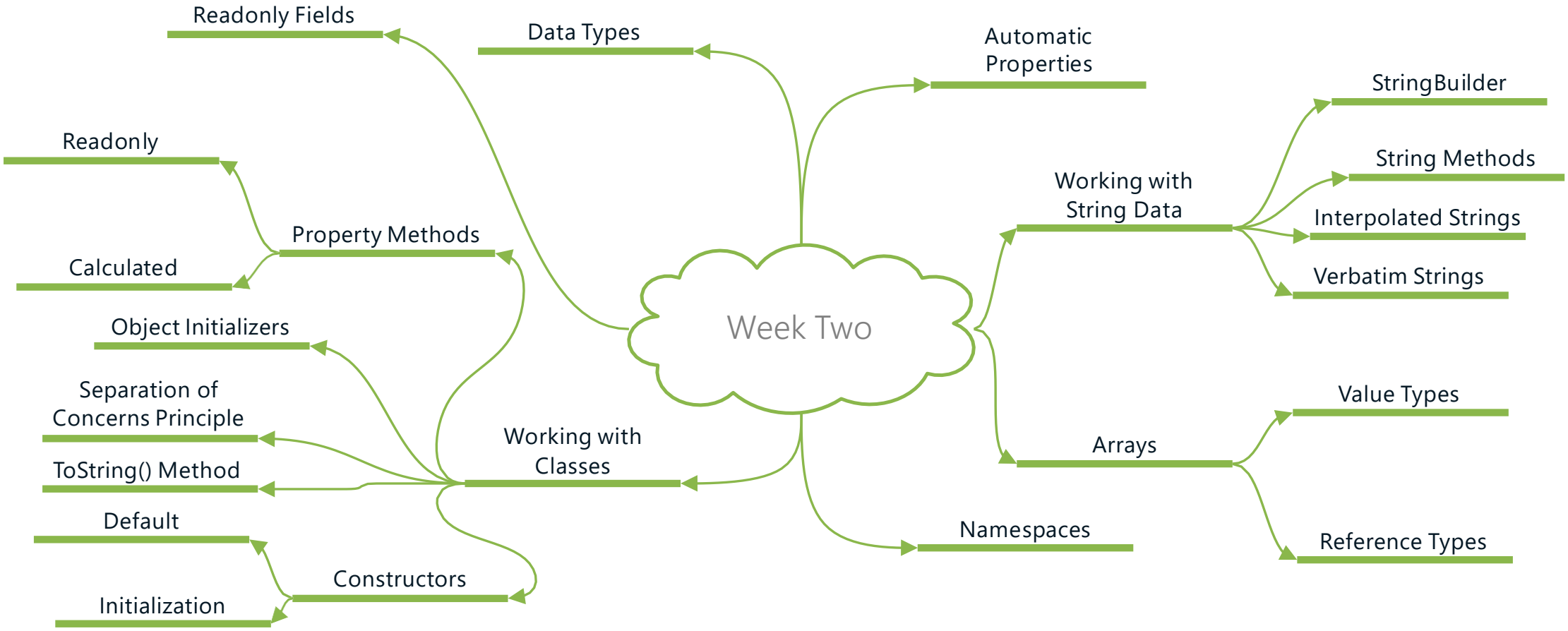
COMP 3602

C# Application Development

Week Two

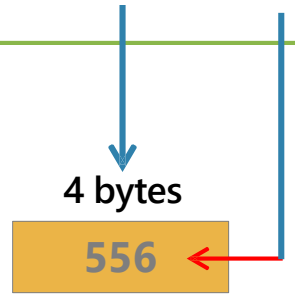
Tonight's Learning Outcomes



Data Types – Value vs Reference

int is a value type

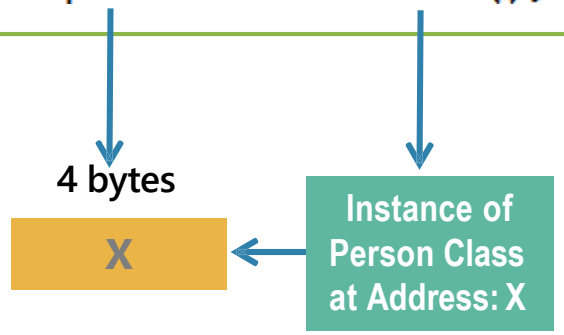
```
int amount = 556;
```



Declaration of amount allocates 4 bytes of memory and is given the label amount.
Assignment stores the value 556 directly at this address.

Person is a reference type

```
Person person = new Person();
```

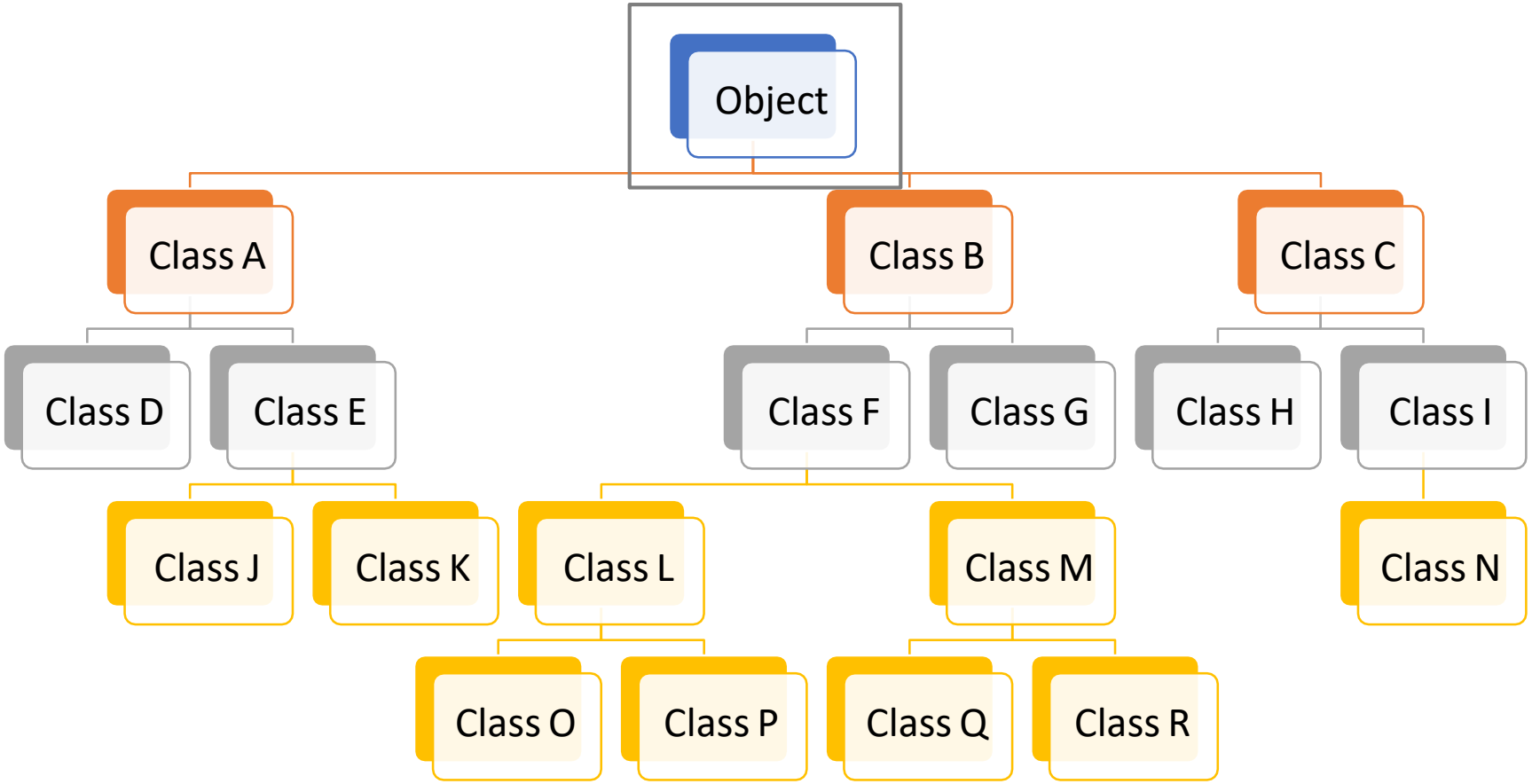


Declaration of person allocates 4 bytes of memory and is given the label person.
The new operator creates an instance of the Person class at address X.
Assignment stores the value of address X at the person label.

Data Types – Other

C# Type Name	.NET Type Name	Value Range	Size (bytes)
bool	System.Boolean	true or false	1
char	System.Char	One Unicode character between 0 and 65,535	2
DateTime	System.DateTime	January 1, 0001 12:00:00am to December 31, 9999 11:59:59pm	8
string	System.String	unlimited sequence of Unicode characters	Varies
object	System.Object	base type of all types	Varies

Base Class Library Object Hierarchy



The Object class sits at the top of the Base Class object hierarchy and serves as the base class for all classes.
All classes can trace their ancestry back to Object.

Working with String Data

String literals are enclosed in double quotes:

```
string name = “Moe”;
```

Char literals are enclosed in single quotes:

```
char target = ‘s’;
```

Precede with @ to specify string as is (WYSIWYG)
aka: Verbatim Strings

```
"C:\\MyFiles\\temp\\Hello.cs”
```

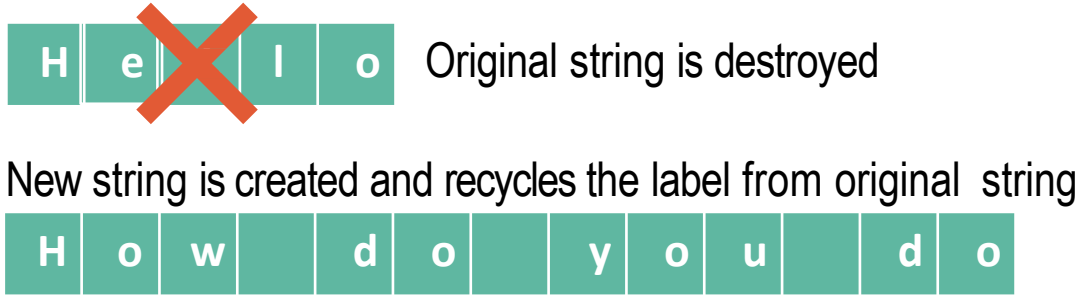
can be written as

```
@"C:\MyFiles\temp\Hello.cs"
```

Also, @ allows multiline strings

Strings are Immutable

```
13
14          string greeting = "Hello";
15
16          Console.WriteLine(greeting.ToUpper());
17          Console.WriteLine(greeting);
18
19          greeting = "How do you do";
20
```



- The string declared on Line 14 is not being altered on Line 19.
- A new string is created with the new value.
- The original string is destroyed and the new string recycles the label of the original string.

StringBuilder

```
13
14      string longLabel = string.Empty;
15      int repetitions = 50000;
16
17      Console.WriteLine("Starting StringBuilder Test");
18
19      StringBuilder sb = new StringBuilder(1000000);
20
21      for (int count = 0; count < repetitions; count++)
22      {
23          sb.Append("text content ");
24      }
25
26      Console.WriteLine(sb.ToString());
27
28      Console.WriteLine("\nPress Enter to Start String Concatenation Test");
29      Console.ReadLine();
30
31      for (int count = 0; count < repetitions; count++)
32      {
33          longLabel += "text content ";
34      }
35
36      Console.WriteLine(longLabel);
37
```

- The StringBuilder class creates a buffer in which to perform string manipulations and concatenations. This eliminates the need to destroy and create new strings every time a change is made.
- Call the StringBuilder ToString() method to assign its buffer contents to a string object.

String concatenation

```
longLabel += "text content ";
```

string longLabel → 0x0FFC: Text Content

First Iteration: Variable longLabel is assigned to a memory location (say 0x0FFC) and the value “Text Content “ is stored there

---

```
longLabel += "text content ";
```

string longLabel → ~~0x0FFC: Text Content~~

string longLabel → 0xD550: Text Content Text Content

Second Iteration: Variable longLabel is assigned to a **new** memory location and the value “Text Content Text Content “ is stored there.

By the 50000th iteration, there will be huge memory blocks that are being allocated and deallocated

StringBuilder

```
StringBuilder sb = new StringBuilder(1000000);
```

StringBuilder sb → 0x045E:

First Iteration: Variable **sb** is assigned to a memory location (say 0x045E) and room for 1000000 characters is allocated (2000000 bytes or around 2mb)

```
sb.Append("text content ");
```

Stringbuilder sb → 0x045E : Text Content

Second Iteration: The memory location that **sb** is assigned has the “Text Content “ stored there.

```
sb.Append("text content ");
```

Stringbuilder sb → 0x045E : Text Content Text Content

Third Iteration: The memory location that **sb** is assigned has the “Text Content “ stored at the end.

string.Format

```
14
15      string name = "Gillian";
16      int numMessages = 1;
17
18      string message = string.Format("Hello {0}, you have {1} {2}."
19                                   , name
20                                   , numMessages
21                                   , numMessages == 1 ? "message" : "messages");
22
23      appendToFile(message);
24
```

- Sometimes, you may want to format a string without outputting to the Console (using Console.WriteLine).
- Use the static method string.Format, which returns a string.
- Takes similar parameters to Console.WriteLine

Interpolated Strings

- An alternate syntax to string.Format
- Preface string with a $
- Placeholders hold actual data instead of index numbers
- Column width and format specifiers same as in string.Format
- Can be used with WYSIWYG strings (use $@ - order was important in previous versions)

```
14
15      decimal amount = 1342.66m;
16      string name = "Bella";
17
18      string output1 = string.Format("Hello {0}, your balance due is {1:N2}"
19                                     , name
20                                     , amount);
21      //becomes
22      string output2 = $"Hello {name}, your balance due is {amount:N2}";
23
```

String Methods

```
13
14      // ToUpper and ToLower
15      string phrase = "How do you do?";
16      Console.WriteLine(phrase);            // How do you do?
17      Console.WriteLine(phrase.ToLower()); // how do you do?
18      Console.WriteLine(phrase.ToUpper()); // HOW DO YOU DO?
19      Console.WriteLine(phrase);            // How do you do?
20
21      // SubString
22      Console.WriteLine(phrase.Substring(0, 3));        // How
23      Console.WriteLine(phrase.Substring(4, 2));        // do
24      Console.WriteLine(phrase.Substring(4));           // do you do?
25      Console.WriteLine(phrase.Substring(4).ToUpper()); // DO YOU DO?
26
```

```
SubString(startPosition, length)
SubString(startPosition)
```

- ToUpper and ToLower return modified versions of, but do not modify the instance that call them.
- SubString returns a portion of a string.
- Passing [start] and [length] will extract [length] characters from [start].
- Passing [start] only will extract all remaining characters from [start]
- Again, SubString does not modify the instance that calls it.

string.Split

```
14
15      string csvString = "one,two,three,four,five,six";
16
17      string[] tokens = csvString.Split(',');
18
19
20
21
22
23
```

tokens {string[6]}

- [0] "one"
- [1] "two"
- [2] "three"
- [3] "four"
- [4] "five"
- [5] "six"

- Split will “break up” a string based on a delimiter character.
- Pass the delimiter as a char.
- Returns a string array of the elements.
- Does not modify the instance that calls it.

The Rule of Definite Assignment

The C# Compiler checks that all local variables have been initialized before being used.

```
20
21      int count;
22      Console.WriteLine(count);
23
```

Error List

Entire Solution | 1 Error | 0 Warnings | 0 Messages

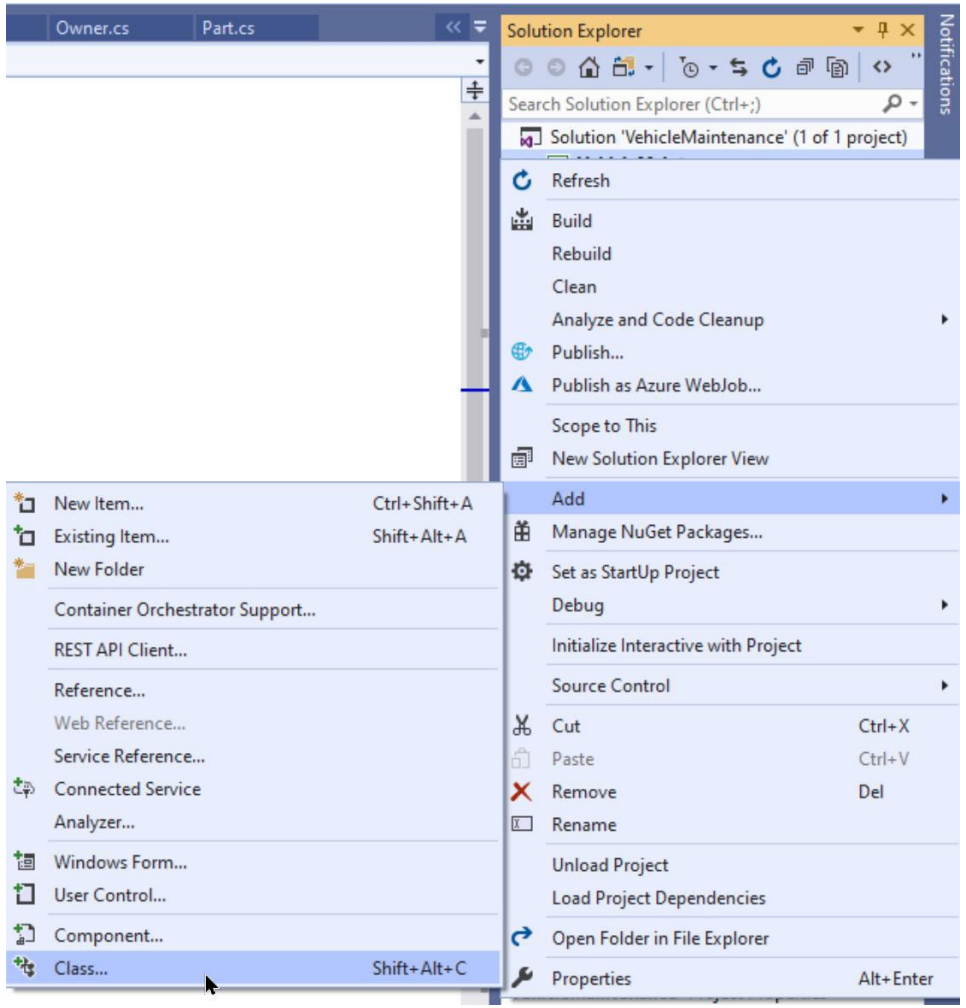
	Code	Description
⊗	CS0165	Use of unassigned local variable 'count'

Working with Classes – Source Files

- In Java, a source (.java) file can contain one class definition.
- File and class names must match.
- In C#, a source (.cs) file may contain multiple class definitions.
- One class may also span multiple source (.cs) files.
  (Partial Classes – more in Week 7)
- File and class names do not have to match.
- **Best practice, however, is to follow the Java convention.**
- By having one Class per source file, you can quickly see a list of your project classes in the Solution Explorer. It also improves code navigation in the code editor.

Adding a Class to a Project

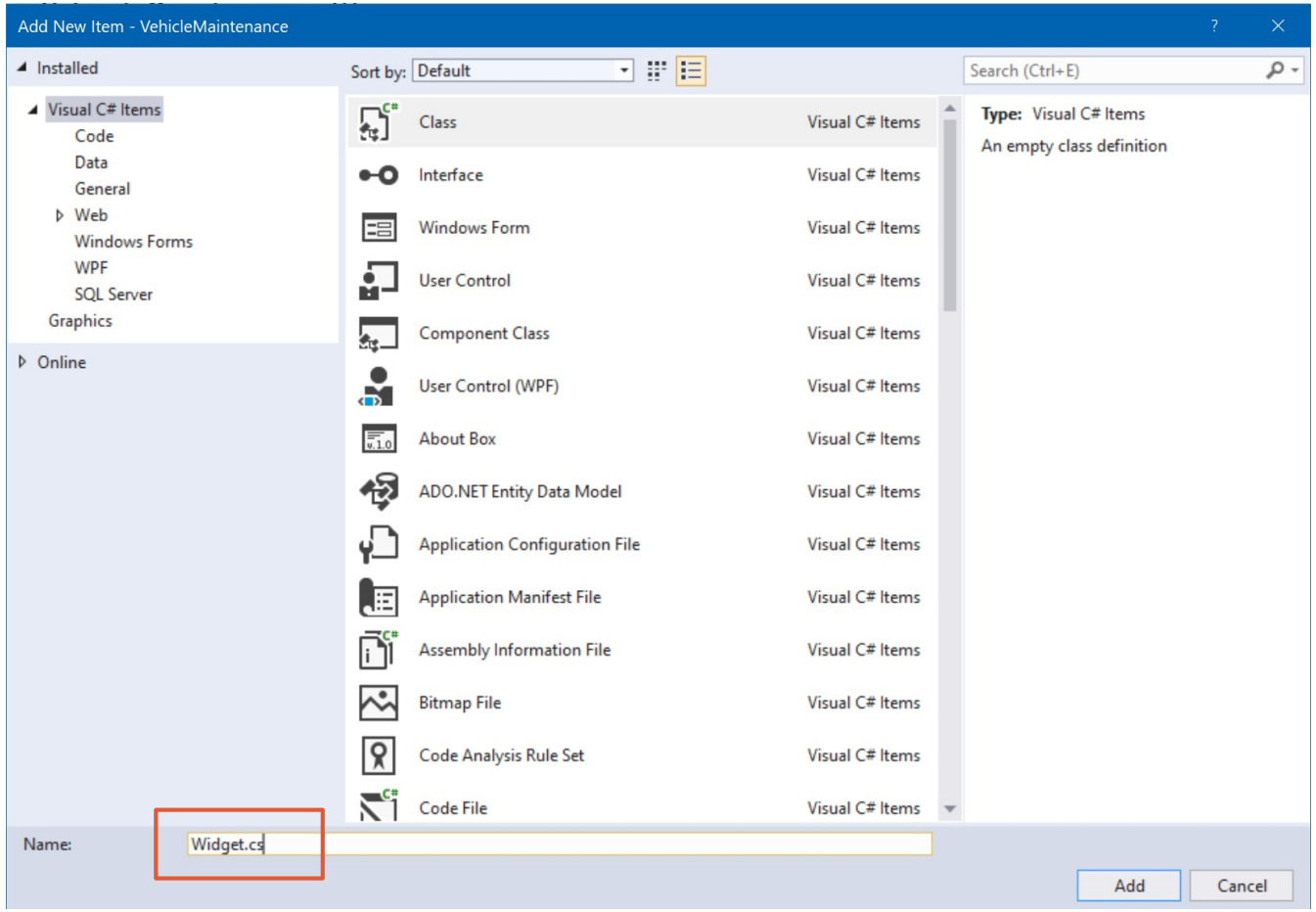
1. Right-click on Project in Solution Explorer
2. Click on Add
3. Click on Class



Adding a Class to a Project

4 Name the new Class (PascalCase)

5 Click Add



Data Class Design

```
2 references
class Contact
{
    private int id;
    private string firstName;
    private string lastName;
    private string address1;
    private string address2;
    private string city;
    private string province;
    private string postalCode;
    private string emailAddress;
    private DateTime dateOfBirth;

    // constructors

    // properties (get/set)

    // other code
}
```

Create a field for each attribute of the entity you are modelling. Choose an appropriate name and data type for each attribute.

Separation of Concerns Principle

```
15 references
class Person
{
    private string firstName;
    private string lastName;

    0 references
    public Person(string firstName, string lastName)
    {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    0 references
    public void PrintFullName()
    {
        Console.WriteLine("{0} {1}", firstNam  lastName);
    }
}
```

Best practices in Object Oriented Design is to design classes with a single purpose.

Classes which represent everyday things are commonly referred to as data classes as they contain the attributes (data) of that particular item.

Data classes should **NOT** perform any UI functions as shown in this sample code. The UI code to output to the Console should be placed in a separate class designed for that purpose.

Utility Class Design

```
10
    1 reference
11  class ConsolePrinter
12  {
        1 reference
13      public static void PrintContacts(Contact[] contacts)
14      {
15          // implementation code
16
17
18
19      }
20  }
21
```

Make utility methods static so they can be invoked
without having to create an instance of the utility class

Make Utility Methods Static

```
3 references
class ConsolePrinter
{   Static method
    1 reference
    public static void PrintContacts(
    {
        // implementation code

    }
    Instance method
    1 reference
    public void PrintContactsInst(Con
    {

        // implementation code
    }
}
```

Static methods can be invoked directly from the class

```
ConsolePrinter.PrintContacts(contacts);
```

A class must b e instantiated prior to invoking an instance method

```
ConsolePrinter printer = new ConsolePrinter();
printer.PrintContactsInst(contacts);
```

Working with Classes – Constructors

```
15 references
class Person
{
    private string firstName;
    private string lastName;

    0 references
    public Person(string firstName, string lastName)
    {
        this.firstName = firstName;
        this.lastName = lastName;
    }
```

- Same name as Class
- No return type (not even void)
- Can have multiple overloads

Invoked at object instantiation.

Used to initialize object state.

A default (no parameters) constructor is automatically created by the compiler.

Once an overloaded constructor has been implemented, the compiler will no longer create a default constructor automatically.

Field Initialization in Default Constructors is Redundant

```
1 reference
public Person()
{
}
```

Fields are auto initialized

String	null
Numeric	0
Bool	false

```
1 reference
public Person()
{
    firstName = "";
    lastName = "";
    email = "";
    age = 0;
}
```

```
1 reference
public Person()
{
    firstName = "Not Set";
    lastName = "Not Set";
    email = "Not Set";
    age = 0;
}
```

The compiler provides a default constructor automatically.

It is usually not necessary to explicitly write a default constructor when no others exist.

When a default constructor is created to accompany a parameterized one, it will usually contain no code. It is not necessary to initialize fields to their default values as this is done automatically.

Property Methods

Conventional Get/Set Methods …

```
12      Backing Field
13      private string description;
14
        0 references
15      public string GetDescription()
16      {
17          return description;
18      }
19
        0 references
20      public void SetDescription(string newDescription)
21      {
22          description = newDescription;
23      }
24
25
26
27
```

… Replaced by a Property Method

```
12      Backing Field
13      private string description;
14
        0 references
15      public string Description
16      {
17          get
18          {
19              return description;
20          }
21
22          set
23          {
24              description = value;
25          }
26      }
27
```

camelCase

PascalCase

Name the property method by taking the field name and capitalizing the first letter

Property methods are preferred over conventional Get/Set methods for C# classes.

Property Method Gotcha

Accessing the Property instead of the field …

```
12      Backing Field
13      private string description;
14
        3 references
15      public string Description
16      {
17          get
18          {
19              return Description;
20          }
21
22          set
23          {
24              Description = value;
25          }
26      }
27
```

Circular Reference

Circular Reference

… Creates a Circular Reference [Boom!!]

```
12      Backing Field
13      private string description;
14
        3 references
15      public string Description
16      {
17          get
18          {
19              re
20          }
21
22          set
23          {
24              De
25          }
26      }
27
```

Exception Unhandled

System.StackOverflowException: 'Exception of type 'System.StackOverflowException' was thrown.'

View Details | Copy Details | Start Live Share session...

Exception Settings

Ensure the Property interacts with the field, and not with itself. Otherwise, a StackOverflowException will be thrown.

Calculated Property Methods

```
4 references
class Rectangle
{    Backing Fields
    private double width;
    private double height;
    // area is NOT stored as a field of the class

    2 references
    public double Width
    {
        get { return width; }
        set { width = value; }
    }

    2 references
    public double Height
    {
        get { return height; }
        set { height = value; }
    }

    // Area is calculated on demand and exposed
    //   as a public, readonly property
    2 references
    public double Area
    {
        get { return width * height; }
    }
}
```

Not all Properties require a corresponding field. Sometimes a value can be calculated from other Property values.

Calculated Properties should only be used when the result can be calculated from other local values.

Use a normal method when calculations require values from outside the class.

Calculated Properties are intrinsically readonly.

Readonly Properties

```
18      // readonly Property method
        1 reference
19      public int Id
20      {
21          get
22          {
23              return id;
24          }
25      }
26
```

Some properties should be readonly. Their value should only be altered internally. In the following example, the Id property is readonly.

Consumers of the class should be able to read the Id but not alter its value.

To make a property readonly, simply omit the set portion.

Readonly Properties vs Readonly Fields

```
 9    class Widget
10    {
11        private readonly int id;
12        private string sku;
13        private decimal price;
14
          1 reference
15        public Widget(int id)
16        {
17            this.id = id;
18        }
19
```

The id field is marked readonly which means is can only be assigned to in a constructor

```
21        private void updateId()
22        {
23            id = 100;
24        }
25
26
          0 references
27        public int Id
28        {
```

(field) int Widget.id

A readonly field cannot be assigned to (except in a constructor or a variable init

An error occurs when attempting to assign a value to the readonly field outside of a constructor

Readonly Properties vs Readonly Fields

```
35          public string Sku
36          {
37              get { return sku; }
38           //   set { sku = value; }
39          }
40
```

A normal field can be exposed as a readonly property by omitting the set portion.

```
            0 references
25          private void updateSku()
26          {
27              sku = "ABC100";
28          }
29
```

The value can be altered from anywhere inside the class.

Automatic Property Methods

When there are no additional code requirements …

Backing Field

```
private string description;

0 references
public string Description
{
    get
    {
        return description;
    }

    set
    {
        description = value;
    }
}
```

… replace with an Automatic Property

Backing Field

```
// private string description;

0 references
public string Description { get; set; }
```

This one line of code replaces ALL code to the left, including the backing field

Init Only Setters (C# 9.0)

```
1 reference
class Widget
{
    2 references
    public int Id { get; init; }

    0 references
    public Widget()
    {
        this.Id = 99;
    }

    0 references
    public void SetId(int id)
    {
        this.Id = id; //Compiler Error
    }
}
```

Properties with init only setters can only be set in the constructor (or during object initialization).

Object Initializers

```
2 references
class Person
{
    private string firstName;
    private string lastName;

    2 references
    public string FirstName
    {
        get { return firstName; }
        set { firstName = value; }
    }

    2 references
    public string LastName
    {
        get { return lastName; }
        set { lastName = value; }
    }
}
```

- Invokes the default constructor.
- Each field must be exposed as a writable public Property.
- Arguments are specified as name/value pairs.
- Intellisense aware.
- Can initialize any number of fields in any order.
- Code is more readable than a parameterized ctor.

```
Person person =
    new Person { FirstName = "Lewis", LastName = "Black" };

Console.WriteLine("{0} {1}", person.FirstName, person.LastName);
```

Arguments are specified as name/value pairs

ToString() Method

```
0 references
public override string ToString()
{
    return string.Format("{0} {1}", firstName, lastName);
}
```

- Defined in class Object
- Can (and should) be overridden in a derived class
- Should not contain Console output code (SoC Principle)
- Returned string should be **general** in content and format,

not "**Application Specific**"

  - So, you should be able to call .ToString() from ***ANY*** application and get a meaningful result.

- Implicitly invoked in some cases (Console.WriteLine)

Namespaces

```
1   namespace HelloWorld
2   {
        0 references
3       class Program
4       {
            0 references
5           static void Main(string[] args)
6           {
7               System.Console.WriteLine("Hello World");
8           }
9       }
10  }
```

Adding a using statement removes the need to include the namespace repeatedly in your code.

```
1   using System;
2
3   namespace HelloWorld
4   {
        0 references
5       class Program
6       {
            0 references
7           static void Main(string[] args)
8           {
9               Console.WriteLine("Hello World");
10          }
11      }
12  }
```

Main purpose of namespaces:

- Organize and categorize classes
- Prevent naming collisions

Each project “lives” in its own namespace

- .NET Base Class Library
- More than 2000 classes
- Roughly 100 namespaces

Namespaces

```
1   using System;
2
3   namespace CompanyA
4   {
        0 references
5       class Invoice
6       {
7           // class implementation
8       }
9   }
10
```

```
1   using System;
2
3   namespace CompanyB
4   {
        0 references
5       class Invoice
6       {
7           // class implementation
8       }
9   }
10
```

Two 3rd party library publishers offer an accounting library with includes an Invoice class:

How can one be distinguished from the other?

The Fully Qualified name of a class consists of the namespace and the class name in the form:

Namespace.Classname

Therefore, the Fully Qualified names are unique:

CompanyA.Invoice

CompanyB.Invoice

Arrays of Value Types

```
14
15      int[] numbers;          // declaration
16      numbers = new int[3];   // instantiation
17
18      numbers[0] = 329;       //
19      numbers[1] = 1956;      // initialization
20      numbers[2] = -392;      //
21
22      // combined declaration and instantiation
23      int[] someNumbersA = new int[3];
24
25      someNumbersA[0] = 100;  //
26      someNumbersA[1] = 300;  // initialization
27      someNumbersA[2] = 500;  //
28
29      // single statement declaration, instantiation
30      //  and initialization
31      int[] someNumersB = new int[3] { 100, 300, 500 };
32
33      // single statement declaration, instantiation
34      //  and initialization (more concise syntax)
35      int[] someNumbersC = { 100, 300, 500 };
36
```

An array is a series of elements of the same type placed in contiguous memory locations that can be individually accessed by adding an index to a unique identifier.

In an array of value types, the values are stores directly in the array elements.

Array of ints:

0	1	2
100	300	500

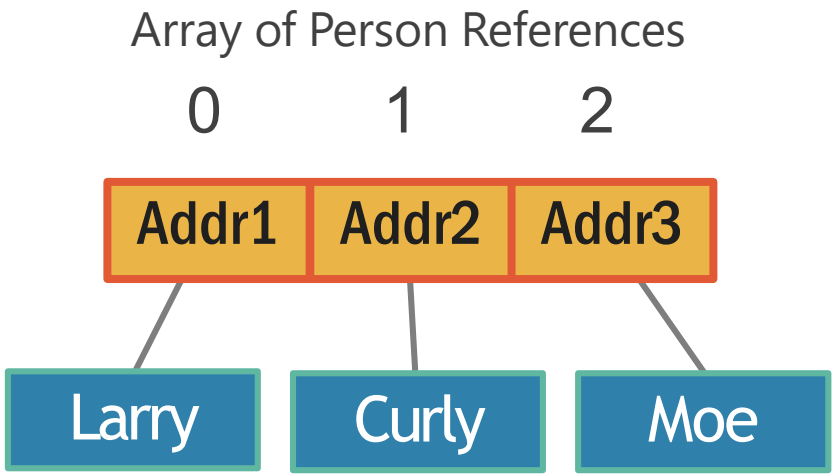
Arrays of Reference Types

```
14
15      // combined declaration and instantiation
16      Person[] contacts = new Person[3];
17
18      // initialization
19      contacts[0] = new Person("Larry", "Fine", 48);
20      contacts[1] = new Person("Curly", "Howard", 47);
21      contacts[2] = new Person("Moe", "Howard", 53);
22
23      // single statement declaration, instantiation
24      //  and initialization
25      Person[] people =
26      {
27            new Person("Larry", "Fine", 48)
28          , new Person("Curly", "Howard", 47)
29          , new Person("Moe", "Howard", 53)
30      };
31
```

In an array of reference types, the array elements do not contain the object themselves.

The objects are instantiated and the references to those objects are stored in the array.

An array of Person objects is actually an array of Person object references



Inline Comments

Rule of thumb:

Comment what the code ***should*** be doing

Not what it ***is*** doing (unless it is not intuitive/complex)

Inline Comments

Example of what not to do:

```
0 references
public int GetTaxableCount(InvoiceList invoices)
{
    int count = 0; //declare variable and assign 0
    foreach (Invoice invoice in invoices) //loop through invoices
    {
        if( invoice.Taxable ) //check if it is taxable
        {
            count++; //increment count
        }
    }
    return count; //return the count
}
```

Everything in the comments is completely obvious by reading the code.

The comments just add clutter and make the code less readable

Inline Comments

Example of good commenting:

```
public int GetTaxableCount(InvoiceList invoices)
{

    int count = 0;
    foreach (Invoice invoice in invoices)
    {
        //Check to see if invoice has taxable set to true - this represents PST taxable
        if( invoice.Taxable )
        {
            count++;
        }
    }
    return count;
}
```

This comment tells us something that we can't know from the code alone – that the "taxable" in this case refers to PST