**Introducing structs**  
  
In Visual C#, a struct is a programming construct that you can use to define custom types. Structs are essentially lightweight data structures that represent related pieces of information as a single item. For example:

* A struct named Point might consist of fields to represent an x-coordinate and a y-coordinate.
* A struct named Circle might consist of fields to represent an x-coordinate, a y-coordinate, and a radius.
* A struct named Color might consist of fields to represent a red component, a green component, and a blue component.

Most of the built-in types in Visual C#, such as int, bool, and char, are defined by structs. You can use structs to create your own types that behave like built-in types.

You use the struct keyword to declare a struct, as shown by the following example:

//Declaring a Struct   
public struct Coffee   
{   
    public int Strength;   
    public string Bean;   
    public string CountryOfOrigin;   
    // Other methods, fields, properties, and events.   
}

The struct keyword is preceded by an access modifier— public in the above example—that specifies where you can use the type. You can use the following access modifiers in your struct declarations:

**Access Modifier Details**

* public - The type is available to code running in any assembly.
* internal - The type is available to any code within the same assembly, but not available to code in another assembly. This is the default value if you do not specify an access modifier.
* private - The type is only available to code within the struct that contains it. You can only use the private access modifier with nested structs.

Structs can contain a variety of members including constructors, fields, constants, properties, indexers, methods, operators, events, and even nested types. Keep in mind that structs are intended to be lightweight therefore if you find yourself adding multiple methods, constructors, and events, you should consider using a class instead.

**Using a Struct**  
  
To create an instance of a struct, you use the new keyword, as shown by the following example:  
  
**Instantiating a Struct**  
  
Coffee coffee1 = new Coffee();  
coffee1.Strength = 3;  
coffee1.Bean = "Arabica";  
coffee1.CountryOfOrigin = "Kenya";  
  
**Initializing Structs**  
  
You might have noticed that the syntax for instantiating a struct, for example, new Coffee(), is similar to the syntax for calling a method. This is because when you instantiate a struct, you are actually calling a special type of method called a constructor. A constructor is a method in the struct that has the same name as the struct.   
  
When you instantiate a struct with no arguments, such as new Coffee(), you are calling the default constructor which is created by the Visual C# compiler. If you want to be able to specify default field values when you instantiate a struct, you can add constructors that accept parameters to your struct.  
  
The following example shows how to create a constructor in a struct:  
  
**Adding a Constructor**  
  
public struct Coffee  
{  
   // This is the custom constructor.  
   public Coffee(int strength, string bean, string countryOfOrigin)  
   {  
      this.Strength = strength;  
      this.Bean = bean;  
      this.CountryOfOrigin = countryOfOrigin;  
   }  
  // These statements declare the struct fields and set the default values.  
   public int Strength;  
   public string Bean;  
   public string CountryOfOrigin;   
   // Other methods, fields, properties, and events.  
}  
  
The following example shows how to use this constructor to instantiate a Coffee item:  
  
**Calling a Constructor**  
  
// Call the custom constructor by providing arguments for the three required parameters.  
Coffee coffee1 = new Coffee(4, "Arabica", "Columbia");  
  
You can add multiple constructors to your struct, with each constructor accepting a different combination of parameters. However, you cannot add a default constructor to a struct because it is created by the compiler.

**Extending structs**

In order to go beyond a simple struct, you can extend it by adding properties and indexers.  This section discusses using properties and indexers in your struct.  Again, if you find yourself going to this extent, evaluate your use of structs against class files.

**Creating Properties**

In Visual C#, a *property* is a programming construct that enables client code to get or set the value of private fields within a struct or a class. To consumers of your struct or class, the property behaves like a public field. Within your struct or class, the property is implemented by using accessors, which are a special type of method. A property can include one or both of the following:

* A get accessor to provide read access to a field.
* A set accessor to provide write access to a field.

The following example shows how to implement a property in a struct:

//Implementing a Property   
public struct Coffee   
{   
    private int strength;   
    public int Strength   
    {   
        get { return strength; }   
        set { strength = value; }   
    }   
}

Within the property, the *get* and *set* accessors use the following syntax:

* The get accessor uses the return keyword to return the value of the private field to the caller.
* The set accessor uses a special local variable named value to set the value of the private field. The value variable contains the value provided by the client code when it accessed the property.

The following example shows how to use a property:

//Using a Property   
Coffee coffee1 = new Coffee();   
// The following code invokes the set accessor. coffee1.Strength = 3;   
// The following code invokes the get accessor. int coffeeStrength = coffee1.Strength;

The client code uses the property as if as it was a public field. However, using public properties to expose private fields offers the following advantages over using public fields directly:

* You can use properties to control external access to your fields. A property that includes only a get accessor is read-only, while a property that includes only a set accessor is write-only.

// This is a read-only property.   
public int Strength   
{   
    get { return strength; }   
}   
// This is a write-only property.   
public string Bean   
{   
    set { bean = value; }   
}

* You can change the implementation of properties without affecting client code. For example, you can add validation logic, or call a method instead of reading a field value.

public int Strength   
{   
    get { return strength; }   
    set   
    {   
        if(value < 1)   
        { strength = 1; }   
        else if(value > 5)   
        { strength = 5; }   
        else { strength = value; }   
      }   
}

* Properties are required for data binding in WPF. For example, you can bind controls to property values, but you cannot bind controls to field values.

When you want to create a property that simply gets and sets the value of a private field without performing any additional logic, you can use an abbreviated syntax.

To create a property that reads and writes to a private field, you can use the following syntax:

public int Strength { get; set; }

* To create a property that reads from a private field, you can use the following syntax:

public int Strength { get; }

* To create a property that writes to a private field, you can use the following syntax:

public int Strength { set; }

In each case, the compiler will implicitly create a private field and map it to your property. These are known as auto-implemented properties. You can change the implementation of your property at any time.

**Creating Indexers**

In some scenarios, you might want to use a struct or a class as a container for an array of values. For example, you might create a struct to represent the beverages available at a coffee shop. The struct might use an array of strings to store the list of beverages.

The following example shows a struct that includes an array:

//Creating a Struct that Includes an Array   
public struct Menu   
{   
    public string[] beverages;   
    public Menu(string bev1, string bev2)   
    {   
        beverages = new string[] { "Americano", "Café au Lait", "Café Macchiato", "Cappuccino", "Espresso" };   
    }  
}

When you expose the array as a public field, you would use the following syntax to retrieve beverages from the list:

//Accessing Array Items Directly   
Menu myMenu = new Menu();   
string firstDrink = myMenu.beverages[0];

A more intuitive approach would be if you could access the first item from the menu by using the syntax myMenu[0]. You can do this by creating an indexer. An indexer is similar to a property, in that it uses get and set accessors to control access to a field. More importantly, an indexer enables you to access collection members directly from the name of the containing struct or class by providing an integer index value. To declare an indexer, you use the this keyword, which indicates that the property will be accessed by using the name of the struct instance.

The following example shows how to define an indexer for a struct:

//Creating an Indexer   
public struct Menu   
{   
    private string[] beverages;   
    // This is the indexer.   
    public string this[int index]   
    {   
        get { return this.beverages[index]; }   
        set { this.beverages[index] = value; }   
    }   
    // Enable client code to determine the size of the collection.   
    public int Length   
    {   
        get { return beverages.Length; }   
    }   
}

When you use an indexer to expose the array, you use the following syntax to retrieve the beverages from the list:

//Accessing Array Items by Using an Indexer   
Menu myMenu = new Menu();   
string firstDrink = myMenu[0];   
int numberOfChoices = myMenu.Length;

Just like a property, you can customize the get and set accessors in an indexer without affecting client code. You can create a read-only indexer by including only a get accessor, and you can create a write-only indexer by including only a set accessor.