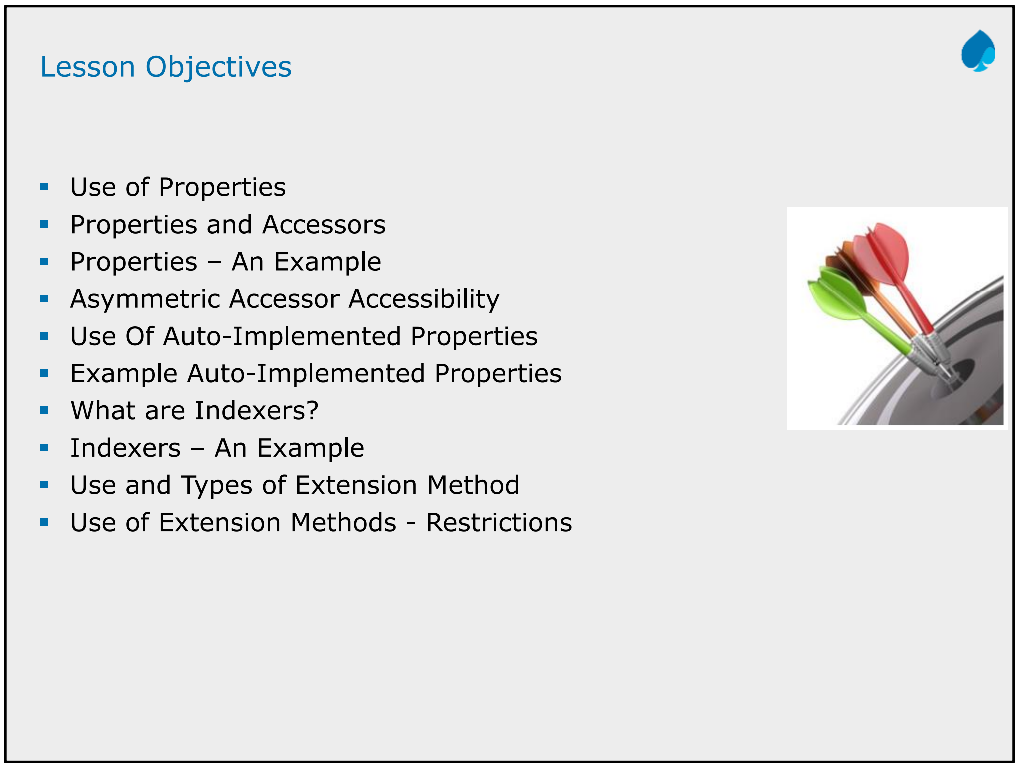
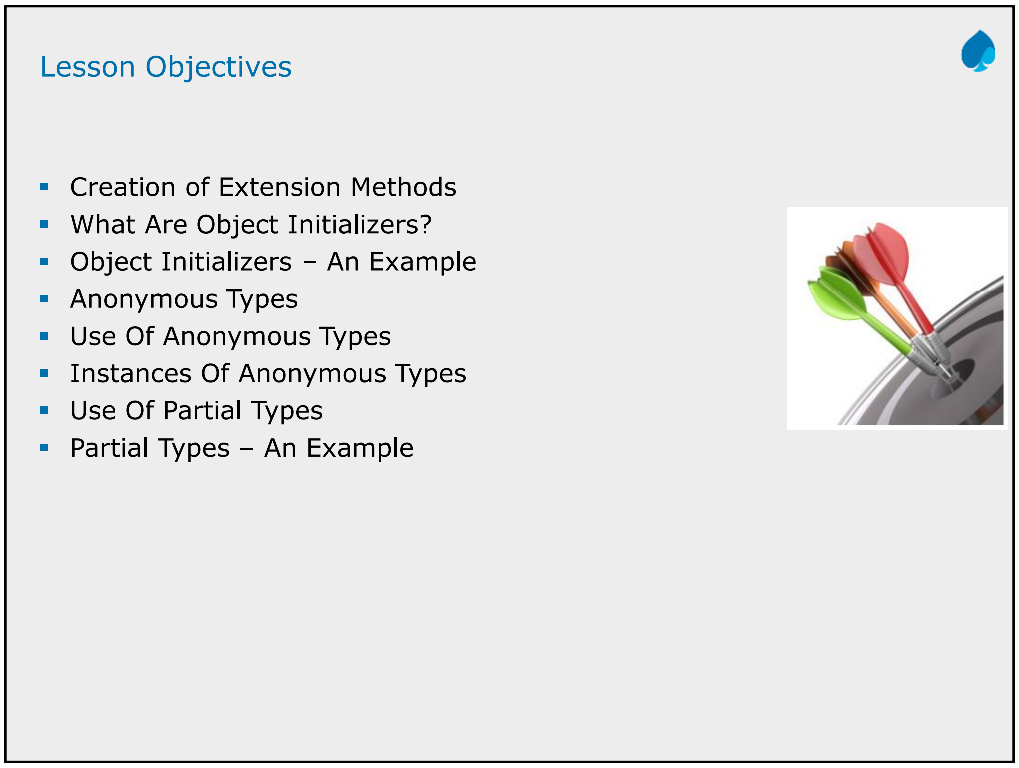
C# 7.0

Lesson 05 : Advanced Language Features



Lesson Objectives

- Use of Properties
- Properties and Accessors
- Properties – An Example
- Asymmetric Accessor Accessibility
- Use Of Auto-Implemented Properties
- Example Auto-Implemented Properties
- What are Indexers?
- Indexers – An Example
- Use and Types of Extension Method
- Use of Extension Methods - Restrictions



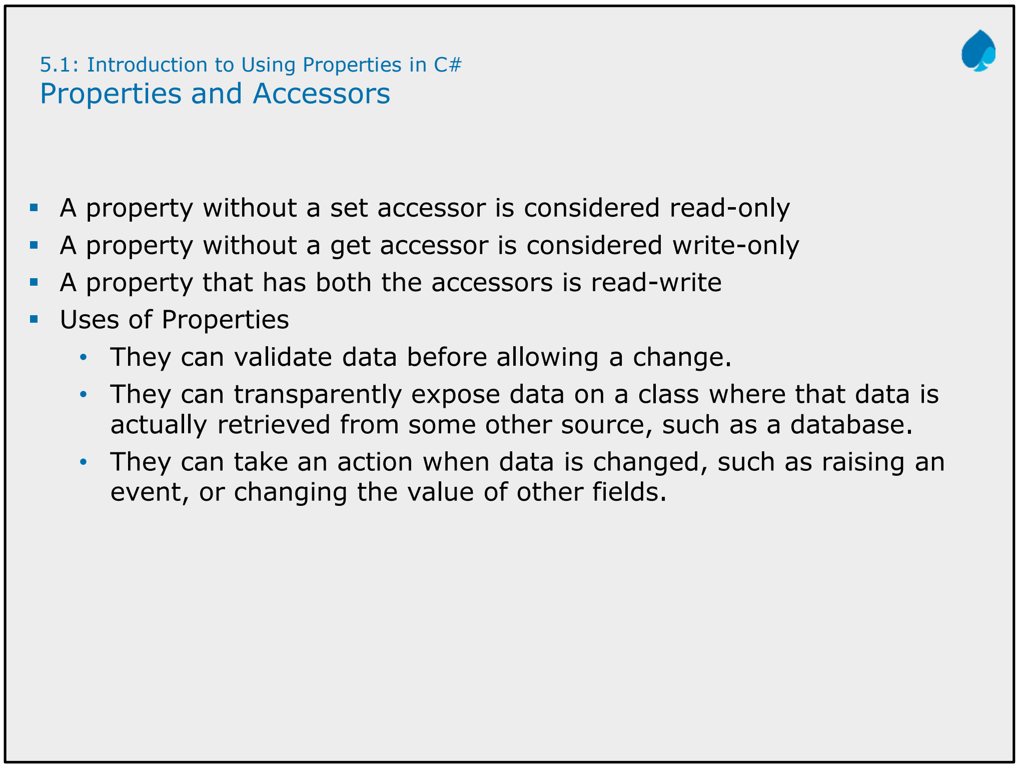
Lesson Objectives

- Creation of Extension Methods
- What Are Object Initializers?
- Object Initializers – An Example
- Anonymous Types
- Use Of Anonymous Types
- Instances Of Anonymous Types
- Use Of Partial Types
- Partial Types – An Example

5.1: Introduction to Using Properties in C#

Use of Properties

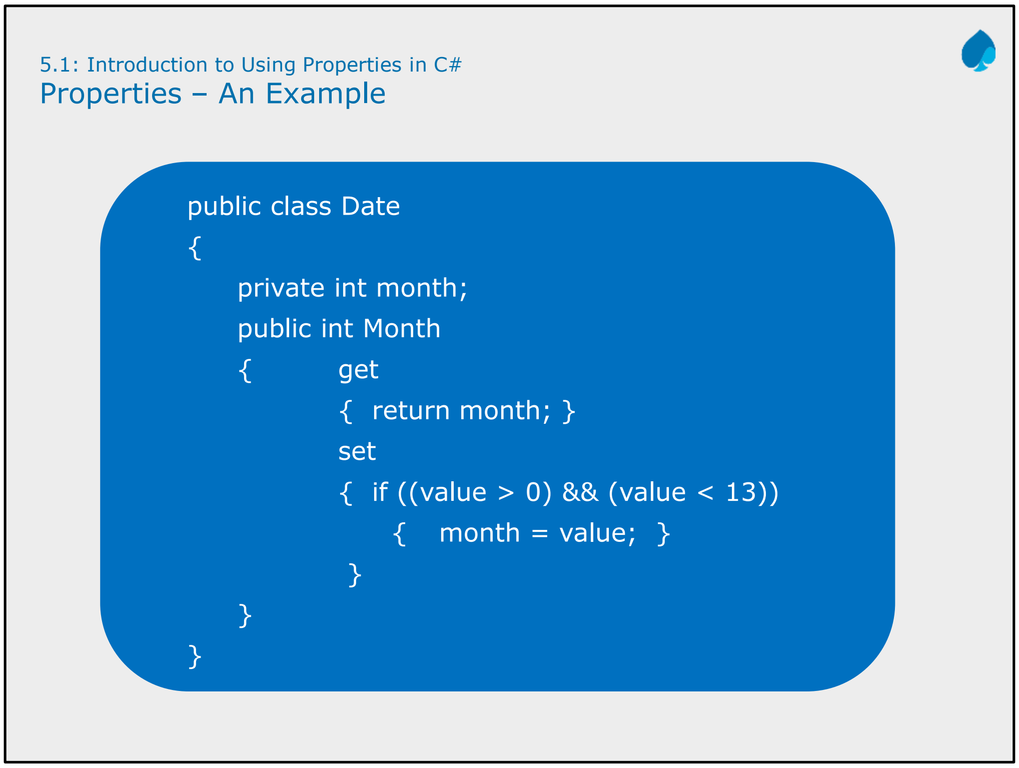
- Properties provide the chance to protect a field in a class by reading and writing to it through the property accessor
- Accomplished in programs by implementing the specialized getter and setter methods
- One or two code blocks are required: Those representing a get accessor and/or a set accessor
- The code block for the get accessor is executed when the property is read
- The code block for the set accessor is executed when the property is assigned a new value



5.1: Introduction to Using Properties in C#

Properties and Accessors

- A property without a set accessor is considered read-only
- A property without a get accessor is considered write-only
- A property that has both the accessors is read-write
- Uses of Properties
  - They can validate data before allowing a change.
  - They can transparently expose data on a class where that data is actually retrieved from some other source, such as a database.
  - They can take an action when data is changed, such as raising an event, or changing the value of other fields.



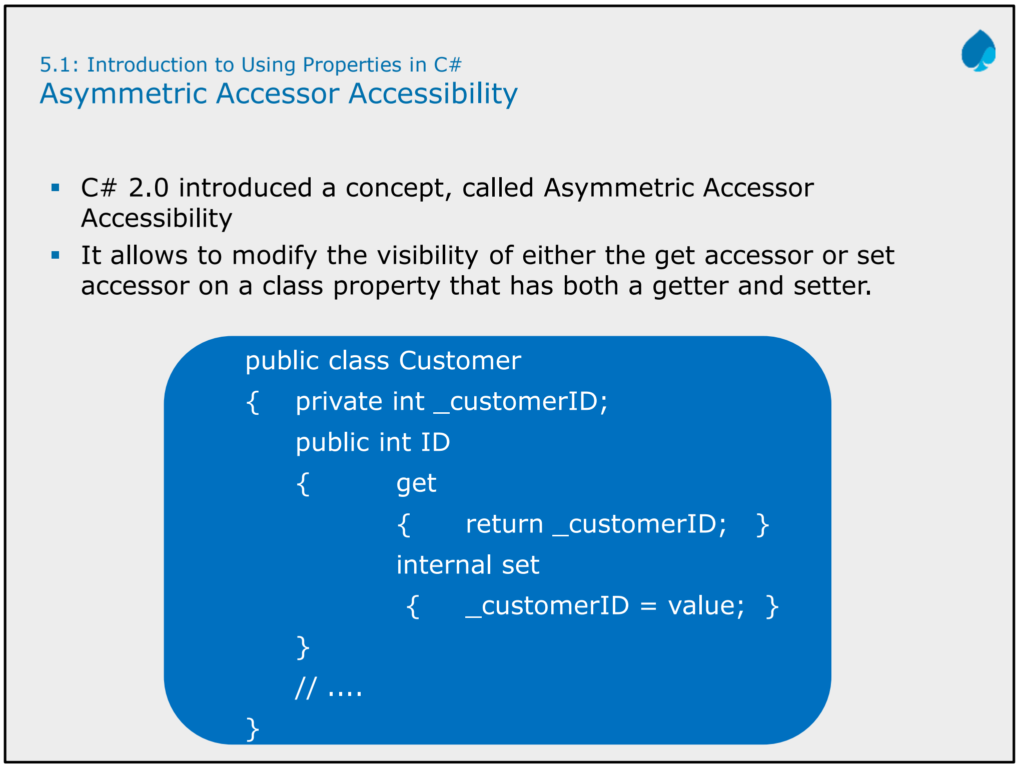
5.1: Introduction to Using Properties in C#

Properties – An Example

```
public class Date
{
    private int month;
    public int Month
    {       get
            {  return month; }
            set
            {  if ((value > 0) && (value < 13))
                 {    month = value;  }
             }
    }
}
```

Example of Properties:

In this example, **month** is declared as a property so that the **set** accessor can make sure that the **month** value is set between **1** and **12**. The **month** property uses a private field to track the actual value. The real location of a property's data is often referred to as the property's "backing store." It is common for properties to use private fields as a backing store. The field is marked private in order to make sure that it can only be changed by calling the property.



5.1: Introduction to Using Properties in C#

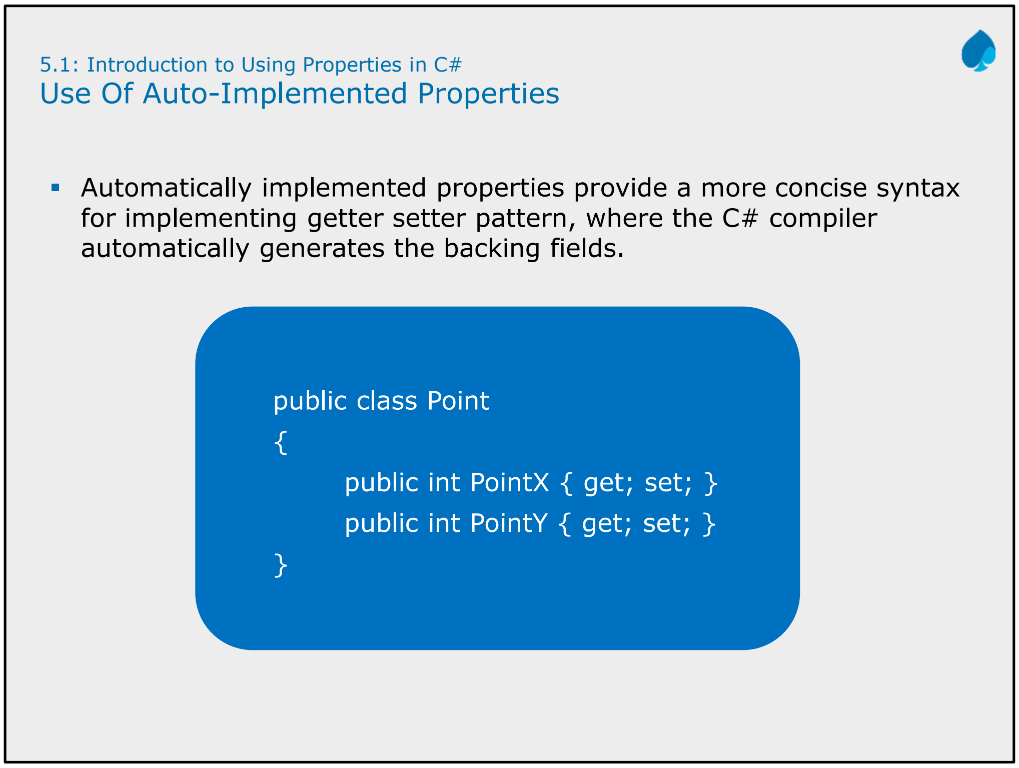
Asymmetric Accessor Accessibility

- C# 2.0 introduced a concept, called Asymmetric Accessor Accessibility
- It allows to modify the visibility of either the get accessor or set accessor on a class property that has both a getter and setter.

```
public class Customer
{    private int _customerID;
     public int ID
     {         get
               {      return _customerID;   }
               internal set
                {     _customerID = value;  }
     }
     // ....
}
```

<u>Asymmetric Accessor Accessibility:</u>
There are a few restrictions to this asymmetric accessor accessibility feature in C# 2.0:

- You can only set a different visibility on one of the two accessors, not both. Trying to restrict visibility on both **get** and **set** to **internal** gives an error:
- You cannot use the feature on properties that do not have BOTH **get** and **set** accessors. Removing the **get** accessor from the example above gives you error:
- The accessibility modifier used on a **get** or **set** accessor can only restrict visibility not increase it.



5.1: Introduction to Using Properties in C#

Use Of Auto-Implemented Properties

- Automatically implemented properties provide a more concise syntax for implementing getter setter pattern, where the C# compiler automatically generates the backing fields.

```
public class Point
{
        public int PointX { get; set; }
        public int PointY { get; set; }
}
```

<u>Use of Auto-Implemented Properties:</u>

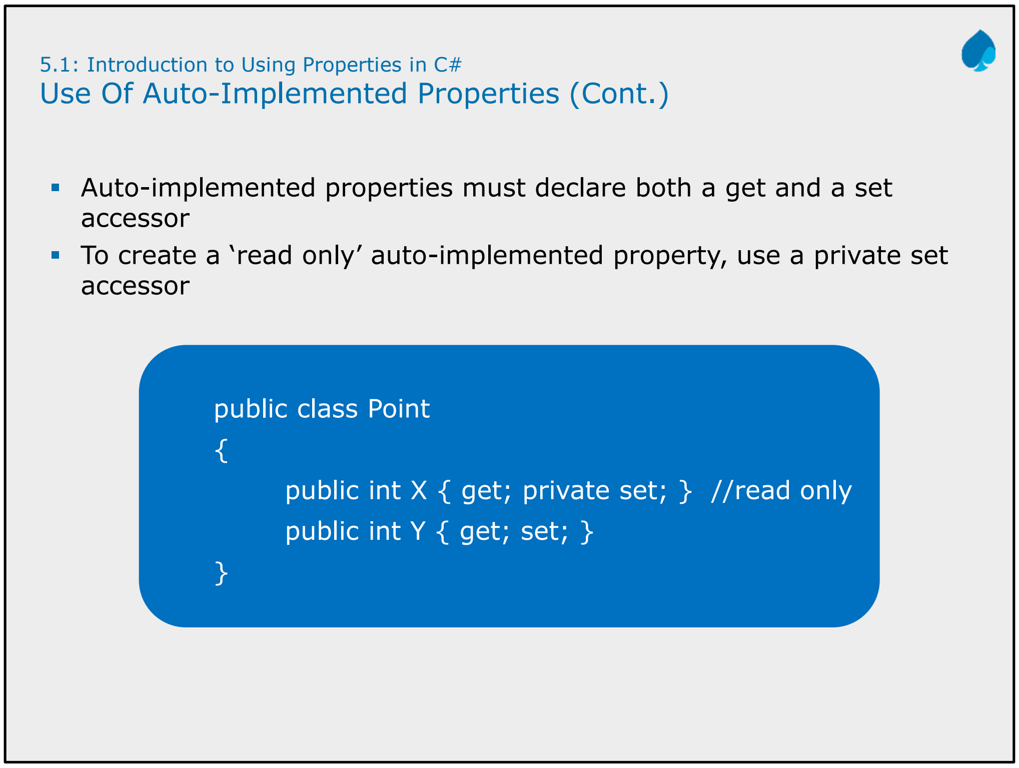
Often, property accessors (**get** and **set**) have trivial implementations and follow the pattern that simply get (return) a private field and set the private field to the value passed.

In the following example, the **Point** class contains two properties:

```
public class Point
{
    private int x;
    private int y;
    public int X { get { return x; } set { x = value; } }
    public int Y { get { return y; } set { y = value; } }
}
```

To simplify this, let the compiler generate the field and you can simply specify the property name as shown below:

```
public class Point
{
    public int PointX { get; set; }
    public int PointY { get; set; }
}
```



5.1: Introduction to Using Properties in C#

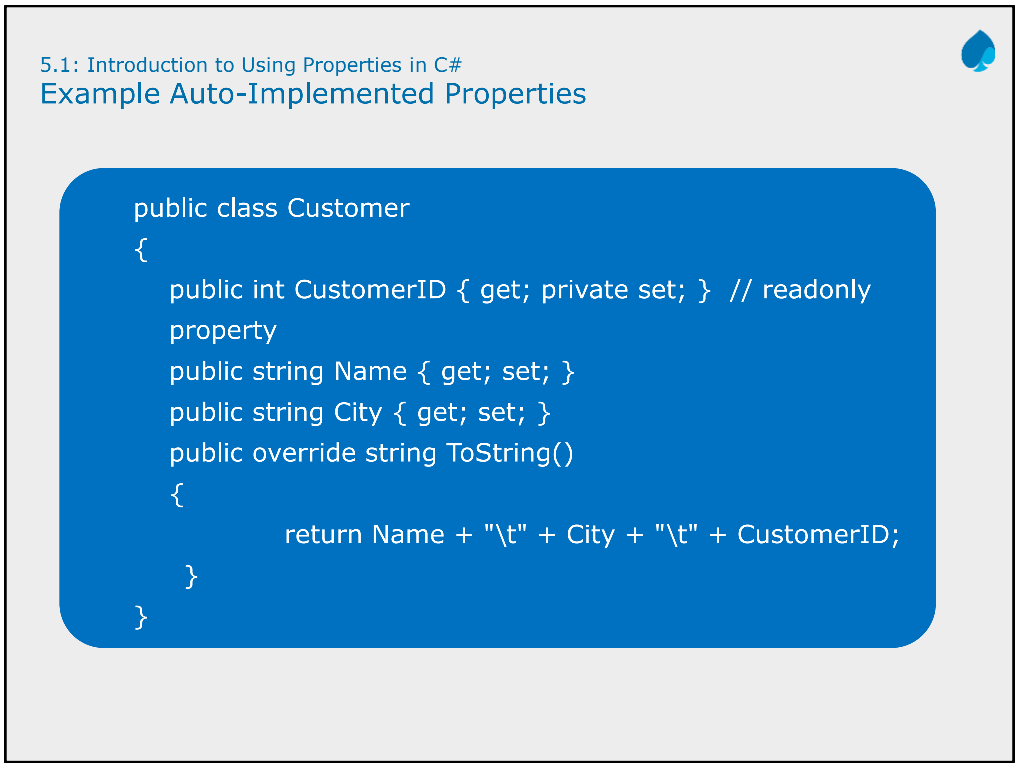
Use Of Auto-Implemented Properties (Cont.)

- Auto-implemented properties must declare both a get and a set accessor
- To create a ‘read only’ auto-implemented property, use a private set accessor

```
public class Point
{
        public int X { get; private set; }  //read only
        public int Y { get; set; }
}
```

<u>Use of Auto-Implemented Properties (Contd.):</u>
Another example:

```
class LightweightCustomer
{
        public double TotalPurchases { get; set; }
        public string Name { get; private set; } // read-only
        public int CustomerID { get; private set; } // read-only
}
```

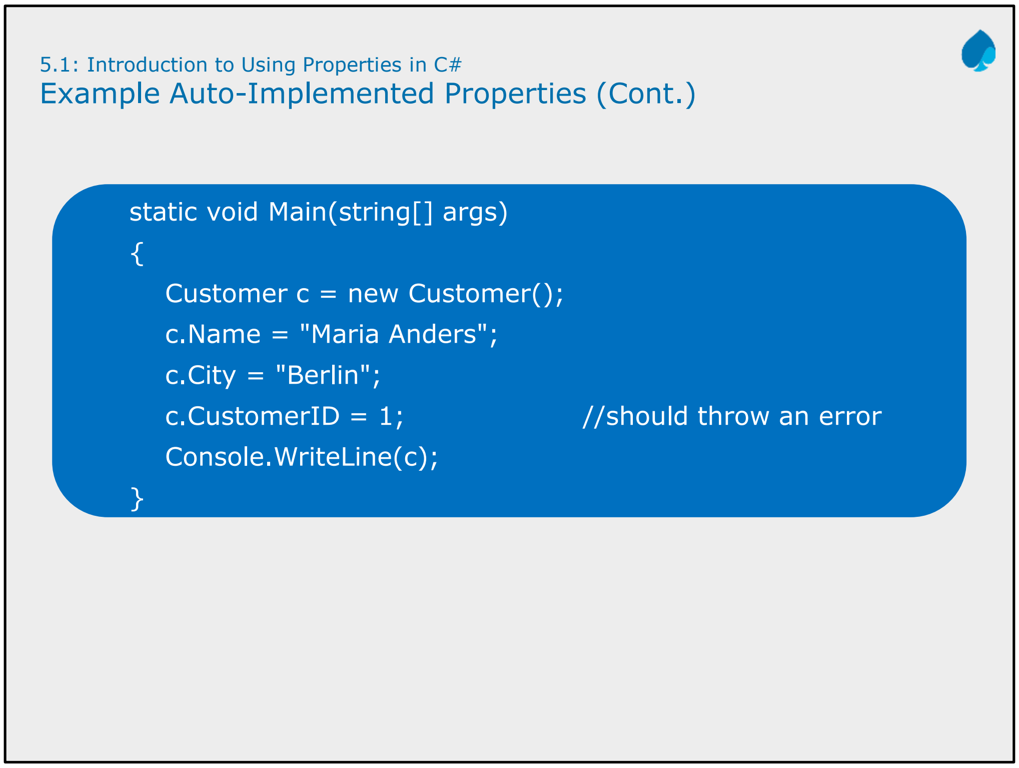


5.1: Introduction to Using Properties in C#

Example Auto-Implemented Properties

```
public class Customer
{
    public int CustomerID { get; private set; }  // readonly
    property
    public string Name { get; set; }
    public string City { get; set; }
    public override string ToString()
    {
                return Name + "\t" + City + "\t" + CustomerID;
     }
}
```

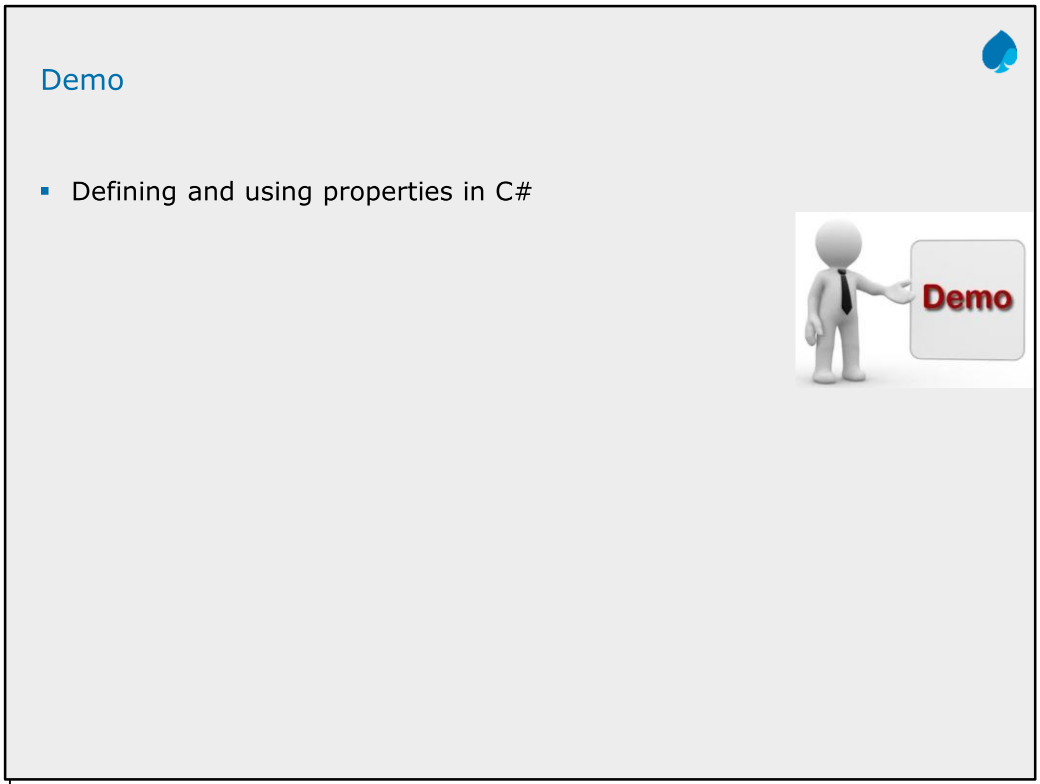
In the above example, an error occurs when you attempt to set the **CustomerID** property directly. This happens due to the private modifier on set. The **CustomerID** property now behaves as if it were read-only.



5.1: Introduction to Using Properties in C#

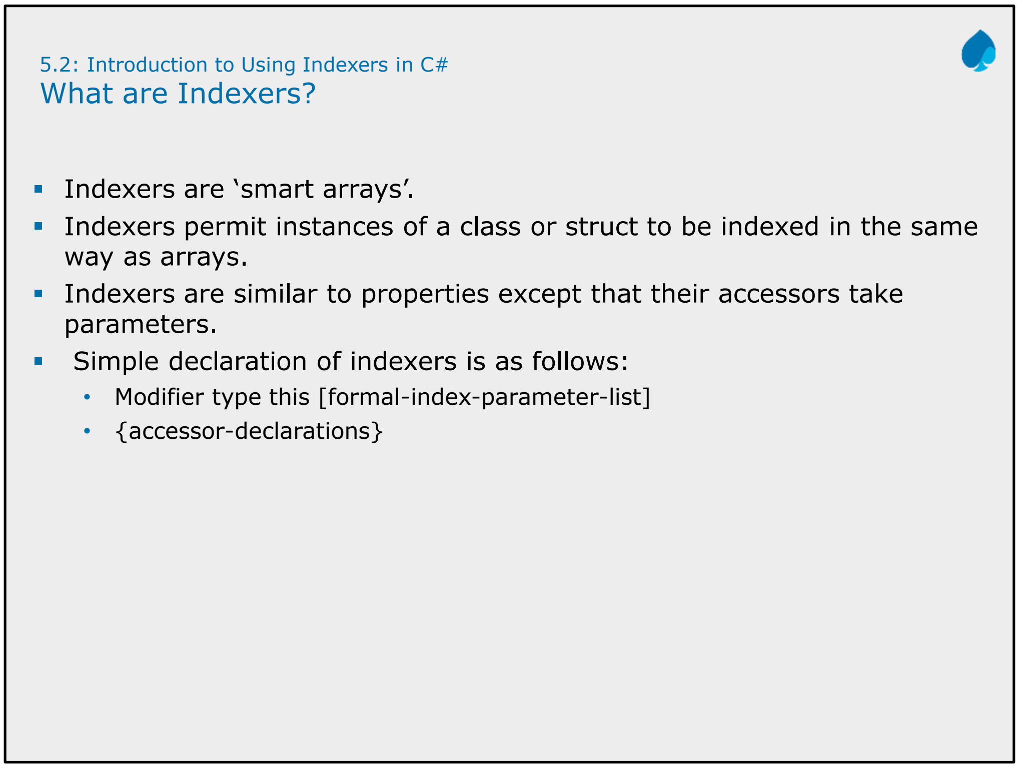
Example Auto-Implemented Properties (Cont.)

```
static void Main(string[] args)
{
    Customer c = new Customer();
    c.Name = "Maria Anders";
    c.City = "Berlin";
    c.CustomerID = 1;                    //should throw an error
    Console.WriteLine(c);
}
```



Demo

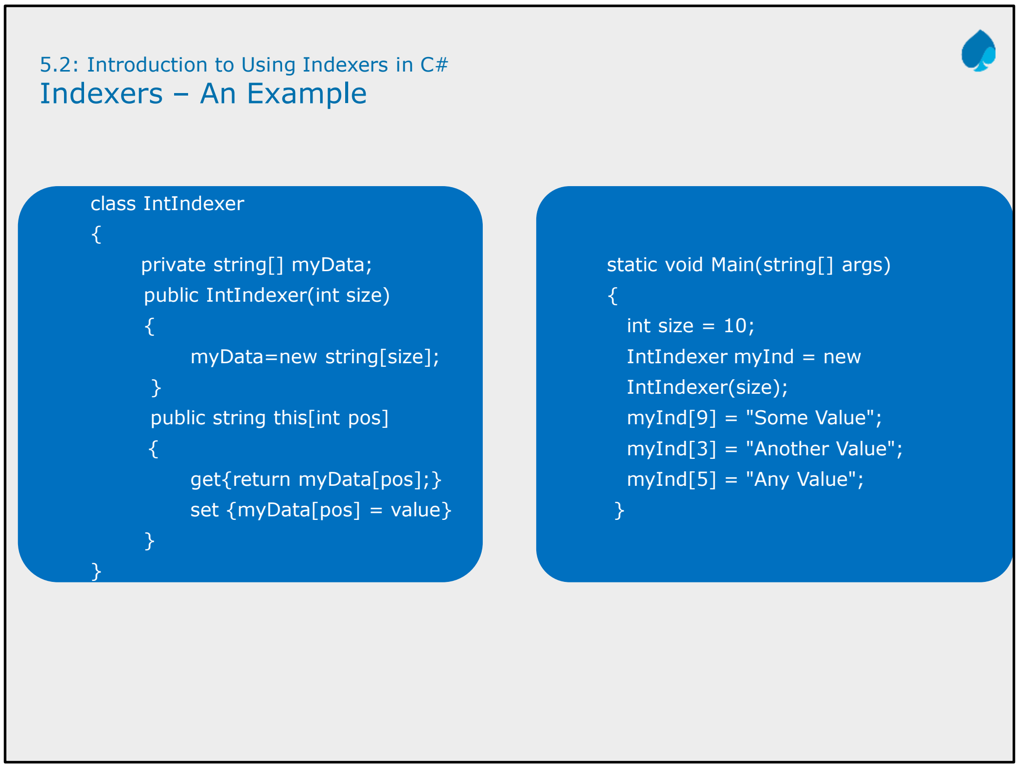
- Defining and using properties in C#



5.2: Introduction to Using Indexers in C#

What are Indexers?

- Indexers are ‘smart arrays’.
- Indexers permit instances of a class or struct to be indexed in the same way as arrays.
- Indexers are similar to properties except that their accessors take parameters.
- Simple declaration of indexers is as follows:
  - Modifier type this [formal-index-parameter-list]
  - {accessor-declarations}



5.2: Introduction to Using Indexers in C#

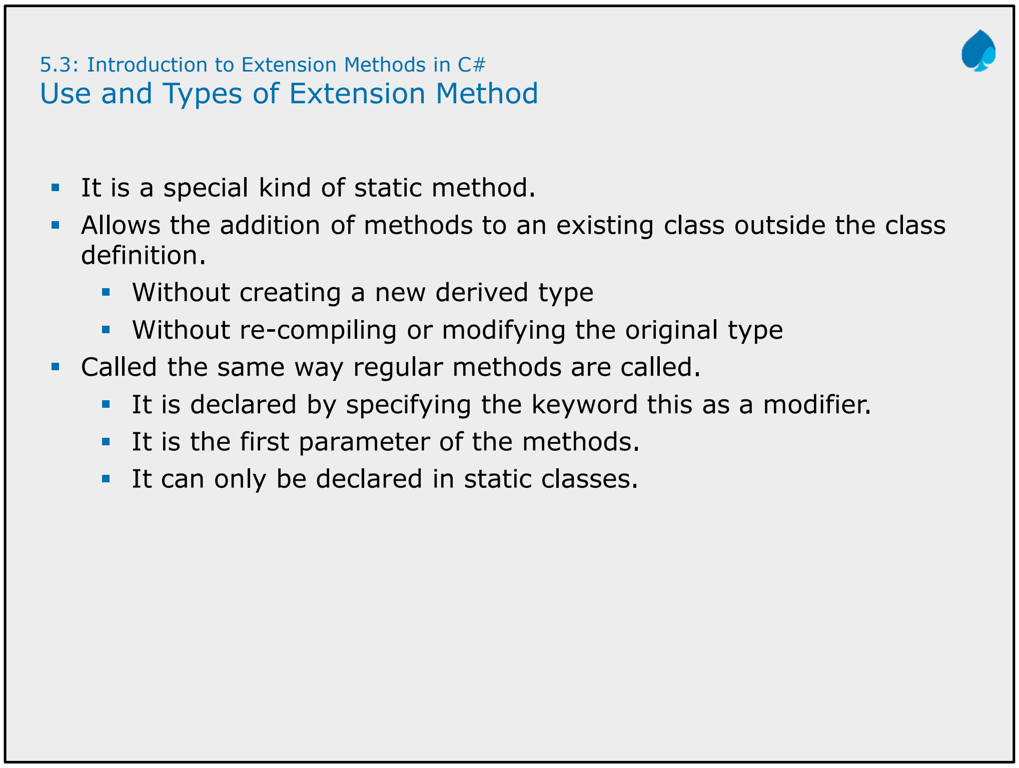
Indexers – An Example

```
class IntIndexer
{
        private string[] myData;
        public IntIndexer(int size)
        {
                myData=new string[size];
         }
         public string this[int pos]
         {
                get{return myData[pos];}
                set {myData[pos] = value}
        }
}
```

```
static void Main(string[] args)
{
   int size = 10;
   IntIndexer myInd = new
   IntIndexer(size);
   myInd[9] = "Some Value";
   myInd[3] = "Another Value";
   myInd[5] = "Any Value";
 }
```

Demo

- Indexers in C#



5.3: Introduction to Extension Methods in C#

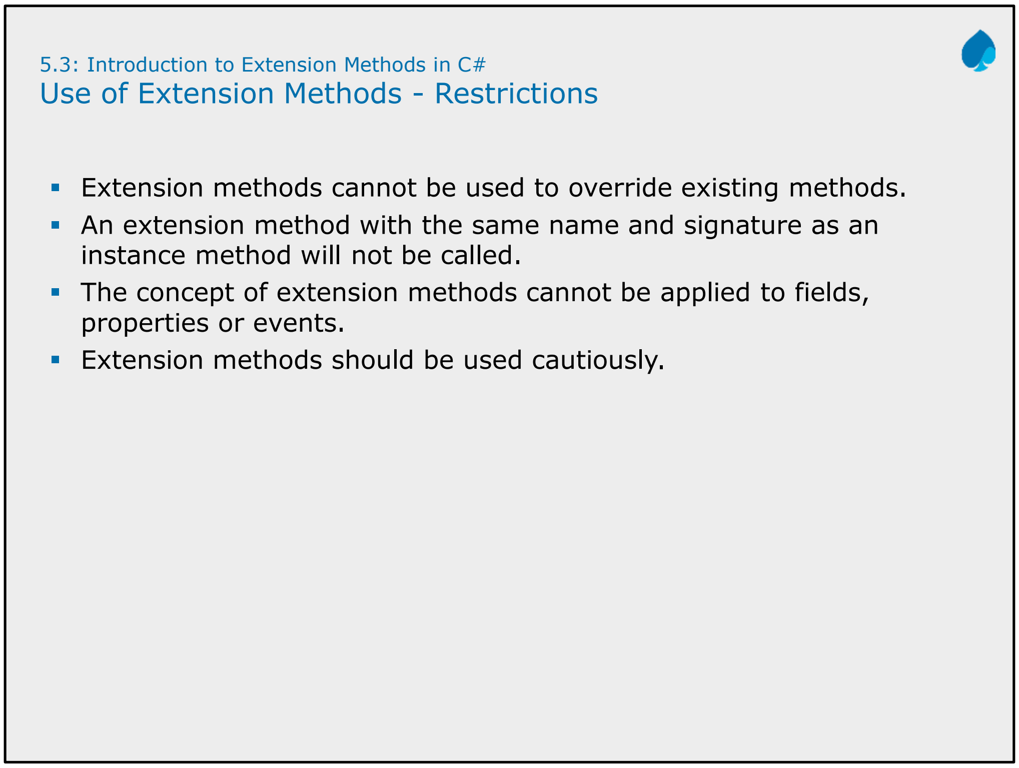
Use and Types of Extension Method

- It is a special kind of static method.
- Allows the addition of methods to an existing class outside the class definition.
  - Without creating a new derived type
  - Without re-compiling or modifying the original type
- Called the same way regular methods are called.
  - It is declared by specifying the keyword this as a modifier.
  - It is the first parameter of the methods.
  - It can only be declared in static classes.

The Extension Methods:
Extension methods are static methods that can be invoked using instance method syntax. In effect, extension methods make it possible to extend existing types and constructed types with additional methods.

Extension methods allow existing classes to be extended without relying on inheritance or having to change the class's source code. This means that if you want to add some methods into the existing String class you can do it quite easily.

Declaring Extension Methods:
Extension methods are declared by specifying the keyword this as a modifier on the first parameter of the methods. Extension methods can only be declared in static classes.



5.3: Introduction to Extension Methods in C#

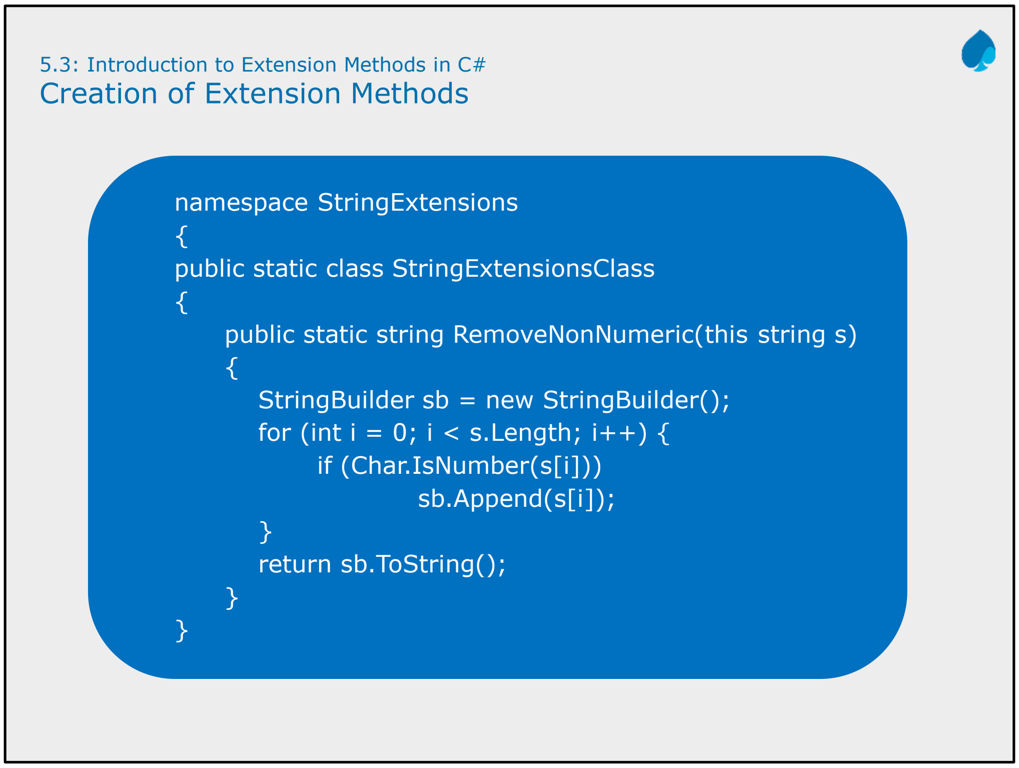
Use of Extension Methods - Restrictions

- Extension methods cannot be used to override existing methods.
- An extension method with the same name and signature as an instance method will not be called.
- The concept of extension methods cannot be applied to fields, properties or events.
- Extension methods should be used cautiously.

```
public static class Utility
   {

      public static int WordCount(this string sentence)
     {

         return sentence.Split(new char[] { ' ' }).Length;

     }
   }

   class UtilityTest
   {
      static void Main(string[] args)
      {
         string testString = “CAPGEMINI LnD";
         Console.WriteLine(testString.WordCount());
      }
   }
```

<u>Use of Extension Methods – Restrictions</u>
The following are a couple of rules to consider when deciding on whether or not to use
extension methods:

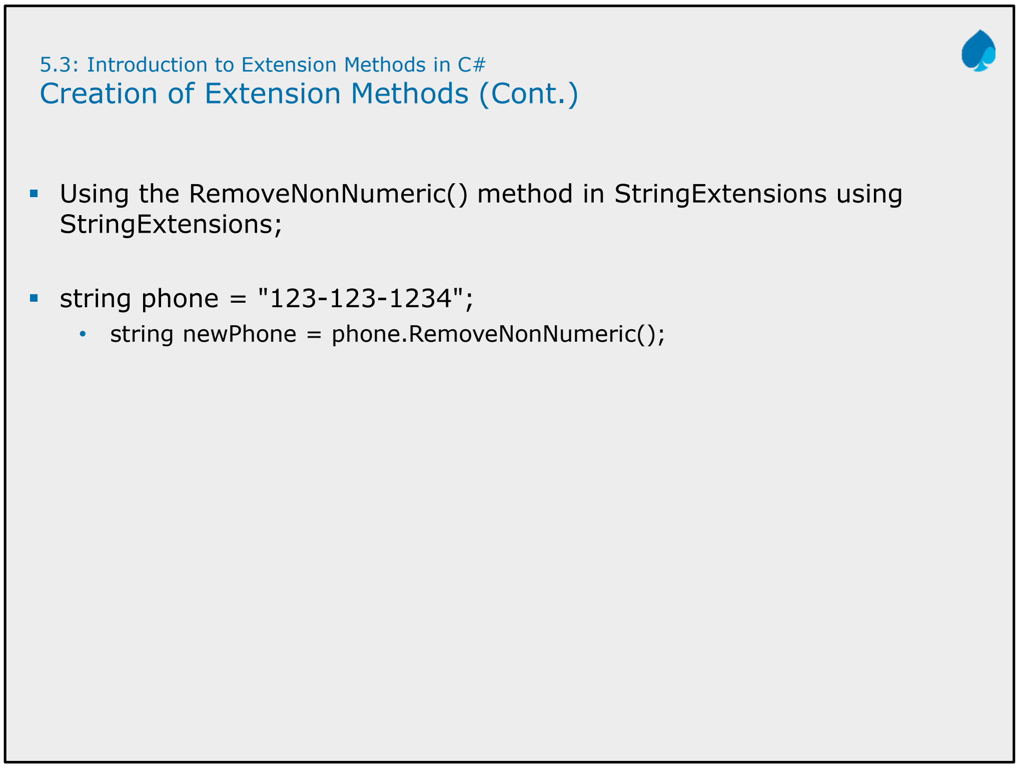
- Extension methods cannot be used to override existing methods.
- An extension method with the same name and signature as an instance method will not be called.
- The concept of extension methods cannot be applied to fields, properties or events.
- Use extension methods sparingly.



5.3: Introduction to Extension Methods in C#

Creation of Extension Methods

```
namespace StringExtensions
{
public static class StringExtensionsClass
{
     public static string RemoveNonNumeric(this string s)
     {
         StringBuilder sb = new StringBuilder();
         for (int i = 0; i < s.Length; i++) {
                if (Char.IsNumber(s[i]))
                          sb.Append(s[i]);
         }
         return sb.ToString();
     }
}
```



5.3: Introduction to Extension Methods in C#

Creation of Extension Methods (Cont.)

- Using the RemoveNonNumeric() method in StringExtensions using StringExtensions;
- string phone = "123-123-1234";
  - string newPhone = phone.RemoveNonNumeric();

<u>Use of Extension Methods - Example:</u>
This is an example of how an extension method can be used. You'll see that the namespace for the extension method class is imported. From there, the compiler treats the **RemoveNonNumeric()** method as if it was originally a part of the standard **System.String** class.

```
using StringExtensions;
....
string phone = "123-123-1234";
string newPhone = phone.RemoveNonNumeric();
```

Demo

- Using Extension Methods in C#

5.4: Introduction to Object Initializers in C#

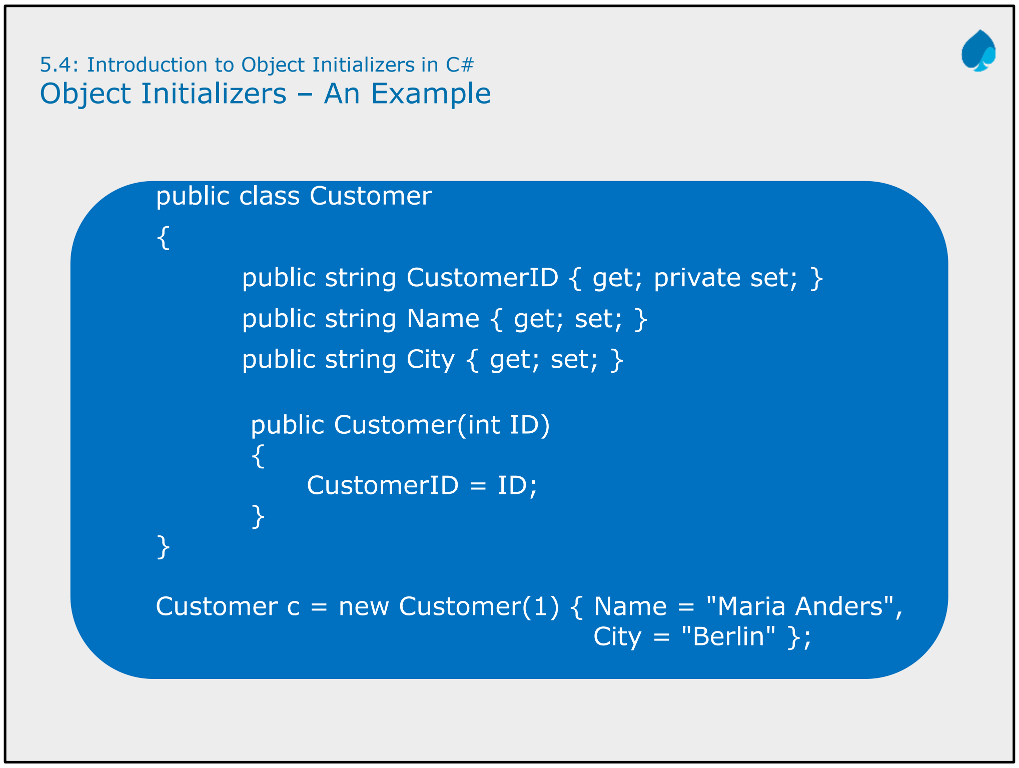
What Are Object Initializers?

- An object initializer is used to assign values to an object fields or properties when the object is created.
- There is no need to explicitly invoke a constructor
- It combines object creation and initialization in a single step.

<u>What are object initializers?</u>
From C# 3.0 onwards, when declaring an object or collection, you may include an initializer that specifies the initial values of the members of the newly created object or collection. This new syntax combines object creation and initialization in a single step.

Using Object Initializers
An object initializer consists of a sequence of member initializers, enclosed by **{and}** tokens and separated by commas. Each member initializer must name an accessible field or property of the object being initialized, followed by an 'equal to' (**=**) sign and an expression or an object or collection initializer. It is an error for an object initializer to include more than one member initializer for the same field or property. For example

```
public class Point
{
   public int X { get; set; }
   public int Y { get; set; }
}
```



5.4: Introduction to Object Initializers in C#

Object Initializers – An Example

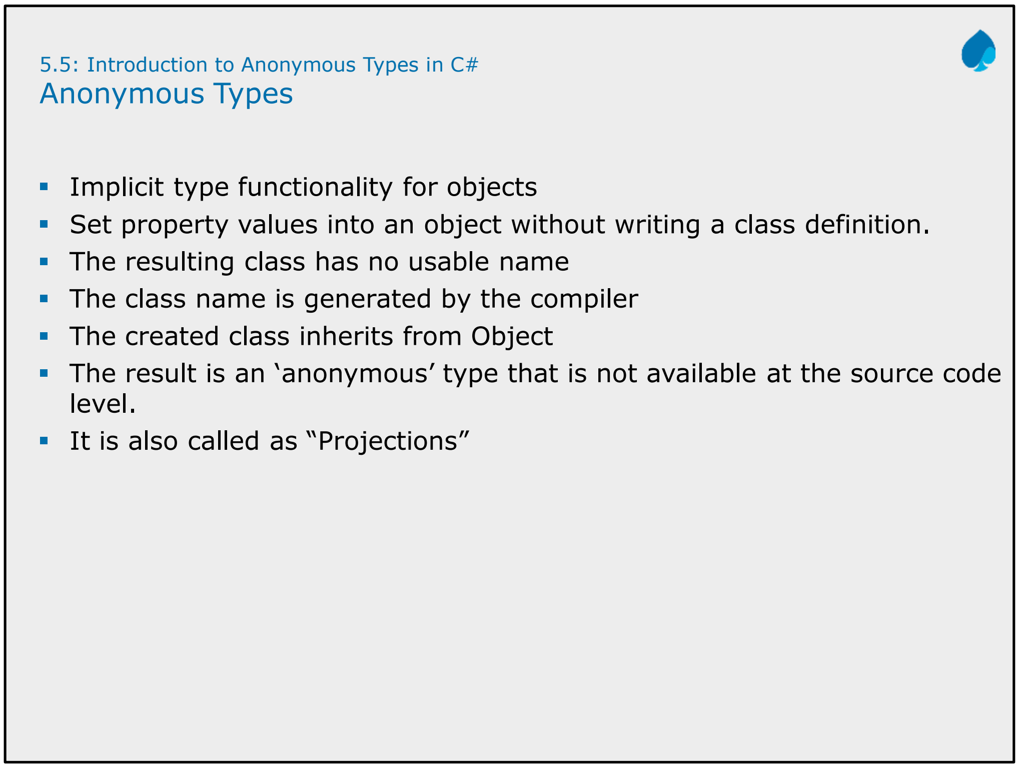
```
public class Customer
{
        public string CustomerID { get; private set; }
        public string Name { get; set; }
        public string City { get; set; }

         public Customer(int ID)
         {
              CustomerID = ID;
         }
}

Customer c = new Customer(1) { Name = "Maria Anders",
                                        City = "Berlin" };
```

<u>Example - Object Initializers :</u>

```
        public class Customer
        {
                public string CustomerID { get; private set; }
                public string Name { get; set; }
                public string City { get; set; }
                public Customer(int ID)
                {
                     CustomerID = ID;
                }
                public override string ToString()
                {
                     return Name + "\t" + City + "\t" + CustomerID;
                }
        }
```

Demo

- Using Object Initializers in C#



5.5: Introduction to Anonymous Types in C#

Anonymous Types

- Implicit type functionality for objects
- Set property values into an object without writing a class definition.
- The resulting class has no usable name
- The class name is generated by the compiler
- The created class inherits from Object
- The result is an 'anonymous' type that is not available at the source code level.
- It is also called as "Projections"

Anonymous Types:

From C# 3.0 onwards the new operator can be used with an anonymous object initializer to create an object of an anonymous type.

Specifically, an anonymous object initializer of the following form declares an anonymous type of the form.

new { p1 = e1 , p2 = e2 , … pn = en }

To facilitate the creation of classes from data values, C# 3.0 and higher versions provide the ability to easily declare an anonymous type and return an instance of that type. To create an anonymous type, the new operator is used with an anonymous object initializer. For example, when presented with the following declaration, the C# compiler automatically creates a new type that has two properties: one, called **Name** of the type string, and another called **Age** having the **int** type:

var person = new { Name = “John Doe”, Age = 33 };

Each member of the anonymous type is a property inferred from the object initializer. The name of the anonymous type is automatically generated by the compiler and cannot be referenced from the user code.

5.5: Introduction to Anonymous Types in C#

Use Of Anonymous Types

- Anonymous types enables developers to concisely define inline CLR types within code, without having to explicitly define a formal class declaration of the type.
- To create an anonymous type, the new operator is used with an anonymous object initializer.
- Example:
  - var person = new { Name = “John Doe”, Age = 33 };
- C# compiler automatically creates a new type that has two properties: one called Name of type string, and another called Age having type int.

When to use Anonymous Types
•Need a temporary object to hold related data
•Don’t need methods
•When there is a need for different set of properties for each declaration
•When there is a need to change the order of the properties for each declaration.

When Not to use Anonymous Types
•There is a need to define a method
•There is a need to define another variable.
•There is a need to share data across methods

5.5: Introduction to Anonymous Types in C#

Instances Of Anonymous Types

- Two anonymous object initializers that specify a sequence of properties of the same names and types in the same order produce instances of the same anonymous type.

```
var p1 = new { Name = "Lawnmower", Price = 495.00 };
var p2 = new { Name = "Shovel", Price = 26.95 };
p1 = p2;
```

Instances of Anonymous Types:
Within the same program, two anonymous object initializers that specify a sequence of properties of the same names and types in the same order produce instances of the same anonymous type.

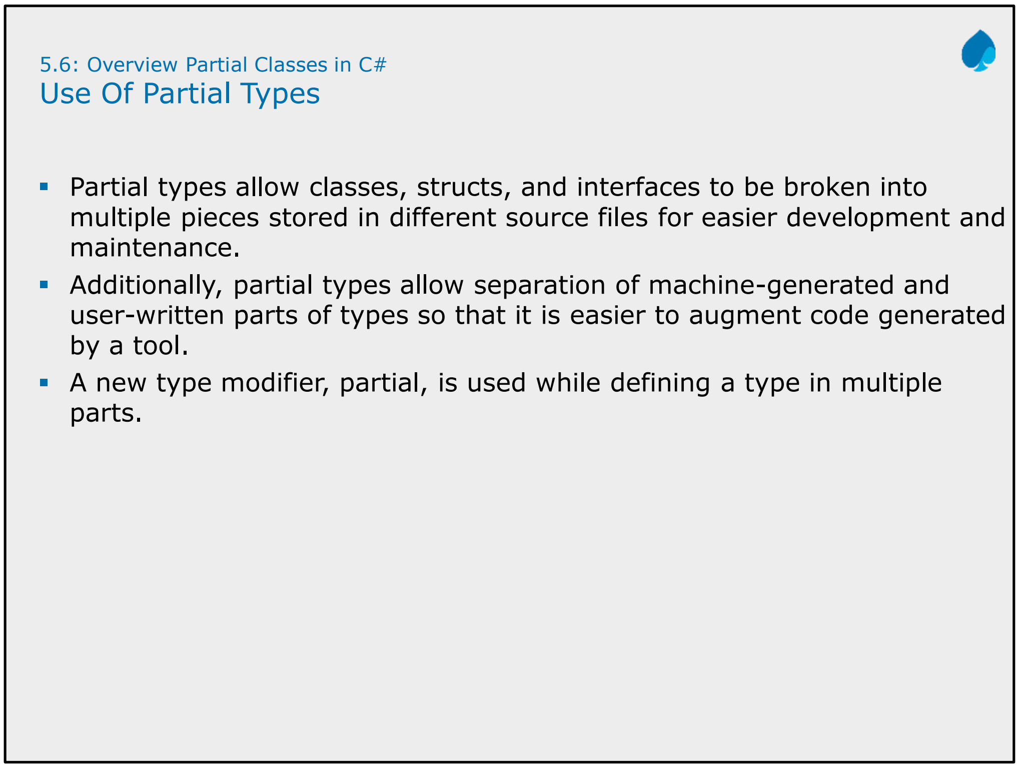
In the example,

```
var p1 = new { Name = "Lawnmower", Price = 495.00 };
var p2 = new { Name = "Shovel", Price = 26.95 };
p1 = p2;
```

The assignment on the last line is permitted because p1 and p2 are of the same anonymous type.

Demo

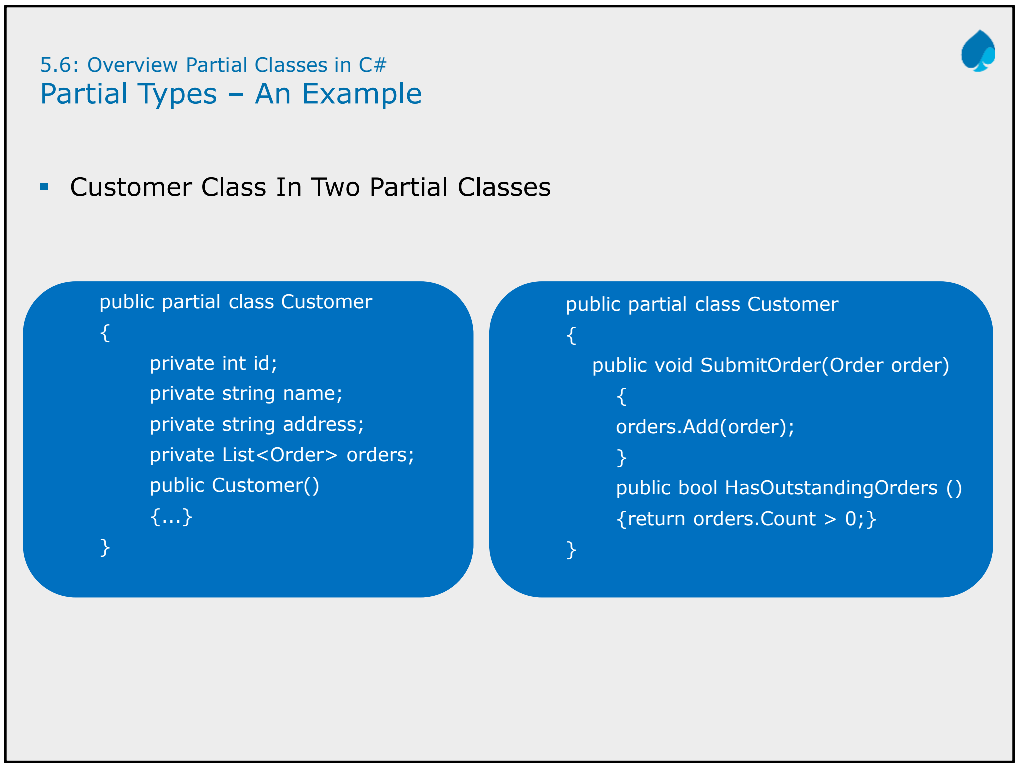
- Implementing Anonymous Types in C#



5.6: Overview Partial Classes in C#

Use Of Partial Types

- Partial types allow classes, structs, and interfaces to be broken into multiple pieces stored in different source files for easier development and maintenance.
- Additionally, partial types allow separation of machine-generated and user-written parts of types so that it is easier to augment code generated by a tool.
- A new type modifier, partial, is used while defining a type in multiple parts.



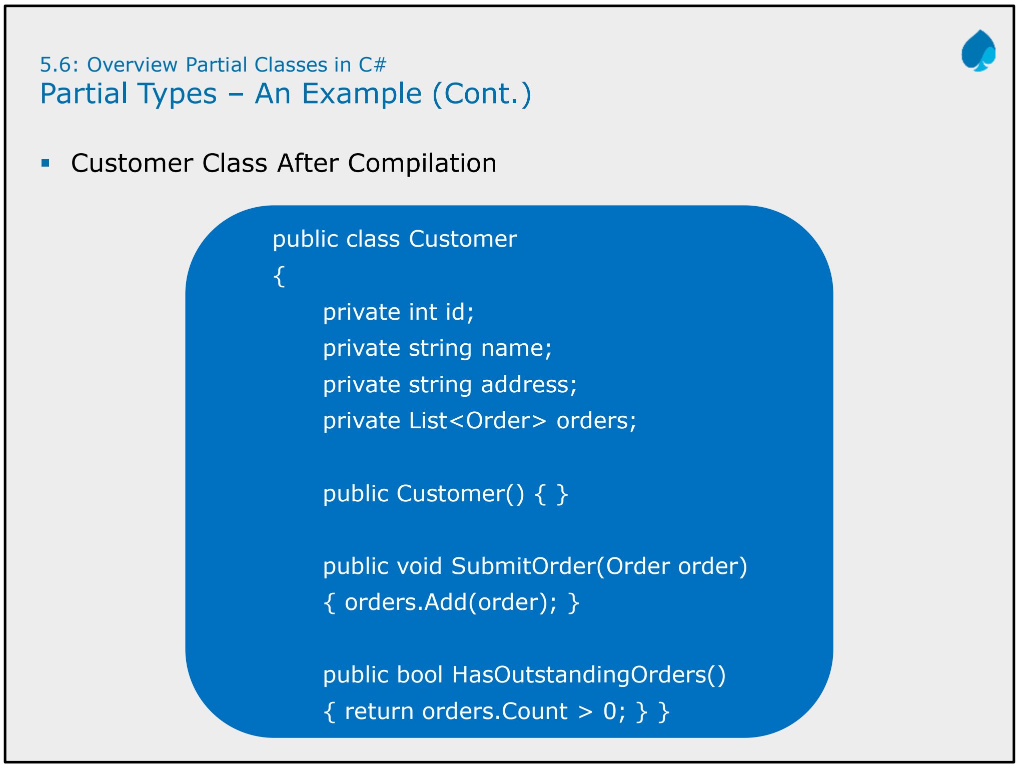
5.6: Overview Partial Classes in C#

Partial Types – An Example

- Customer Class In Two Partial Classes

```
public partial class Customer
{
        private int id;
        private string name;
        private string address;
        private List<Order> orders;
        public Customer()
        {…}
}
```

```
public partial class Customer
{
    public void SubmitOrder(Order order)
        {
        orders.Add(order);
        }
        public bool HasOutstandingOrders ()
        {return orders.Count > 0;}
}
```



5.6: Overview Partial Classes in C#

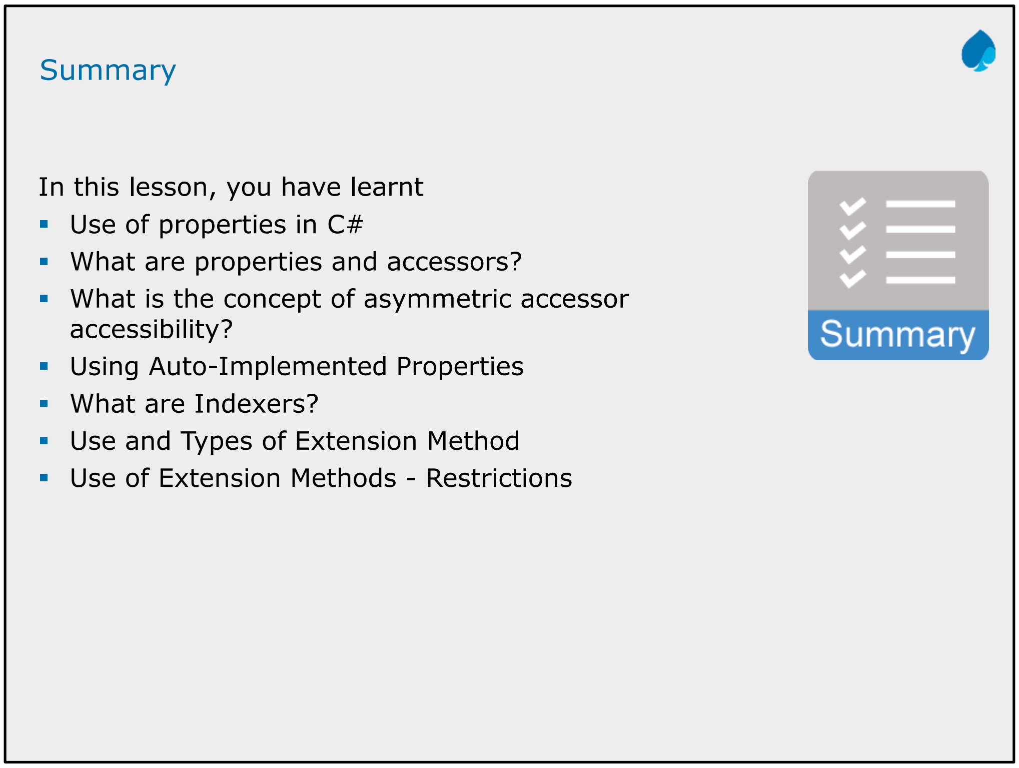
Partial Types – An Example (Cont.)

- Customer Class After Compilation

```
public class Customer
{
    private int id;
    private string name;
    private string address;
    private List<Order> orders;

    public Customer() { }

    public void SubmitOrder(Order order)
    { orders.Add(order); }

    public bool HasOutstandingOrders()
    { return orders.Count > 0; } }
```

Demo

- Implementing Partial Classes in C#



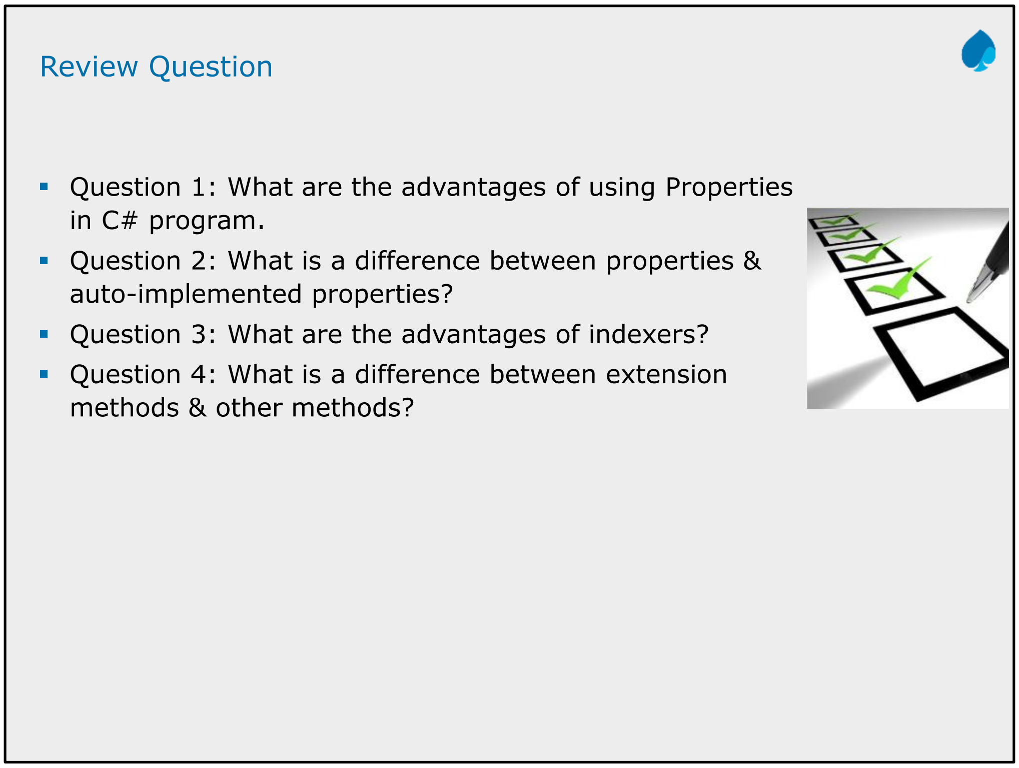
Summary

In this lesson, you have learnt

- Use of properties in C#
- What are properties and accessors?
- What is the concept of asymmetric accessor accessibility?
- Using Auto-Implemented Properties
- What are Indexers?
- Use and Types of Extension Method
- Use of Extension Methods - Restrictions

Summary

Add the notes here.



Review Question

- Question 1: What are the advantages of using Properties in C# program.
- Question 2: What is a difference between properties & auto-implemented properties?
- Question 3: What are the advantages of indexers?
- Question 4: What is a difference between extension methods & other methods?