Session 8: Properties, Indexers, and Record Types

Objectives

- Define properties in C#
- Explain properties, fields, and methods
- Explain indexers
- Describe init only setters and record types

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.
  - Can validate values before allowing you to change them.
  - Ensure security of private data.

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

- 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.

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.

Syntax

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

Init Only Setters 1-2

- In earlier versions of C#, to create immutable properties, there were two ways, namely, using private setters and defining setters that check current value of the property and then, set it if there was no value.
- To avoid problems that were occurring in these approaches, C# 9.0 introduced an 'Init only setter'.
- These setters can be used to create an immutable field without any complexity, while still allowing them to be set in both the constructor and during Nested Object Creation.

Features			
Init accessor is a variant of `set` accessor that can be called during object initialization.	In C# 9.0, `init` accessors can be created instead of set accessors for properties and indexers.	Upon using `init` keyword, it restricts a property to only being used by a constructor.	In other words, `init` only is used to make an object immutable. Code Snippet demonstrates usage of `init` setters

Init Only Setters 2-2

- Following code demonstrates how to use `init` setters:

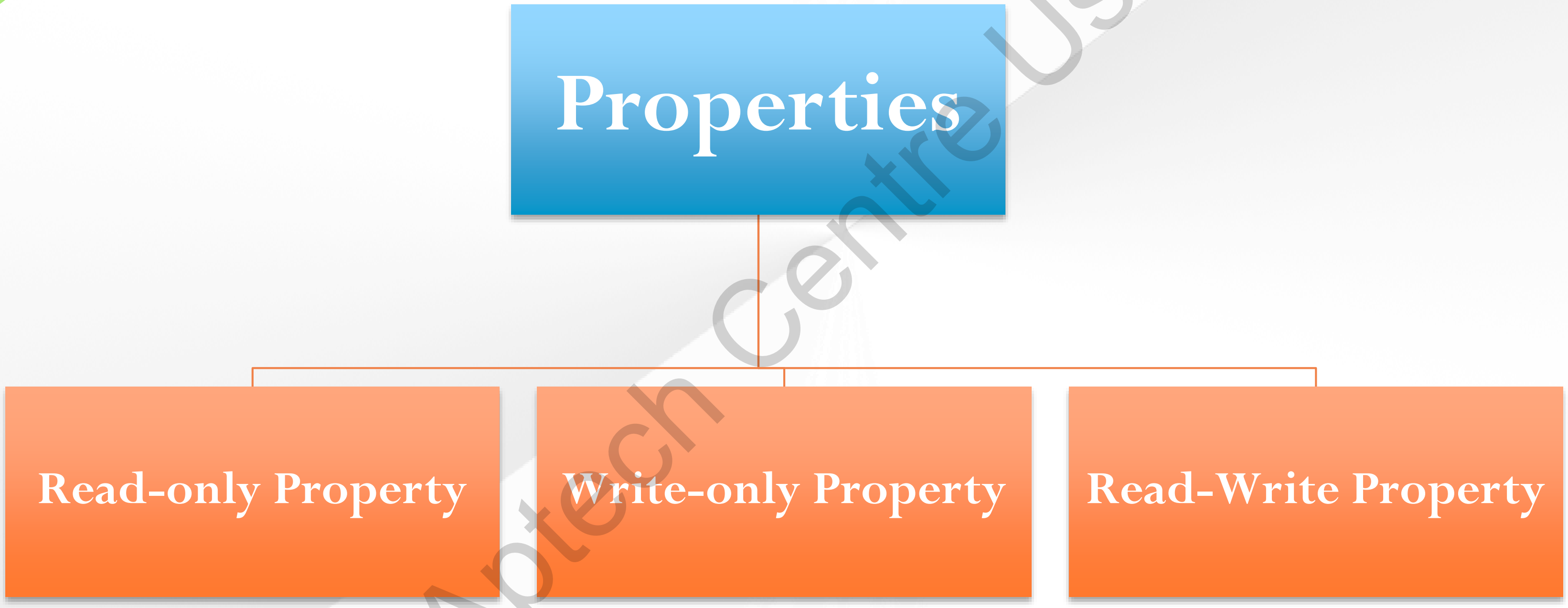
Snippet

```
public class Person {
public string? FirstName { get; init; }
public string? LastName { get; init; }
}
```

- In this code, `init` keyword is used. Upon using this keyword, one cannot change the **`FirstName`** and `LastName` values. If these variables are changed, then the code will show an error.

Categories of Properties 1-4

- Properties are broadly divided into three categories:



Categories of Properties 2-4

- 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.
  - Following syntax creates a read-only property:

Syntax

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

Categories of Properties 3-4

- 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.
  - Following syntax creates a write-only property:

Syntax

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

Categories of Properties 4-4

- 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.
  - Following syntax creates a read-write property:

Syntax

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

Static Properties 1-2

- 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.
- 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;
      }
      }
      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);
      }
}
```

Static Properties 2-2

- 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 1-2

- 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.

- 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;
```

Abstract Properties 2-2

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

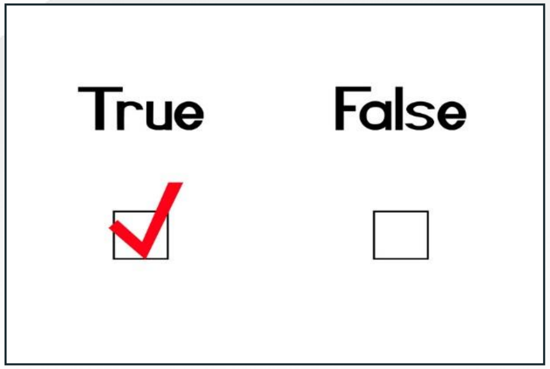
Output

```
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 must ensure that the `get` accessor returns the boolean value.

.



Implementing Inheritance 1-2

- Properties can be inherited just like other members of the class.
- The base class properties are inherited by the derived class.
- 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;
```

Implementing Inheritance 2-2

```
            }
            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 + "$");
        }
}
```

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.
- Following syntax creates an auto-implemented property:

Syntax

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

- 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.
- 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

- 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.
- 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`**.

Output

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

Record Types – Immutability 1-3

- Record types provide immutability in C#. Before C# 9.0, C# did not support immutability features.
- With inclusion of record types and init setters, C# now supports immutability that helps to provide security and improve memory management.
- A `record` type is a light-weight class or data type that has read-only properties. It cannot be changed after it has been created.
- Features of record types are as follows:
  - `record` keyword is used to define record types as shown:

    ```
    public record Student (string subject, int marks);
    ```

  - Record type can also be created using mutable properties.
  - Following code demonstrates record type using mutable properties:

Snippet

```
public record Student {
public string Subject { get; set; }
public int marks {get; set; }
};
```

Record Types – Immutability 2-3

- In C#, there are two ways in which space in memory is allocated. It is either based on value type or reference type of data.
- In value type, the data type holds the value of variable on its own memory. Whereas, in reference type, the variable value is not stored in its own memory rather, it contains a pointer that points to the memory location where the data is stored. Record types offer concise syntax for creating reference types that have immutable properties.
- Immutability in record type is defined as a property where once an object or file is created, it cannot be changed. While the records can be mutable, it can be used to create immutable data models easily. As the record type is immutable, it has read only properties.
- Unlike structure type, `record` type supports inheritance. A record cannot be inherited from a class and a class cannot be inherited from a record however, a `record` can inherit the data from another record. Record type is lightweight as compared to structure type.
- Note: A structure type includes collection of different types of data under a single unit.

Record Types – Immutability 3-3

- A record can be declared using `record` keyword.
- Following code demonstrates declaration of record:

Snippet

```
public record Player {
 public string Name { get; set; }
 public string sports { get; set; }
 public int matchPlayed { get; set; }
 public int MatchesWon { get; set; }
 public string Country { get; set; }
}
```

- In this code, the record `Player` is not immutable.
- Following code demonstrates immutable record:

Snippet

```
public record Player {
 public string Name { get; init; }
 public string sports { get; init; }
public int matchPlayed { get; init; }
 public int MatchesWon { get; init; }
 public string Country { get; init; }
}
```

Record Types – Value Equality 1-2

- When two variables of a record type are equal and all the properties of variables match, it is called Value equality.
- In C#, the Object class has two methods – `Equals()` and `ReferenceEquals()`. `Equals()` is based on a process that checks the data, whether it is equal or not whereas, `ReferenceEquals()` checks whether the object is same or not.
- Reference equality is required for some data models. For example, in the Entity Framework Core database mechanism, developer must ensure only one instance of the entity should be available throughout. Thus, this is a case where reference equality can play a key role by comparing instances.
- Another example to illustrate reference equality involves comparing two objects, **`obj1`** and **`obj2`**.`ReferenceEquals()` returns true if **`obj1`** is the same instance as **`obj2`** or if both are null otherwise, it returns false.
- Record overrides these two methods to check the equality of values one by one recursively.

Record Types – Value Equality 2-2

- Following code demonstrates record override:

Snippet

```
using System;
namespace ConsoleApplication1 {
public record Cricketer(string FirstName, string LastName, string[]
HighScore) {
 public static void Main() {
          var HighScore = new string[2];
          Cricketer Cricketer1 = new("Steve", "Smith", HighScore);
          Cricketer Cricketer2 = new("Steve", "Smith", HighScore);
          Console.WriteLine(Cricketer1 == Cricketer2);
          Cricketer1.HighScore[0] = "183";
          Console.WriteLine(Cricketer1 == Cricketer2);
          Console.WriteLine(Cricketer1.Equals(Cricketer2));
          Console.WriteLine(ReferenceEquals(Cricketer1, Cricketer2));
 } } }
```

Output

```
True
True
True
False
```

Record Types – Nondestructive Mutation 1-3

- Nondestructive mutation is a concept used to mutate immutable properties.
- As we know, immutable properties cannot be changed once created. However, nondestructive mutation can make changes in immutable properties.
- Nondestructive mutations allow us to create a copy of that immutable property and we can add changes in that copy.
- In simple terms, nondestructive mutation enables you to change the state of an object by creating a copy with the change – rather than making changes to the original object.
- To set positional properties or properties created by using standard property syntax, the with expression can be used. In the case of non-positional properties, it must have an `init` or `set` accessor that can be changed in a with expression.

Record Types – Nondestructive Mutation 2-3

- Following code demonstrates nondestructive mutation:

Snippet

```
public class NondestructiveDemo {
 public record Player(string Name, int Avg, int age);
 public static void Main() {
 var p1 = new Player("Jim Smith", 50, 30);
 Console.WriteLine($"{nameof(p1)}: {p1}");
 var p2 = p1 with { Name = "Chris Dane", Avg = 55 };
 Console.WriteLine($"{nameof(p2)}: {p2}");
 var p3 = p1 with {Name = "Andrew Flintoff", age =38 };
 Console.WriteLine($"{nameof(p3)}: {p3}");
 Console.WriteLine($"{nameof(p1)}: {p1}");
 }
}
```

- In the code:
  - The `with` keyword is used to create a copy of the instance immutable properties.
  - This way now you can make changes in the second and third **`Player`** instances, **`p2`**, and **`p3`** respectively that you have created using `with` keyword and the original item will remain immutable.

Record Types – Nondestructive Mutation 3-3

- For reference properties, only the reference to an instance is copied and thus, the outcome of a with expression is a shallow copy.
- This means that both the original record and the copy have a reference to the same instance.
- The compiler achieves this by simulating a clone method and a copy constructor.
- The virtual clone method returns a new record initialized by the copy constructor. When a `with` expression is used, the compiler creates code that is used to call the clone method.
- After calling the clone method, it then sets the properties that are specified in the `with` expression

Record Types – Built-In Formatting for Display 1-5

- The record type has built-in compiler-generated `ToString` method which is used to display names and values of public properties.
- The `ToString` method returns a string having following syntax:

Snippet

```
<record type name> {<property name> = <value>, <property
name> = <value>,

…

}
```

- In case of reference types, type name of the object referred to by the property is printed instead of property value.

Record Types – Built-In Formatting for Display 2-5

- Following code demonstrates an example where the array is a reference type, so `System.String[]` is displayed instead of actual array element values:

Snippet

```
Avengers { FirstName = Tony, LastName = Stark, SuperHeroNames =
System.
String[] }
```

- To render the formatting, the compiler makes use of a virtual `PrintMembers` method and a `ToString` override. The `ToString` override helps to create a `StringBuilder` object with the type name which is followed by an opening bracket.

Record Types – Built-In Formatting for Display 3-5

- Following code demonstrates an example of `ToString()` and `StringBuilder`:

Snippet

```
class TestRecord2{
public record Employee(string FirstName, string LastName);
public static void Main(){
 Employee employee1 = new("Hank", "Williams");
 Console.WriteLine(employee1);
}
}
```

Output

```
Employee { FirstName = Hank, LastName = Williams }
```

Record Types – Built-In Formatting for Display 4-5

- In the code, we created an instance of an `Employee` record and provided it to the `Console.WriteLine` method, which calls its `ToString()` implementation. The output displays the name of the record type and its properties, including their values. The output format looks similar to JSON, but starts with the name of the `record` type.
- Internally, the built-in `ToString()` override looks like this:

```
public override string ToString() {
StringBuilder stringBuilder = new StringBuilder ();
stringBuilder.Append ("Employee") ; // type name
stringBuilder.Append (" { ") ;
if (PrintMembers (stringBuilder)) {
                stringBuilder.Append (" ");
 }
 stringBuilder.Append ("} ");
 return stringBuilder.ToString();
}
```

Record Types – Built-In Formatting for Display 5-5

- Records also support inheritance Inheritance rules for records are as follows:
  - A record can inherit only from another record
  - A class cannot inherit from a record
- Records can be useful to log data, copy data structures, compare different objects' properties in business applications, and so on.

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.
- 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.
- Differences between properties and methods are listed in the table.

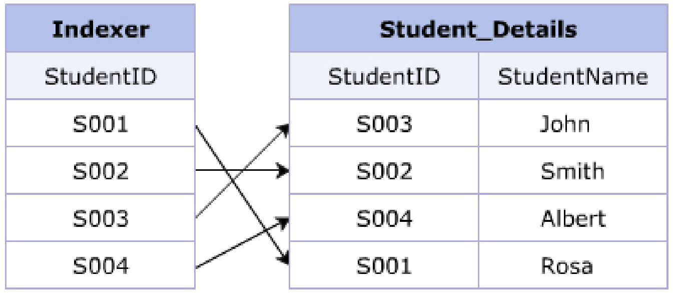
Properties	Fields
Properties represent characteristics of an object.	Methods represent the behavior of an object.
Properties contain two methods which are automatically invoked without specifying their names.	Methods are invoked by specifying method names along with the object of the class.
Properties 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.



Indexer
StudentID
S001
S002
S003
S004

Student_Details	
StudentID	StudentName
S003	John
S002	Smith
S004	Albert
S001	Rosa

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

- 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.
- Following syntax creates an indexer:

Syntax

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

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 must 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

- Indexers can be inherited like other members of the class.
- It means the base class indexers can be inherited by the derived class.
- 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]);
        }
    }
}
```

Output

```
2
4
0
```

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.

Multiple Parameters in Indexers 1-2

- 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.
- 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++)
{
for (int j = 0; j < 1; j++)
{
objAccount[i, j] = id[counter];
objAccount[i, j+1] = name[counter++];
}
}
Console.WriteLine("ID Name");
```

Multiple Parameters in Indexers 2-2

```
        Console.WriteLine();
        for (int i = 0; i< 4; i++)
        {
        for (int j = 0; j < 2; j++)
        {
        Console.Write(objAccount[i, j]+ " ");
        }
        Console.WriteLine();
        }
        }
}
```

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.

Difference between Properties and Indexers

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

Properties	Fields
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.
- To create immutable properties, there were two ways, namely, using private setters and defining setters that check current value of the property.
- A field is a data member that stores some information.
- Properties enable you to access the private fields of the class.
- Record types provide immutability in C#. Before C# 9.0, C# did not support immutability features.
- 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 ([]).