**Properties In C#**

* Properties allow you to control the accessibility of a class variables, and are the recommended way to access variables from the outside in c#.
* Properties in C# allow you to set and retrieve values of fields declared with any access modifier in a secured manner.
* A property is much like a combination of a variable and a method - it can't take any parameters, but you are able to process the value before it's assigned to our returned.
* Properties are like data fields (variables), but have logic behind them.
* From the outside, they look like any other member variable.
  + But they act like a member function.
* Defined like a field, with “get” and “set” accessors code added.
* Properties are also used for encapsulation.

**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.

**Properties:**

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

**Types Of Properties In C#**

* Read / write properties
* Read only properties
* Write only properties
* Auto implemented properties

**The following syntax is used to declare a property in C#.**

<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**

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

}

}

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

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 Above 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

**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.**

<access\_modifier><return\_type><PropertyName>{

get

{

// return value

}

}

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

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 Above 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.**

<access\_modifer><return\_type><PropertyName>

{

set

{

// assign value

}

}

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

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 Above 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.**

<access\_modifer><return type><PropertyName>

{

get

{

// return value

}

set

{

// assign value

}

}

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

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 Above 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**

* The static property is used to access and manipulate static fields of a class in a safe manner.
* The static property declared by using the static keyword.
* The static property accessed using the class name and thus, belongs to the class rather than just an instance of the class.
* The static property called by a programmer without creating an instance of the class.
* We cannot initialize instance fields within static property.
* The static property accessed using the class name and thus, belongs to the class rather than just an instance of the class.
* The static property is used by a programmer without creating an instance of the class.
* The static property is used to access and manipulate static fields of a class in a safe manner.

**The following code demonstrates a class with a static property.**

using System;

class University

{

private static string \_department;

private static string \_universityName;

public static string Department

{

get

{

return \_department;

}

set

{

\_department = value;

}

}

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 Above 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**

* The word Abstract means incomplete, which means no implementation in programming.
* This is a duty of child class to implement an abstract member of their parent class.
* Abstract property declared by using the abstract keyword.
* Abstract properties 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).
* Abstract properties are only allowed in an abstract class.

**Abstract Properties 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.**

using System;

public abstract class Figure {

public abstract float DimensionOne {

  set;

}

public abstract float DimensionTwo {

set; }

}

class Rectangle : Figure {

float \_dimensionOne;

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) {

Rectangle objRectangle = new Rectangle();

objRectangle.DimensionOne = 20;

objRectangle.DimensionTwo = 4.233F;

Console.WriteLine("Area of Rectangle: " + objRectangle.Area());

}

}

**In Above 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.

**Source Code Of Properties**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace PROPERTIES

{

class Student

{

public string firstName { get; private set; }

public string lastName { get; private set; }

public Student(string fname, string lname)

{

firstName = fname;

lastName = lname;

}

}

class Program

{

static void Main(string[] args)

{

Student s = new Student("Adil","Mehmood");

Console.WriteLine(s.firstName);

Console.WriteLine(s.lastName);

//s.StdId = 23;

//s.Name = "Adil";

//s.FName = "Mehmood";

//Console.WriteLine(s.StdId);

//Console.WriteLine(s.Name);

//Console.WriteLine(s.FName);

Console.ReadLine();

}

}

}

**Source Code Of Abstract Properties**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Abstract\_Properties

{

abstract class person

{

public abstract uint Id { get; set; }

public abstract string Name { get; set; }

}

class student : person

{

uint StudentId;

string StudentName;

public override uint Id

{

set

{

if (value == 0)

{

Console.WriteLine("Id cannot be Zero !!");

}

else

{

this.StudentId = value;

}

}

get

{

return this.StudentId;

}

}

public override string Name

{

set

{

if(string.IsNullOrEmpty(value))

{

Console.WriteLine("Name cannot be null or empty !!");

}

else

{

this.StudentName = value;

}

}

get

{

return this.StudentName;

}

}

}

class Program

{

static void Main(string[] args)

{

student Anas = new student();

Anas.Id = 22;

Anas.Name = "Anas Qureshi";

Console.WriteLine(Anas.Id);

Console.WriteLine(Anas.Name);

Console.ReadLine();

}

}

}

**Source Code Of Static Properties**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace STATIC\_PROPERTY

{

class university

{

private static string UniversityName;

private static string DepartmentName;

public static string \_UniversityName

{

set

{

if (string.IsNullOrEmpty(value))

{

Console.WriteLine("You cannot enter null or empty value in University NAme !!");

}

else

{

UniversityName = value;

}

}

get

{

return UniversityName;

}

}

public static string \_DepartmentName {

set

{

if(string.IsNullOrEmpty(value))

{

Console.WriteLine("You are not allowed to insert null or empty value in department Name !!");

}

else

{

DepartmentName = value;

}

}

get

{

return DepartmentName;

}

}

}

class Program

{

static void Main(string[] args)

{

university.\_UniversityName = "Mehran University Jamshoro";

university.\_DepartmentName = "Software Engineering";

Console.WriteLine("University Name is: {0}",university.\_UniversityName);

Console.WriteLine("University Department Name is: {0}", university.\_DepartmentName);

Console.ReadLine();

}

}

}