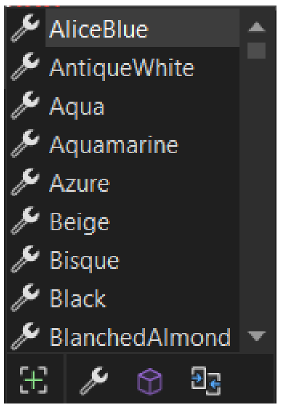
Enumerations

- An *enumeration* (or *enumerated type*) lets you define a new data type that can hold only certain whole number values (constants). Each constant is known as a *member* of the enumeration and has its own name.
- Enumerations are value types – when assigned to a variable name, they are treated like any of the intrinsic data types. If an enumeration is passed to a method, it is passed by value – a copy is made and used in the called method.
- The enumerations provided by .NET are used mostly to set object properties and to specify the values that are passed to methods.

Example:

```
//set the color of text box in code using the Colors enumeration

txtTextbox.Background = new SolidColorBrush(Colors.AliceBlue);
```



- Since an enumeration member represents a whole number – you can store the enumeration member to an integer it in an `int` variable…

```
int red = Colors.AliceBlue.R;
int blue = Colors.AliceBlue.B;
int green = Colors.AliceBlue.G;
int alpha = Colors.AliceBlue.A;
```

Creating and Using an Enumeration

- Enumerations are usually declared at the namespace level – outside of any class – and are declared public.

General Form:

```
enum EnumerationName : type
{
    ConstantName1 = value1,
    ConstantName2 = value2,
    …
    ConstantNameN = valueN
}
```

Notes:
: type – optional
- Specify a whole number data type

= value – optional
- If values are not specified, the first constant is assigned the value 0, the next is assigned the value 1, and so on…

General Usage of an Enumeration:

```
// declare an enumeration data type variable and assign it one of the constants in the enumeration
EnumerationName variableName = EnumerationName.ConstantName;

// using a selection structure with an enumeration
if(variableName == EnumerationName.ConstantName)
{
    // code
}
```

Example: Code Comparison – Using vs. Not Using an Enumeration

```
/****************************************
 *               No Enumeration                *
 ****************************************/
private void NoEnum_Click(object sender, EventArgs e)
{
    int num1 = 3, num2 = 2;

    // addition
    int result = PerformOp(0, num1, num2);
    string display = "Addition: " + result + "\n";

    // subtraction
    result = PerformOp(1, num1, num2);
    display += "Subtraction: " + result + "\n";

    // multiplication
    result = PerformOp(2, num1, num2);
    display += "Multiplication: " + result + "\n";

    txtOutput.Text = display;
}

// method not using enumerated values
private int PerformOp(int op, int value1, int value2)
{
    int result = 0;
    switch (op)
    {
        case 0: // addition
            result = value1 + value2;
            break;
        case 1: // subtraction
            result = value1 - value2;
            break;
        case 2: // multiplication
            result = value1 * value2;
            break;
    }
    return result;
}
```

Notes:

1. The PerformOp( ) method is overloaded.
2. Enumerations are value types – when passed to a method, a copy of the original value is used – just like the intrinsic data types.
3. An *if* could be used instead of a *switch*
   ex. `if(op == Operation.Addition)`
4. An alternate (and equivalent) way to define the enumeration:

```
public enum Operation : int
{
    Addition = 0,
    Subtraction = 1,
    Multiplication = 2
}
```

```
/****************************************
 *               Using an Enumeration               *
 ****************************************/
private void Enum_Click(object sender, EventArgs e)
{
    int num1 = 3, num2 = 2;

    // addition
    int result = PerformOp(Operation.Addition, num1, num2);
    string display = "Addition: " + result + "\n";

    // subtraction
    result = PerformOp(Operation.Subtraction, num1, num2);
    display += "Subtraction: " + result + "\n";

    // multiplication
    result = PerformOp(Operation.Multiplication, num1, num2);
    display += "Multiplication: " + result + "\n";

    txtOutput.Text = display;
}

// method using enumerated values
// Note: Enumerations are value types - they are passed by value
private int PerformOp(Operation op, int value1, int value2)
{
    int result = 0;
    switch (op)
    {
        case Operation.Addition:
            result = value1 + value2;
            break;
        case Operation.Subtraction:
            result = value1 - value2;
            break;
        case Operation.Multiplication:
            result = value1 * value2;
            break;
    }
    return result;
}

// user defined enumeration
public enum Operation
{
    Addition,         // values start at zero
    Subtraction,      // and increase by one
    Multiplication    // for every member
}
```

Structures

Structures are code blocks that can define anything a class can define – variables, constructors, properties, methods, events, etc.
Structures should be used to primarily *encapsulate* data that is not intended to be modified after the structure is created and initialized.
Structures are typically declared at the *namespace* level – outside of any class block of code.
Structures are value types – when passed to a method or assigned to another variable, a copy is created.
Structures cannot be inherited (inheritance is covered in the discussion on classes). A structure can implement interfaces (covered in the discussion on interfaces).

Defining and Creating a Structure

Structures start with an optional access modifier:
- **internal** (default if not specified): the structure is only accessible by the code in the same project
- **public**: the structure can be accessed by all code
- The private access modifier can only be used by a structure nested (contained) inside another structure or class and this will restrict access of the structure to only the code inside the containing class or structure.

The **struct** keyword is used to identify a block of code as a structure.
Structures then get an identifier (name) that follows the Pascal naming convention (start with an uppercase letter).
Within a structure declaration, fields (structure level variables) cannot be initialized unless they are declared as **const** or **static**.
A structure may not declare a default constructor (a constructor without parameters) or a destructor. Structures can declare constructors that have parameters.
Structure members can use one of the following access modifiers:
- **private** (default if not specified): the member is only accessible by the code in the structure
- **public**: the member can be accessed by all code
- **internal**: the member can only be accessed by code in the same project

Unlike classes, structures can be instantiated (copied) without using a new operator – this is the case when you do not declare a constructor.
If there are any private variables in the structure, only a constructor can initialize these values. To create an instance of a structure with private variables, the *new* operator is used followed by a call to the constructor. Parameterless constructors are not allowed for structures – one is supplied by C# and cannot be changed.

General Form:

```
struct StructureName
{
    // declare structure level variables (fields) and constants

    // declare constructor(s) – optional but required for private field initialization

    // declare properties – optional but recommended

    // declare methods – optional
}
```

Static vs. Nonstatic Members

Static members are declared with the keyword **static** after an optional access modifier.
Static members are shared between all instances (copies) of a structure and are accessed using the name of the structure.
Constants are static even though the static keyword is not used.
Static fields can (and should) be initialized.

```
// code inside the structure
public static int count = 0;      // declares a public static field that is shared with all instances of this structure
// code inside another class/structure
StructureName.count++;            // increments the static field value by one
```

Nonstatic members are declared with just an optional access modifier.
Every instance of a structure will get their own copy of a nonstatic member.
Nonstatic fields (known as **instance** variables) cannot be initialized until a structure object is created.

```
// code inside the structure
public int count;             // declares a public nonstatic field – every instance of the structure will have its own copy of this
    variable
// code inside another class/structure
StructureName objectName;     // declares an instance of the structure with an object name
objectName.count++;           // increments this particular instance of the variable by one
```

Fields

Fields are the structure level variables accessible to all the code in the structure.
Fields are typically declared private so they cannot be accessed directly outside the structure – this is known as *information hiding*.
Public properties are defined for each private variable to control how the variable is accessed.
Any nonstatic private or otherwise inaccessible fields can only be initialized in a constructor.
Public fields are used primarily for constants and read-only variables.
Fields of a structure cannot be initialized when declared except when they are declared const or static.
Read-only variables (declared using the keyword **readonly**) are the nonstatic equivalent of a constant – with the exception that they don't get initialized when declared (in a structure). Once their value is set, it cannot be changed.

```
private static int count = 0;          // private static field initialized
public const string Owner = "Cliff";   // public constant initialized – static by default
public readonly string Type;           // public read-only field – initialized after copy of structure is created
private string title;                  // private field – constructor required to initialize
```

Constructors

- Constructors are methods that are typically used to initialize all fields that have not been initialized when declared.
- C# provides a default parameterless constructor that initializes all uninitialized fields with default values (numerical types to zero, booleans to false, object types to *null*)
- Constructors start with the public access modifier – this allows an instance of the structure to be created from anywhere in the program.
- Constructors do not have a return type.
- Constructors are named with the same name as the structure/class name.

General Form:

```
public StructureName(parameter list)
{
    // initialize all uninitialized variables
}
```

Usage:

```
StructureName objectName = new StructureName();   // call the default C# supplied constructor
StructureName objectName = new StructureName(arguments); // call the user defined constructor
```

Properties

Public properties are used to control access to private fields.
Static properties are used to access static fields and nonstatic properties are used to access nonstatic fields.
Properties can code two accessor blocks:

get{ } Accessor: code that will get the value of the private variable – uses a return statement.
set{ } Accessor: code that will set the value of the private variable – uses the keyword **value** as the new value for the private variable.

One of the accessor blocks can be left out to create a read-only property (no *set* accessor) or a write-only property (no *get* accessor).

General Form:

```
public dataType PropertyName
{
    get
    {
        return variableName;
    }
    set
    {
        variableName = value;
    }
}
```

`value`
is a keyword only when used in the *set* accessor block in a property definition
automatically declared and has the same data type as that specified in the property header

Usage:

```
dataType variable = objectName.PropertyName;    // uses the get accessor to get the value represented by the property
objectName.PropertyName = variable;     // uses the set accessor to assign a value to the variable represented by the property
```

Note:

When a struct contains only fields without any constructors or properties the struct can be declared without the new operator and stored directly on the stack:

```
struct Name
{
   public string First;
   public string Last;
}

Name name;
name.First = "Peter";
name.Last = "Vanscoy";
```

Example:

```
// a structure to hold basic movie information
public struct Movie
{
    private static int count = 0;           // private static field initialized
    public const string Owner = "Cliff";    // public constant initialized – static by default
    public string type;                     // public field – initialized by C#s default constructor
    private string title;                   // private field – initialized by C#s default constructor

    // public static property to access private variable 'count'
    public static int MovieCount
    {
        get
        {
            return count;
        }
        set
        {
            // value is automatically declared as an int variable
            count = value;           }
    }
    // public nonstatic property to access private variable 'title'
    public string MovieTitle
    {
        get
        {
            return title;
        }
        set
        {
            // value is automatically declared as a string variable
            if (value.Length == 0)
            {
                title = "Unknown";
            }
            else
            {
                title = value;
            }
        }
    }
}// struct
```