**Lesson 2 Notes**

**Assembly –** a deployment unit. Represents one file on disk. If you have 4 projects in your solution, there will be 4 assemblys.

**Namespace**

You can have same types within different assemblies. Problem is when you try to use two different assemblies with the same type. To limit the possibility of this, you use namespaces. It’s sole purpose is for scoping.

A type is unique as far as it’s fully qualified name is unique. Fully qualified name is namespace + type name. The compiler always works with fully qualified namespaces.

Namespaces are always pascal cased. They follow identifier rules.

The defacto standard is: company name.product[.subnamespaces]

Company types are included in company namespace. Product types are in the product namespace.

Generally, the namespace name follows the assembly name.

The compiler always required the fully qualified type name. The compiler has a shortcut so you don’t have to use the FQTN every time.

Using name;

Now you can just use ‘product’ in your code.

Example. Int32 is not the FQTN. It is actually System.Int32. So you would put ‘using System’ a the top of the page. They are USING DECLARATIONS.

Using Keyword followed by Namespace name;

Namespace Nile.Host (contains two namespaces… Nile… and sub namespace Host).

To create a namespace, you create a namespace statement.

Namespace ‘name of namespace’

{

}

You can also nest namespaces; but, it isn’t advisable.

Projects talk to each other through assemblies

Declaring a class:

Class T

{

}

This declares the existence of this type with this name. Instead of the curly braces are the members of the class.

Class

Instance/Object

A class is a template (a blueprint). It describes functionalities.

An Instance is the actual creation of that class (type).

Rectangle

Length

Width

A Rectangle class would declare that there is a length and a width. You could also have a function to calculate the area given the length and width.

You could have a function to draw the rectangle.

The new keyboard takes a type and creates and instance of it in memory. So your variable is storing an instance of the class.

Classes are all about grouping together the functionality and properties of something.

Classes are almost always nouns. Are always pascal case. The fields are camel cased.

Class definition

[Access] class T

Types can only be one of two accessibility types. **Public or Internal.**

**Accessibility Types:**

**Public –** anyone can use it.

**Internal –** anything in the assembly can use it.

**Private –** only the type (class) can use it.

Accessibility determines who has access at compile time.

**Variables that are a part of a class are referred to as fields**

**(Class Members)**

**Fields**

**Properties  
Methods**

**Functions**

[access] T funcId([params])

Public Functions are Pascal Cased

Functions are always named using verbs because they are actions

(look up methods)

Inside functions, there is always an initial parameter, even when the signature is empty. It is the This parameter. It represents the calling instance. It gives you access to the members of the instance.

So the following two functions are the same:

Public decimal GetPrice()

Public decimal GetPrice(this)

**A function cannot be a member of a type (class)**

**So functions inside of a class is a method.**

**Fields represent data**

**Methods represent functionality**

**Properties**

[access] T id

{

Get{}

Set{}

}

Get and Set are methods

You have no control over what a user does with a field. Fields work just like regular variables.

Properties are a mix of fields and methods. They have the syntax of fields. But they have the behavior of methods. When reading a property, you are calling a method. The user sees them as fields; but, we have control over what is written because they are actually methods.

Fields store data. So if you need to save data through a property, you have to use a field. Create a private field in order to save these fields. **Fields are always camel cased.**  Start private fields with an underscore. Parameters will often have similar names as private fields. Using an underscore helps to keep these from being confused.

Data Property: Many properties are used to store data. This is where the private fields come in . These are referred to as backing fields.

Properties allow you to manipulate the data. Fields cannot do this.

If you are going to expose data, use properties, not fields.

Properties can be any kind of accessor. They are usually public, but can be any kind.

Summary and value are the only kinds of descriptors (look up) you can put on properties.

Public decimal Price

{

Get {return \_price;}

Set {price = \_value;}

}

This is so used, they have come up with a short cut called auto properties:

Public decimal Price {get; set;}

You cannot use the short cut if you do any kind of modification in the property.

If you want to initialize a field backed by a property, use equal and the value after the get; set; You can only do this on auto properties.

Ex: public decimal Price {get; set; } = 0;

Constant values are baked into source code. The only way to change the value is to recompile the code.

Constants are one field you might want to make public. A property cannot be a constant because it exposes methods.

Public decimal DiscontinuedDiscountRate = 0.10M;

Look up difference between CONST and ReadOnly as far as compile time, etc.

You don’t have to initialize a ReadOnly. The main use for ReadOnly is for setting things? That are expensive… like database connections.

|  |  |  |
| --- | --- | --- |
| **Property** | Method |  |
| Fast |  |  |
| Gets – no side effects | Get – side effect |  |
|  |  |  |
|  |  |  |

DateTime.Now() called 3 times would have a side effect. Properties return the same value over and over until something happens to change the value…. By calling the setter. DateTime.Now would be a method because calling it 3 times would have 3 different results.

If the member you are trying to represent is not data, use a method. Properties represent data.

If you need parameters, use a method. Properties only have one ???.

If in doubt, use methods and not properties.

If a good amount of memory is going to be used, do a method, not a property.

Methods are designed to perform a function.

**WinForms**

Don’t leave cs file and design file open at the same time

The Text property is the Title of the Form

Controls are designed to be hosted by a form. Each control is a class.

Control is the base class of every UI element on the screen. Including the actual form. Every control has a Text Property.

Every form has a parent. The main form’s parent is the desktop for Windows.

Modal – the user cannot get back to the parent window. They are blocked. Used when you need to force the user to do something. (ShowDialog())

Modeless – allows user to work with other windows while open. (Show)

Specify parent window in showdialog. You need to use this. This is the current form.

Name is a property that every control has. It serves several purposes. It specifies the name of the control. Has to be unique within the form. It also the name of the field that gets generated to back the control. Every name generates a private field in the designer.

ShowDialog() 🡪 **DialogResult** (values such as ok, cancel, yes/no)

Do not set DialogResult property through the properties window of the designer. It causes problems. Do it in the OnSave Method.

This.DialogResult = DialogResult.OK;

**Constructors look like methods. It has the same name as the class. There is also no return type.**

Every type has at least a default constructor.

Var obj = new T()

Three statements:

1. new – allocates memory at compile time… and sets aside enough memory to store an instance of the type.
2. initialize fields
3. Ctor (call the constructor) (Ctor is not a typo. It stands for Call Constructor)

Constructors give you a chance to initialize the object. Generally, you are just setting up fields to have values. Once initialized, you get back a fully constructed object.

Constructors either succeed or fail. This is part of why you do minimal work in constructors.

The preferred approach is an initializer field and not a constructor. Only create a constructor if you need one… and you usually don’t need one. Ex: public decimal Price { get; set; } = 0; This initializer field keeps you from needing a constructor.

There are a few situations where you do need Constructors:

1. Cross-field initialization. Rare, but does happen.
2. If you have to use a field ???

Constructors are run starting fgrom the base type, then the next type, then the next type, ,then your type. This causes problems when you use the ‘this’ keyword.

Constructors follow same rules for overloading and parameters as all other methods.

The default constructor is automatically there for types that do not have a constructor. But it is no longer available if a user-entered constructor is implemented.

# - pre-processor. Look up.

Base means ‘my base type’.

Protected override :

**Override** tells the compiler this method is already defined in a base Type. But I am overriding that base definition. Don’t use the Base definition…. Use my definition.

**Protected**  - only valid in inheritance. Derived types see it as public. Everyone else sees it as private.

(look up protected)

**Protected Internal**  - very rare. Used a lot in libraries.

Not all members of a type can be overwritten. Typing override first shows you the members that can be overwritten.

A virtual member is a member that is designed to be overwritten. (look this up) Can be applied to properties and methods. Rarely do virtual properties. Don’t mark things virtual unless you need them to be. It is a runtime hit.

When you type override, it pulls back all members marked as virtual.

You can use abstract instead of virtual. They do the same thing. They identify a member as being overridable. Look up the differences!

Never call a virtual method inside a constructor! The correct way to initialize types is to put initializing in the constructor.

Constructor chaining…. You can have your constructor call other

Value types can never have a base type. They do not inherit! They have default constructors, and you cannot change them. You must explicitly set every field’s value before calling ???

Command (String)

1. Class HelloCommand : Command
   1. Public HelloCommand() : base (“Hello”)

{

}

Virtual – you may override and/or use base implementation.

Abstract – you must override implementation.

Delegate represents the parameters and return type. The name does not matter.

Delete = void (object, EventArgs)

Edit= void (object, EventArgs)

Add = void (object, EventArgs)

All parameters and return types are the same. Only the name is different.

Delegates Provide you a way to execute methods you don’t necessarily know about until run time. Delegates only work with functions.

Declare a function type:

Public delegate void ButtonClickCall (object sender, EventArgs eventArgs );

Note: ButtonClickCall is now a Type

Private void CallButton (ButtonClickCall functionToCall )

{

functionToCall(this, EventArgs.Empty);

}

When you use a function name by itself (no parentheses), it is always a delegate. When you pass the function, don’t put the parentheses on the name.

Look up Is and At keywords

1. Event – Notification between objects
2. Call Onsave EventHandler when button1 clicked

Button1.click += OnSave;

OnSave() is a method call

OnSave is a delegate. Any method that matches the signature of a delegate can be used.

EventHandlers all have the same signature (object sender, EventArgs e). They all return void.

**Form Life cycle**

Ctor (Call Constructor)

**Load**

1. Called only once… very first time form is loaded.
2. First render. Point at which form is ready to go on the screen.
3. Place where you set up the UI explicitly.

**Form Closing** – form still on screen (pre closed validation)

Form is giving the opportunity to stop the closing. You can use CancelEventArgs to stop a form from closing.

**FormClosed –** form is gone. Chance to do any cleanup after form has been removed from the screen.

WinForms uses the convention of On[Method Name] for events. Ie. OnFormClosing, OnLoad, etc. It’s approach is to keep developers from having to wire up an eventhandler (button1.click += OnSave).

NEVER create a derived types for an event. Instead, just use the event. If y ou want to extend what happens when something occurs, you will override the event.

An override of an event method occurs before the actual event happens.

**--Quiz Notes?**

1. **Validate Textbox**
   1. **[Error provider]**
   2. **Validating**
   3. **ValidateChildren**
2. **Ensure Control Remains**
   1. **Fixed distance from bottom/right corner 🡪 Anchor = Bottom | Right**
   2. **Dock**