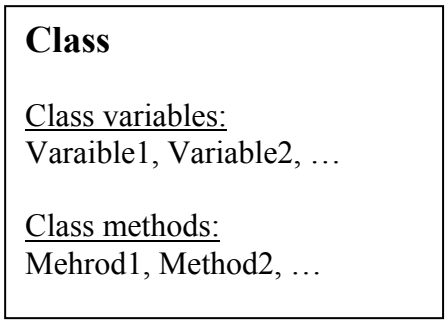
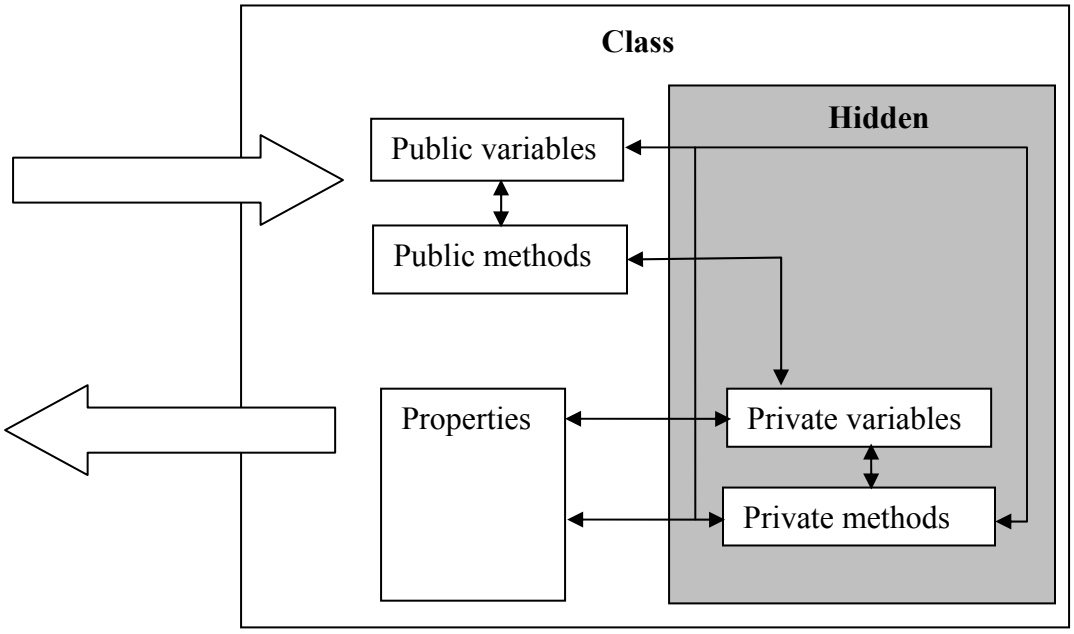
Classes and Structs

Introduction to Classes and Structs

A "class" is the template for creating objects in C#. A "struct" is similar to a class, but is value type as opposed to the class which is reference type. A typical class contains a set of variables and associated methods.



Some of the variables and methods might be public, and some might be private. The public variables and methods can be accessed from outside the class, but the private variables and methods are accessible only from inside the class.



Classes can be instantiated, which means that one can produce difference instances of the same class. Since the class is just a template, it is only when an instance of a class is created that memory gets allocated.

As an example of creating a class, let us create the Ladder class described in the introduction. Let us give this class three variables: name, material and height. Also, let us have a method in the class that calculates the number of steps. Finally, let us force the materials to be selected from a list of three material types: wood, steel, aluminum. To construct this class, start a new application in Visual Studio and call it HardwareStore and insert in it a new class with the name Ladder. Visual studio will create a skeleton for the Ladder class that looks like this

```
namespace HardwareStore
{
	/// <summary>
	/// Summary description for Ladder.
	/// </summary>
	public class Ladder
	{
		public Ladder()
		{
			//
			// TODO: Add constructor logic here
			//
		}

	}
}
```

Let us take a look at what Visual Studio did for us. In the namespace of our application HardwareStore, Visual Studio created a public class Ladder and a method with the same name Ladder. The class was put in the namespace HardwareStore because it is our application namespace and will make this class available for use in other parts of our application. The actual class declaration is only this part:

```
public class Ladder
{
	public Ladder()
	{
		//
		// TODO: Add constructor logic here
		//
	}

}
```

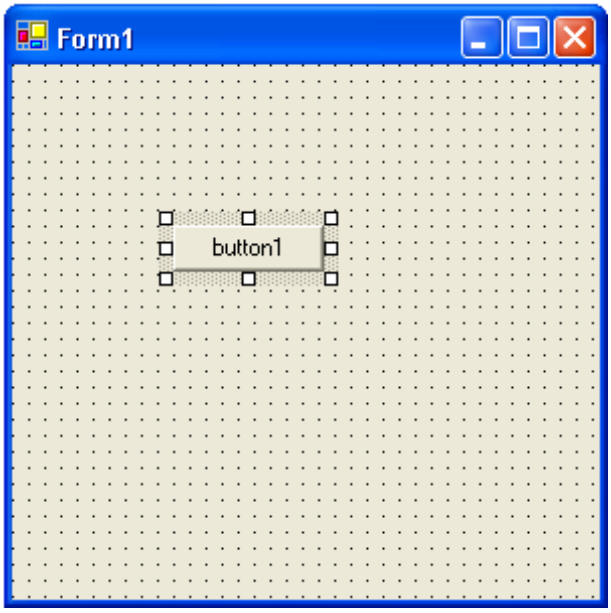
The method Ladder() inside the class Ladder is the constructor of the class. The constructor is used to provide any initialization when creating an instance of the class. We will soon see how this works. For now, add the following to the class declaration.

```
namespace HardwareStore
{
	/// <summary>
	/// Summary description for Ladder.
	/// </summary>
	public class Ladder
```

```
        {
                public string  name;
                public MaterialType material;
                public double height;
                public Ladder()
                {
                        //
                        // TODO: Add constructor logic here
                        //
                        material = MaterialType.wood;
                        height = 5;
                }
                public double Steps()
                {
                        return this.height*12.0/8.0;
                }
                public enum MaterialType
                {
                        wood, steel, aluminum
                }
        }
}
```

Note that we have added three variables to the class, a string type name , a MaterialType type material, and a double type height. The string type and double type are described under standard variable types, but the MaterialType type is an “enum” that we ourselves have declared as having three possible values: wood, steel, aluminum. Enums are used when we need to select from a fixed number of choices. Also note that in our constructor method Ladder(), we have selected to make the default material type to be wood and the default ladder height to be five feet. These will be set every time we create a new instance of the Ladder class. Also note that we have another method declared called Steps that provides us the number of steps.

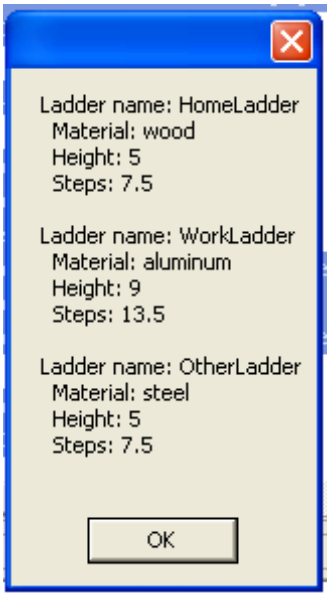
Now that the Ladder class has been declared, let us go back to the HardwareStore Form1 and add a button to it so that we get



Double click on the button to get the method that is run when the button is clicked and create an array of three Ladders as shown and output the results to the MessageBox.

```
Ladder[] myLadders = new Ladder[3];
for(int i=0; i<3; i++)
{
	myLadders[i] = new Ladder();
}
myLadders[0].name ="HomeLadder";
myLadders[1].name = "WorkLadder";
myLadders[1].height = 9;
myLadders[1].material = Ladder.MaterialType.aluminum;
myLadders[2].name = "OtherLadder";
myLadders[2].material = Ladder.MaterialType.steel;

string ladderInfo = "";
for(int i=0; i<3; i++)
{
	ladderInfo += "Ladder name: "+myLadders[i].name +"\n";
	ladderInfo += "  Material: "+myLadders[i].material.ToString() +"\n";
	ladderInfo += "  Height: "+myLadders[i].height.ToString() +"\n";
	double steps = myLadders[i].Steps();
	ladderInfo += "  Steps: "+steps.ToString() +"\n";
	ladderInfo += "\n";
}
MessageBox.Show(ladderInfo);
```

If you build and run the application, when you press the button you will get the following:



You will note several things. First, we created the array of three Ladders through the command line:

```
Ladder[] myLadders = new Ladder[3];
```

At this point no memory is allocated to the three Ladders. Next, we initialized each one of the three myLadders[0], myLaddrs[1], and myLadders[2] through the use of the for loop:

```
for(int i=0; i<3; i++)
{
      myLadders[i] = new Ladder();
}
```

This step is essential since C# will not allocate memory until we initialize each of the three myLadders. If we had tried to assign values to any of the variable of these three Ladders before we did this, the compiler would return an error. Finally, we assigned each Ladder a name and set other parameters. You will note that even though we did not set the material or height of the myLadders[0], when we instantiated myLadders[0] the default values of wood and 5 feet were automatically assigned to these variables.

Let us now take a more formal look at the class and struct in C#.

Class declaration

The minimum declaration for a class is as follows:

```
public class ClassName
{
      public ClassName()
      {
```



```
            //
            // TODO: Add constructor logic here
            //
        }
}
```

where *ClassName* is any name you choose for the class. Each class contains at least one method of the same name which is its constructor method. In this case, "public class *ClassName*" is the declaration of the class and "public *ClassName*()" is the constructor method.

Once a class is declared, objects of this type can be created and used. The process of creating an object of a particular class type is called instantiation and, and as in the case of variables, is a two step process. First a variable of the class type is declared and then an instance of it is instantiated. For example, if we wanted to create an instance of our example class Ladder, with the name myLadder, we would do this with the following code:

```
Ladder myLadder;
myLadder = new Ladder();
```

This process can also be done in one line with the code:

```
Ladder myLadder = new Ladder();
```

Variables and their scope

We have already seen the use of three different variables in our Ladder class. The declaration of these variables was put inside the class, but outside all methods. We also declared each variable to be public. That is, we instructed the compiler that all objects accessing an instance of this class can have access to these variables.

As a rule, a variable that is declared within a structure, is accessible from within it and from all other structures that reside within the original structure. Typically a structure is marked by "{}" so that the variable X declared within structure B in the following example is accessible from within structure B and C, but not from A

```
{ Structure A
        { Structure B
                Varaible X
                { Structure C
                }
        }
}
```

Methods

Methods are functions that provide functionality to the class. We have already seen the step method that returns the calculated number of steps for our Ladder. In general the declaration of a method is as follows:

MethodScope Keyword ReturnType MethodName(*VariabelType Argument1Name*, …)

The *MethodScope* can be *public* or *private*. The possible *Kywords* are *static, virtual, abstract, override, external*. The *ReturnType* may be any type such as *int, double*, …, or any user defined types.

If a method is declared as *public*, it can be accessed from outside the class, but if defined as *private* it can only be accessed from methods inside the class.

If a method is declared as *static*, it will be available without the need to instantiate a member of the class. You can directly access it using the class name, and therefore cannot use variables that would only be created with the instance of a class.

The arguments of a method can be prefixed with the *out* or *ref* modifiers. These are used to control how variables are transferred to the method. If the *ref* modifier is used, then only the address of the variable is transferred to the method and so any changes done to the variable in the method will effect the original value of the variable in the program calling the method. If the modifier *out* is used, the method will not need the variable to be instantiated as its value before entering the method is not used. The prefix *ref* and *out* when added to the declaration of a method, must also be used when calling the method. For example, a method declared as:

```
private void myMethod( ref double myVariable)
```

when called must be called as:

```
double x;
myMehtod(ref x);
```

Properties

Properties are a convenient way of controlling access to the variables of class and also for implementing functionality in this access. The typical property declaration in C# would be as follows:

```
public class ClassName
{
      private VariableType variable;
      public VariableType PropertyName
      {
            get
```



```
            {
                  return variable;
            }
            set
            {
                  variable = value;
            }
      }
      public ClassName()
      {
            //
            // TODO: Add constructor logic here
            //
      }
}
```

In this case, a variable that is private is allowed to be accessed through the property, either to set its value or to retrieve its value. If the variable is not to be changed, the set clause can be removed. If the variable is not to be retrieved, the get clause can be removed. Other functionality can be added by adding code within the get or set "{}".

If in our Ladder example, the material of the ladder was to be determined based on the height, we could provide this type of functionality by the following code:

```
public class Ladder
{
      public string  name;
      private MaterialType material;
      private double height;

      public double Height
      {
            get
            {
                  return height;
            }
            set
            {
                  height = value;
                  if(height < 8)
                  {
                        material = MaterialType.wood;
                  }
                  else if( height < 12 )
                  {
                              material = MaterialType.aluminum;
                  }
                  else
                  {
                        material = MaterialType.steel;
                  }
            }
      }

      public MaterialType Material
```

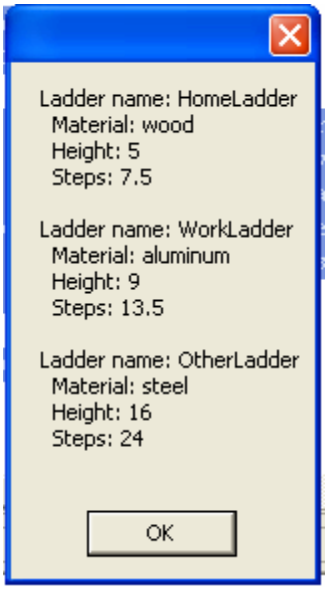
```
        {
                get
                {
                        return material;
                }
        }

        public Ladder()
        {
                //
                // TODO: Add constructor logic here
                //
                material = MaterialType.wood;
                height = 5;
        }
        public double Steps()
        {
                return this.height*12.0/8.0;
        }
        public enum MaterialType
        {
                wood, steel, aluminum
        }
}
```

In this example, since we have made material and height private variables, they are no longer accessible by other elements of the application. To set the height, in this case we need to use the property “Height,” which in the process of setting the height, also selects the material. Since the material is being selected by the selection of the height, only a get method is provided for the “Material” property.

In our Form1 button event code replace the previous code by the following:

```
Ladder[] myLadders = new Ladder[3];
for(int i=0; i<3; i++)
{
        myLadders[i] = new Ladder();
}
myLadders[0].name ="HomeLadder";
myLadders[1].name = "WorkLadder";
myLadders[1].Height = 9;
myLadders[2].name = "OtherLadder";
myLadders[2].Height = 16;

string ladderInfo = "";
for(int i=0; i<3; i++)
{
        ladderInfo += "Ladder name: "+myLadders[i].name +"\n";
        ladderInfo += "  Material: "+myLadders[i].Material.ToString() +"\n";
        ladderInfo += "  Height: "+myLadders[i].Height.ToString() +"\n";
        double steps = myLadders[i].Steps();
        ladderInfo += "  Steps: "+steps.ToString() +"\n";
        ladderInfo += "\n";
}
MessageBox.Show(ladderInfo);
```

The output of your application will now look as follows:



The constructor of a class can be overloaded, which means we can have several constructors, each with a different set of arguments. For example, if we wanted to have the option to set the material and height during the instantiation, we could redo the code for our class as follows:

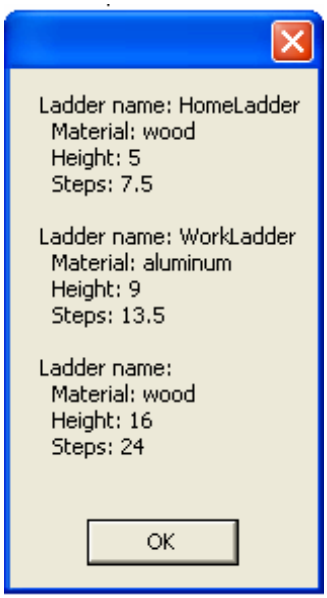
```
public Ladder()
{
      //
      // TODO: Add constructor logic here
      //
      material = MaterialType.wood;
      height = 5;
}
public Ladder(MaterialType LadderMaterial, double LadderHeight)
{
      material = LadderMaterial;
      height = LadderHeight;
}
```

The compiler will select the constructor to be used based on what the arguments are. As long as the arguments have a unique signature (number of arguments and types are not the same), then the compiler can choose the proper constructer to use based on the arguments provided. Now change the code in the application button event as follows:

```
for(int i=0; i<2; i++)
{
      myLadders[i] = new Ladder();
}
myLadders[0].name ="HomeLadder";
myLadders[1].name = "WorkLadder";
myLadders[1].Height = 9;
```

```
myLadders[2] = new Ladder(Ladder.MaterialType.wood, 16);
```

Note that the myLadders[2] will use our new constructor because of the arguments provided. The result of running this new application will be:



Classes are reference type

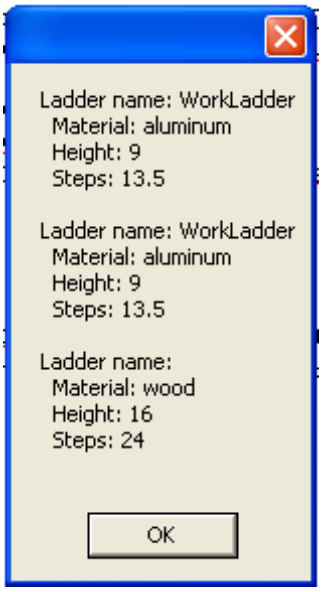
One thing to be aware of is that classes are of reference type. That is, they have an address stored in the stack, but the actual class variables are stored in the heap. When we assign one instance of Ladder to another, the only thing that happens is that the address is copied, so that both instances now point to the same location in the heap. As a result, when you make a change to variables of one of these classes, the value for the other class also automatically changes since it is pointing to the same memory location.

As an example, change the Hardware store button event click method by adding the one line "`myLadders[1] = myLadders[0];`" as follows:

```
Ladder[] myLadders = new Ladder[3];
for(int i=0; i<2; i++)
{
      myLadders[i] = new Ladder();
}
myLadders[0].name ="HomeLadder";

myLadders[1] = myLadders[0];

myLadders[1].name = "WorkLadder";
myLadders[1].Height = 9;
myLadders[2] = new Ladder(Ladder.MaterialType.wood, 16);

string ladderInfo = "";
```

```
for(int i=0; i<3; i++)
{
      ladderInfo += "Ladder name: "+myLadders[i].name +"\n";
      ladderInfo += "  Material: "+myLadders[i].Material.ToString()
+"\n";
      ladderInfo += "  Height: "+myLadders[i].Height.ToString() +"\n";
      double steps = myLadders[i].Steps();
      ladderInfo += "  Steps: "+steps.ToString() +"\n";
      ladderInfo += "\n";
}
MessageBox.Show(ladderInfo);
```

Since myLadders[1] and myLadders[0] are now both pointing to the same memory location, any changes to either will effect the value for both of them. The result of running this program will be:



As can be seen, both myLadders[0] and myLadders[1] have the same values.

Frequently, when we assign one instance of a class to another, as we did for myLadders[0] and myLadders[1], we want the new object to be a copy of the original. This can be done through defining a special method for the class to copy the contents into a new instance of the object.

Inheritance

Inheritance is a powerful way of duplicating the variables and methods of one class in another class without writing any additional code. For example, if we had a HardwareItem class that included itemPrice as a variable and Tax as a method, we could have our Ladder class inherit from it and then each Ladder would have an itemPrice and a method to calculate tax without really writing any additional code. The syntax of introducing inheritance in a class is as follows:

```
public class ClassName : ParentClassName
{
	public ClassName()
	{
		//
		// TODO: Add constructor logic here
		//
	}
}
```

Note that the only difference here is the addition of ": *ParentClassName*" to indicate that the class is inheriting from another class. There is only single inheritance in C# so that a class can only inherit from one other class, even though the class it inherits from can inherit from another class, and so one.

Following our example, create a new class named HardwareItem and add change it to look as follows:

```
public class HardwareItem
{
	private double itemPrice;
	public double ItemPrice
	{
		get
		{
			return itemPrice;
		}
		set
		{
			itemPrice = value;
		}
	}
	public HardwareItem()
	{
		//
		// TODO: Add constructor logic here
		//
	}
	public double Tax()
	{
		return itemPrice*0.067;
	}
}
```

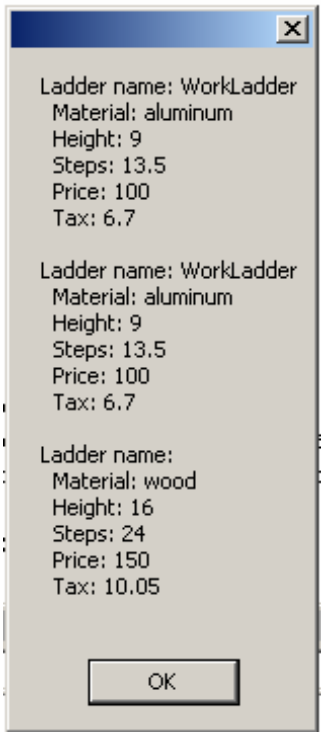
Also, let us change the declaration of the Ladder class to

```
public class Ladder : HardwareItem
```

Now in the click event method of your button on Form1, add a price for each of myLadders. Note that automatically a ItemPrice property has been added. If you also add the lines to display each Ladder's price and tax, you will have a code as follows:

```
Ladder[] myLadders = new Ladder[3];
for(int i=0; i<2; i++)
{
       myLadders[i] = new Ladder();
}
myLadders[0].name ="HomeLadder";

myLadders[0].ItemPrice = 100;

myLadders[1] = myLadders[0];
myLadders[1].name = "WorkLadder";
myLadders[1].Height = 9;
myLadders[2] = new Ladder(Ladder.MaterialType.wood, 16);

myLadders[2].ItemPrice = 150;

string ladderInfo = "";
for(int i=0; i<3; i++)
{
       ladderInfo += "Ladder name: "+myLadders[i].name +"\n";
       ladderInfo += "  Material: "+myLadders[i].Material.ToString() +"\n";
       ladderInfo += "  Height: "+myLadders[i].Height.ToString() +"\n";
       double steps = myLadders[i].Steps();
       ladderInfo += "  Steps: "+steps.ToString() +"\n";

       ladderInfo += "  Price: "+myLadders[i].ItemPrice.ToString() +"\n";
       ladderInfo += "  Tax: "+myLadders[i].Tax().ToString() +"\n";

       ladderInfo += "\n";
}
MessageBox.Show(ladderInfo);
```

The result of running the program will now be:



Enum

Enums are ordered sets of constants. They are used when a fixed number of choices are available. As in the example above, an *enum* is declared in a class, just like a method, and is accessed through the instance of the class.

Structs

Structs are similar to classes, but are value type. A default constructor is created for the struct so there is no need to create a constructor. Structs are intended for small data structures since every time you assign a struct to another struct, all the struct must be copied. This is in contrast to the class which is a reference type and when assigned only the address is copied.

