates an instance of the derived class **Ferrari** and invokes the derived class property **CarType**, the virtual property is overridden.
- However, since the `set` accessor of the derived class invokes the base class property **CarType** using the `base` keyword, the output displays both the car type and the Ferrari car type.
- Thus, the code shows that the properties can be overridden in the derived classes and can be useful in giving customized output.

Output

- `Car Type: Utility Vehicle`
- `Ferrari Car Type: Sports Car`

Properties, Fields, and Methods 1-2

- A class in a C# program can contain a mix of properties, fields, and methods, each serving a different purpose in the class.
- It is important to understand the differences between them in order to use them effectively in the class.
- Properties are similar to fields as both contain values that can be accessed. However, there are certain differences between them.
- The following table shows the differences between properties and fields.

Properties	Fields
Properties are data members that can assign and retrieve values.	Fields are data members that store values.
Properties cannot be classified as variables and therefore, cannot use the ref and out keywords.	Fields are variables that can use the ref and out keywords.
Properties are defined as a series of executable statements.	Fields can be defined in a single statement.
Properties are defined with two accessors or methods, the get and set accessors.	Fields are not defined with accessors.
Properties can perform custom actions on change of the field's value.	Fields are not capable of performing any customized actions.

Properties, Fields, and Methods 2-2

- The implementation of properties covers both, the implementation of fields and the implementation of methods.
- This is because properties contain two special methods and are invoked in a similar manner as fields.
- The differences between properties and methods are listed in the table.

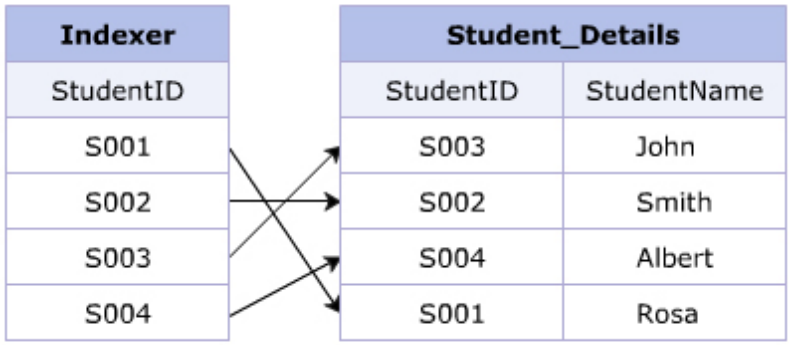
Properties	Methods
Properties represent characteristics of an object.	Methods represent the behavior of an object.
Properties contain two methods which are automatically invoked without specifying their names.	Methods are invoked by specifying method names along with the object of the class.
Properties cannot have any parameters.	Methods can include a list of parameters.
Properties can be overridden but cannot be overloaded.	Methods can be overridden as well as overloaded.

Indexers 1-2

- 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.

Example

- Consider a high school teacher who wants to go through the records of a particular student to check the student's progress.
- Calling the appropriate methods every time to set and get a particular record makes the task tedious.
- Creating an indexer for student ID:
  - makes the task of accessing the record much easier as indexers use index position of the student ID to locate the student record.



Indexers 2-2

- Indexers:
  - are data members that allow you to access data within objects in a way that is similar to accessing arrays.
  - provide faster access to the data within an object as they help in indexing the data.
  - allows you to use the index of an object to access the values within the object.
  - 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.

Declaration of Indexers 1-2

- 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.

Syntax

```
<access_modifier><return_type> this [<parameter>]
{
get { // return value }
set { // assign value }
}
```

Declaration of Indexers 2-2

- The following code demonstrates the use of indexers.

Snippet

```
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");
      }
      }
}
```

- 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

- 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.

Implementing Inheritance 1-2

- 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.

Snippet

```
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]);
      }
   }
}
```

Implementing Inheritance 2-2

- In the 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 1-3

- 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.

Snippet

```
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] {
```

Implementing Polymorphism Using Indexers 2-3

Snippet

```
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");
}
}
}
```

Implementing Polymorphism Using Indexers 3-3

- In the 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 1-3

- 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.

Snippet

```
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++)
```

Multiple Parameters in Indexers 2-3

Snippet

```
        {
        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();
        }
        }
}
```

Multiple Parameters in Indexers 3-3

- In the 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 1-3

- 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.

Indexers in Interfaces 2-3

- The following code demonstrates the implementation of interface indexers.

Snippet

```
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]);
    }
}
}
```

Indexers in Interfaces 3-3

- In the 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.

Difference between Properties and Indexers

- Indexers are syntactically similar to properties.
- However, there are certain differences between them.
- The following table lists the differences between properties and indexers.

Properties	Indexers
Properties are assigned a unique name in their declaration.	Indexers cannot be assigned a name and use the this keyword in their declaration.
Properties are invoked using the specified name.	Indexers are invoked through an index of the created instance.
Properties can be declared as `static`.	Indexers can never be declared as `static`.
Properties are always declared without parameters.	Indexers are declared with at least one parameter.
Properties cannot be overloaded.	Indexers can be overloaded.
Overridden properties are accessed using the `syntax base.Prop`, where `Prop` is the name of the property.	Overridden indexers are accessed using the syntax `base[indExp]`, where `indExp` is the list of parameters separated by commas.

Summary

- Properties protect the fields of the class while accessing them.
- Property accessors enable you to read and assign values to fields.
- A field is a data member that stores some information.
- Properties enable you to access the private fields of the class.
- Methods are data members that define a behavior performed by an object.
- Indexers treat an object like an array, thereby providing faster access to data within the object.
- Indexers are syntactically similar to properties, except that they are defined using the this keyword along with the bracket notation ([ ]).