

# C# Fundamentals

An Introduction C# and .NET

Van Pham

Linkedin: <https://www.linkedin.com/in/phamhongvan/>

---

# .NET

.NET is a software framework

Your Application

Common Language Runtime  
(CLR)



Framework Class Library  
(FCL)

# CLR

## The CLR manages your application

- Memory management
- Operating system and hardware independence
- Language independence

Your Application

Common Language Runtime  
(CLR)



Framework Class Library  
(FCL)

# FCL

## Framework class library

- A library of functionality to build applications

Your Application

Common Language Runtime (CLR)



Framework Class Library (FCL)

# C#

- .. One of many languages for .NET
- .. Syntax is similar to Java, C++, and JavaScript

```
public static void Main()
{
    if (DateTime.Now.DayOfWeek == DayOfWeek.Monday)
    {
        Console.WriteLine("Another case of the Mondays!");
    }
}
```

# csc.exe

## .. The C# command line compiler

- Transforms C# code into Microsoft Intermediate Language



```
C:\>hello Scott Joy Sara
```



args[0] is "Scott"

args[1] is "Joy"

args[2] is "Sara"

# Visual Studio

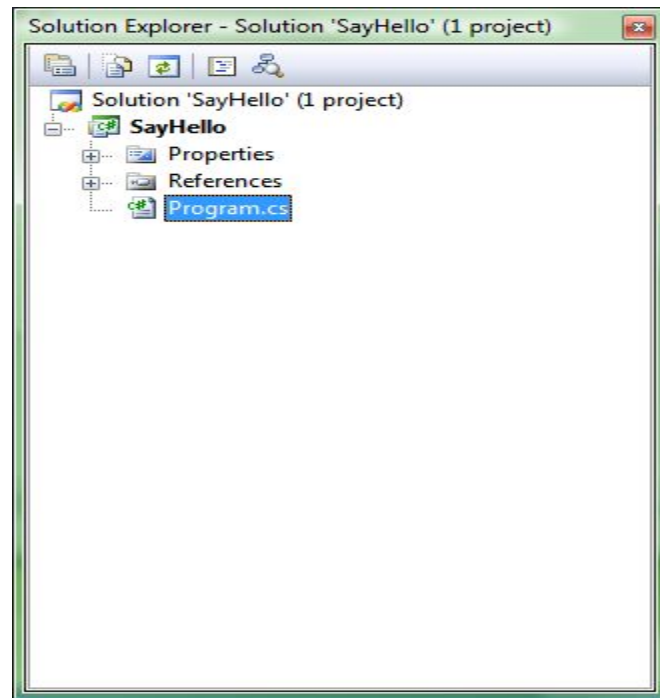
## -- An integrated development environment

- Edit C# (and other) files
- Runs the C# compiler
- Debugging
- Testing



# Solution Explorer

- .. **Will contain at least one project**
  - Contains one or more source code files
  - Each project produces an assembly
- .. **Projects organized under a solution**
  - Manage multiple applications or libraries



# Types

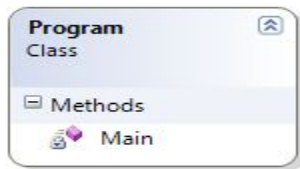
## .. C# is strongly typed

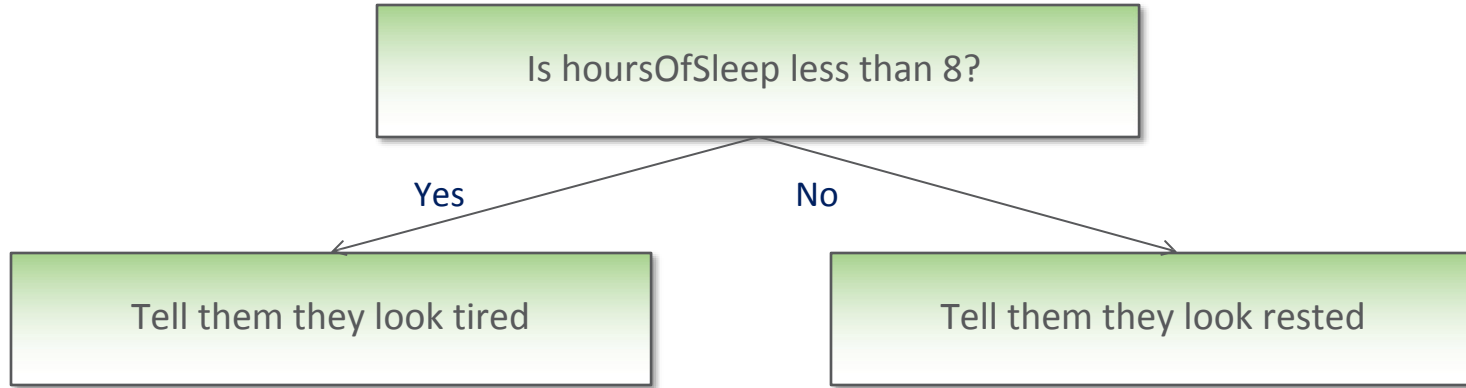
- One way to define a type is to write a class
- Every object you work with has a specific type
- 1,000s of types are built into the .NET framework
- You can define your own custom types

## .. Code you want to execute must live inside a type

- You can place the code inside a method
- We'll explore other things you can add to a type later

...





# Summary

- .. C# is a strongly typed & case sensitive language for .NET
- .. Visual Studio is an IDE to work with C# applications of all types

```
static void Main(string[] args)
{
    Console.WriteLine("Your name:");
    string name = Console.ReadLine();

    Console.WriteLine("How many hours of sleep did you get last night?");
    int hoursOfSleep = int.Parse(Console.ReadLine());

    Console.WriteLine("Hello, " + name);
    if(hoursOfSleep > 8)
    {
        Console.WriteLine("You are well rested");
    }
    else
    {
        Console.WriteLine("You need more sleep");
    }
}
```

# Classes and Objects

---

We need an electronic grade book to read the scores of an individual student and then compute some simple statistics from the scores.

The grades are entered as floating point numbers from 0 to 100, and the statistics should show us the highest grade, the lowest grade, and the average grade.

# Constructors

- .. Special methods used to initialize objects

```
GradeBook book = new GradeBook();
```



```
public GradeBook()  
{  
    // ... initialization code  
}
```

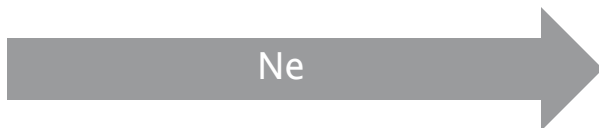
# Classes Versus Variables

- .. A class is a blueprint for creating objects
- .. A class can also be used to type a variable
  - A variable can refer to any object of the same type

```
class GradeBook
{
    public GradeBook()
    {
        grades = new List<float>();
    }

    public void AddGrade(float grade)
    {
        grades.Add(grade);
    }

    List<float> grades;
}
```



GradeBook  
Object



# Reference Types

- .. Classes are reference types
- .. Variables hold a pointer value

```
GradeBook book1 = new GradeBook();
```

```
GradeBook book2 = book1;
```

GradeBook  
Object

# Statics

- .. Use static members of a class without creating an instance

```
public static float MinimumGrade = 0; public  
static float MaximumGrade = 100;
```

```
Console.WriteLine("Hello!");  
Console.WriteLine(GradeBook.MaximumGrade);
```

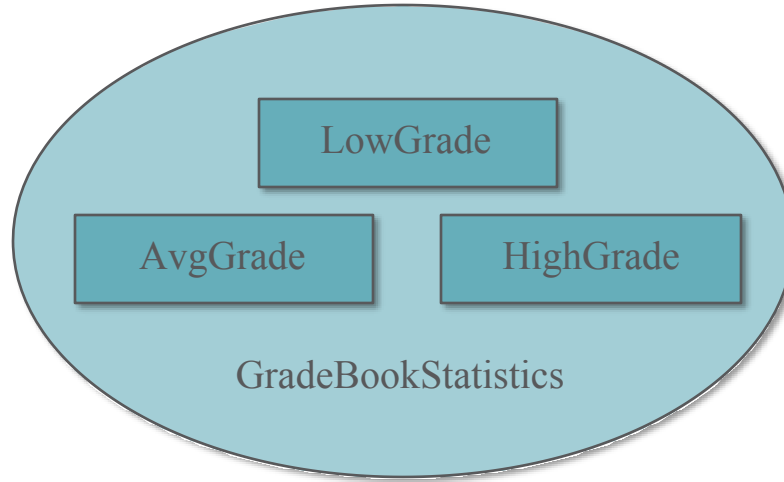
# Classes

- .. **A class definition creates a new type**
  - Use the type for variables and arguments
- .. **Use a class to create objects**
  - Invoke methods and save state in objects

# Object Oriented Programming

- .. **Objects are nouns**
- .. **Methods are verbs**
- .. **Objects encapsulate functionality**

# Encapsulation



# Access Modifiers

```
class GradeBook
{
    public GradeBook()
    {
        grades = new List<float>();
    }

    public void AddGrade(float grade)
    {
        grades.Add(grade);
    }

    List<float> grades;
}
```

publi

Constructor

AddGrade

privat

e

grades

# Summary

```
class GradeBook
{
    public GradeBook()
    {
        grades = new List<float>();
    }

    public GradeStatistics ComputeStatistics()
    {
        GradeStatistics stats = new GradeStatistics();

        float sum = 0;
        foreach(float grade in grades)
        {
            stats.HighestGrade = Math.Max(grade, stats.HighestGrade);
            stats.LowestGrade = Math.Min(grade, stats.LowestGrade);
            sum += grade;
        }
        stats.AverageGrade = sum / grades.Count;
        return stats;
    }

    public void AddGrade(float grade)
    {
        grades.Add(grade);
    }

    private List<float> grades;
}
```

# Assemblies





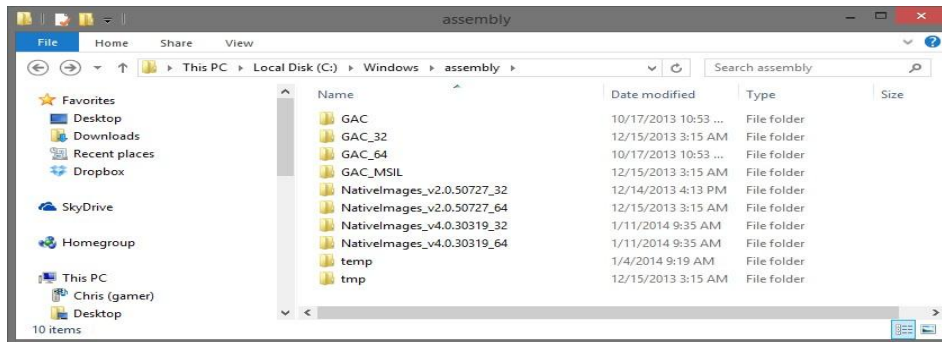
# csc.exe

## The C# Command Line Compiler



# Assemblies

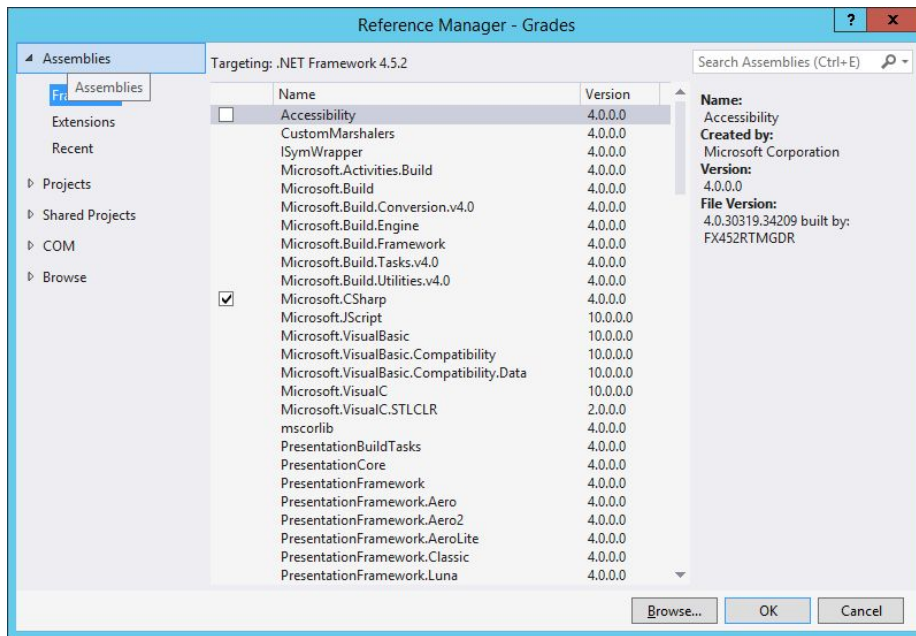
- .. **Assemblies are .exe or .dll files**
  - Contain metadata about all types inside
- .. **Global Assembly Cache**
  - A central location to store assemblies for a machine



# References

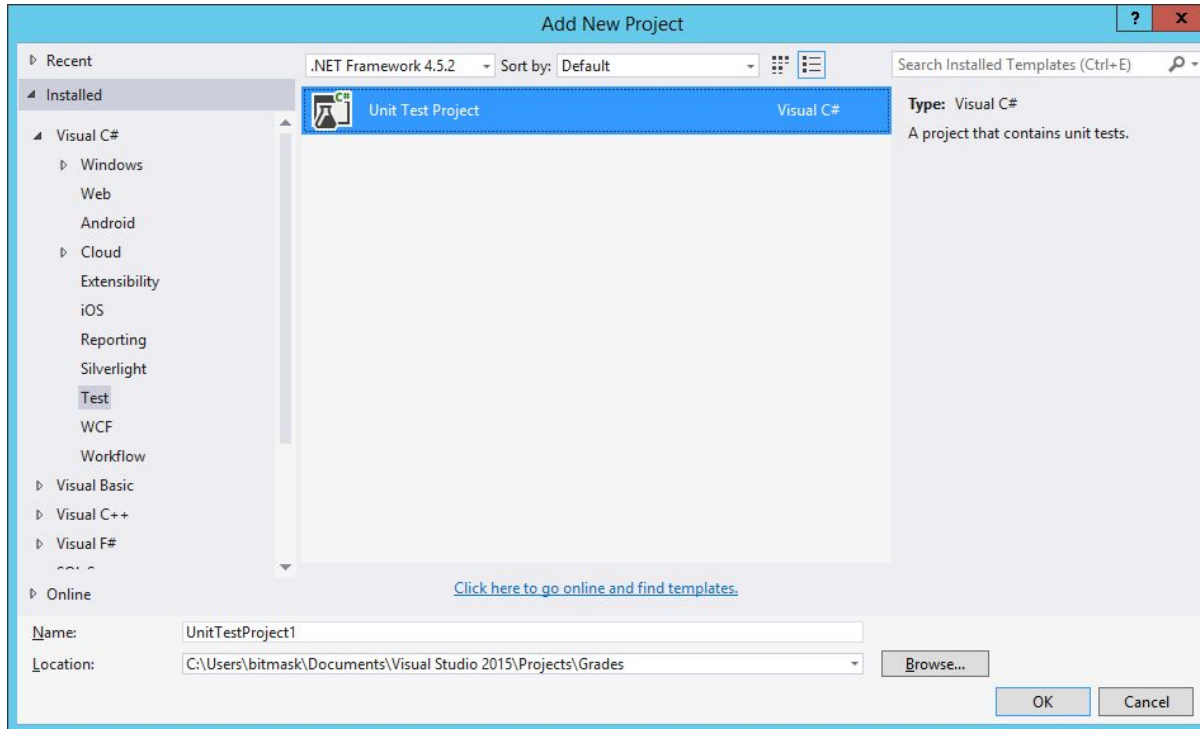
.. Must load assembly into memory before using types inside

- Easy approach – reference the assembly in Visual Studio



# Unit Testing

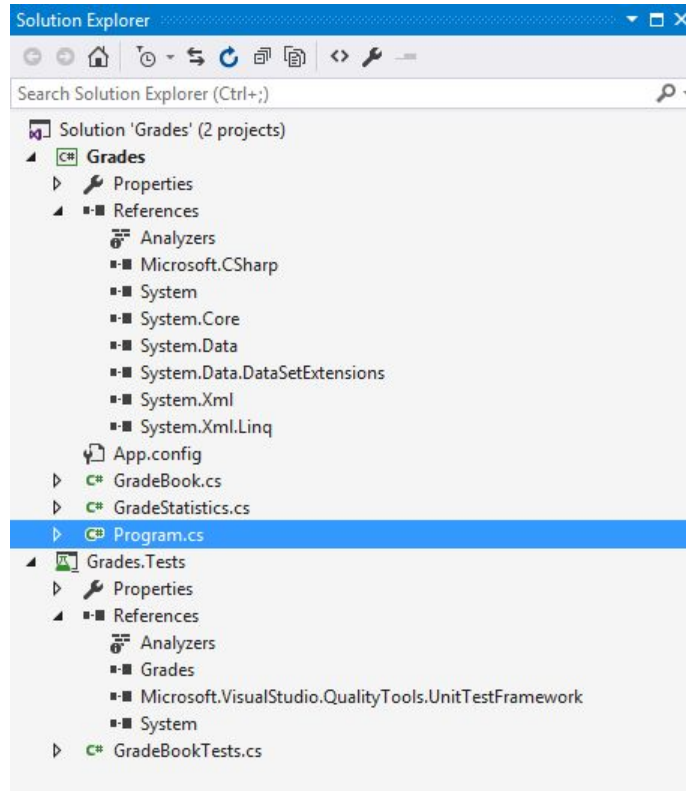
- .. Write code to test code!



# Access Modifiers

Keyword	Visibility
public	Everywhere
private	Only in the same class
internal	Only in the same assembly

# Summary



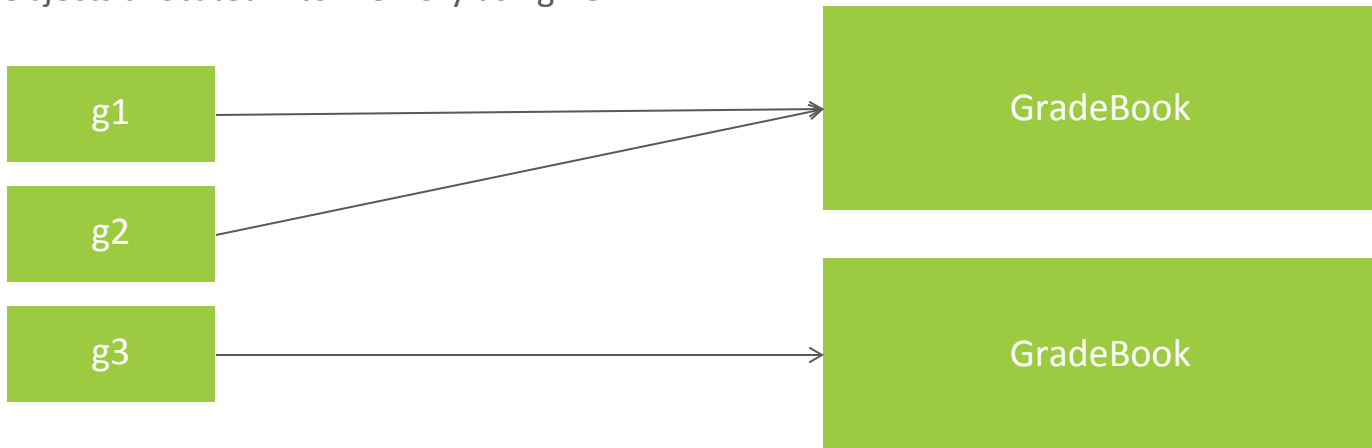
# Types



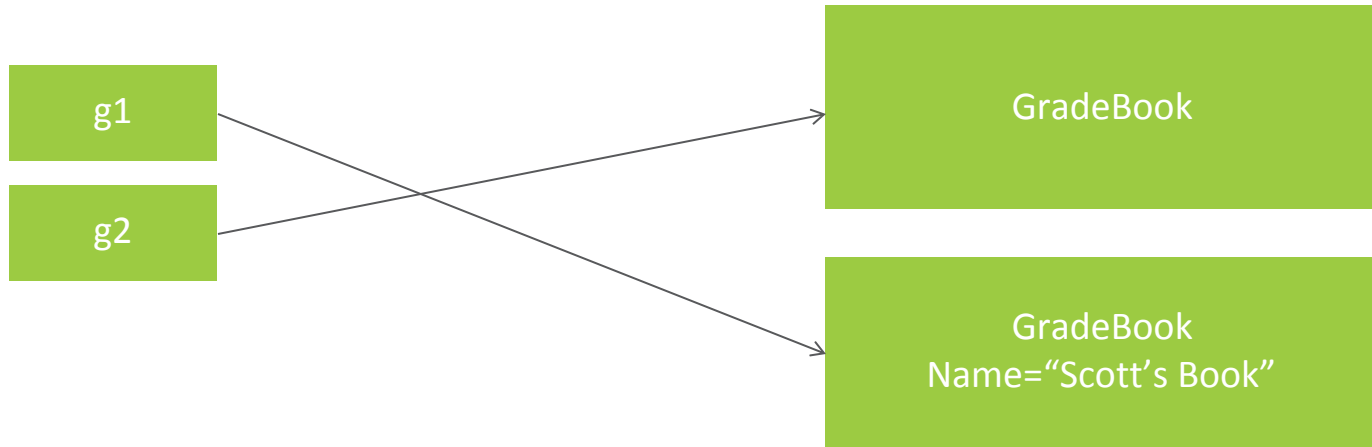
# Reference Types

## .. Variables store a reference to an object

- Multiple variables can point to the same object
- Single variable can point to multiple objects over it's lifetime
- Objects allocated into memory using new







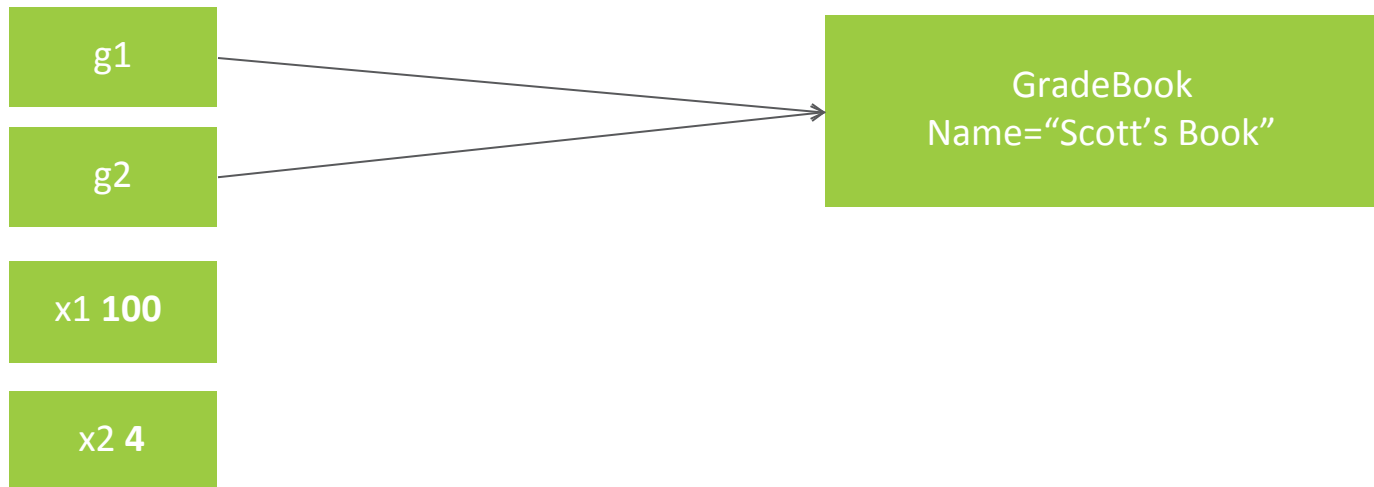
# Value Types

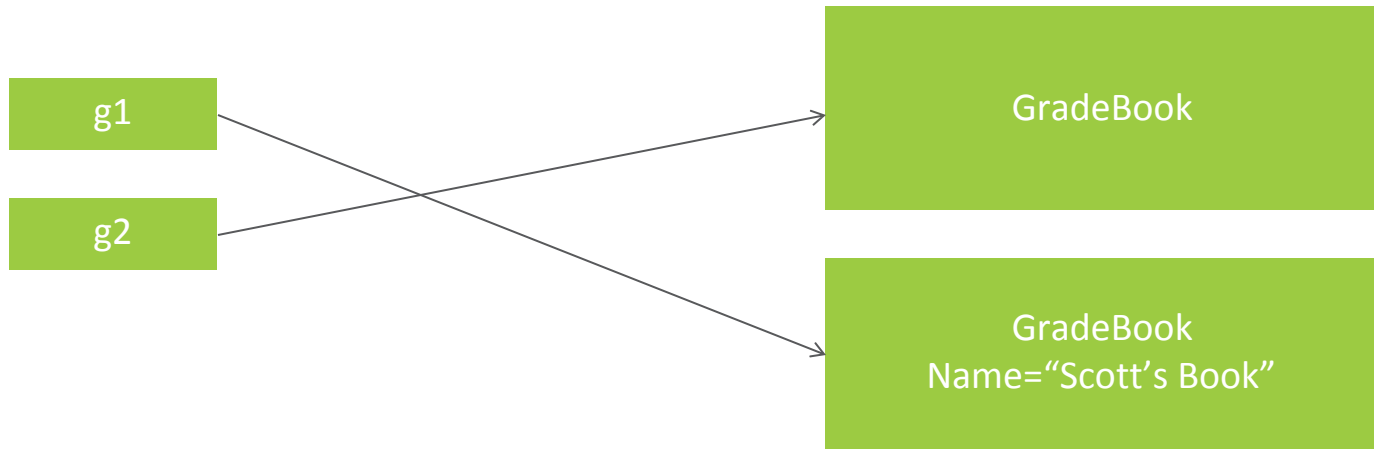
- Variables hold the value

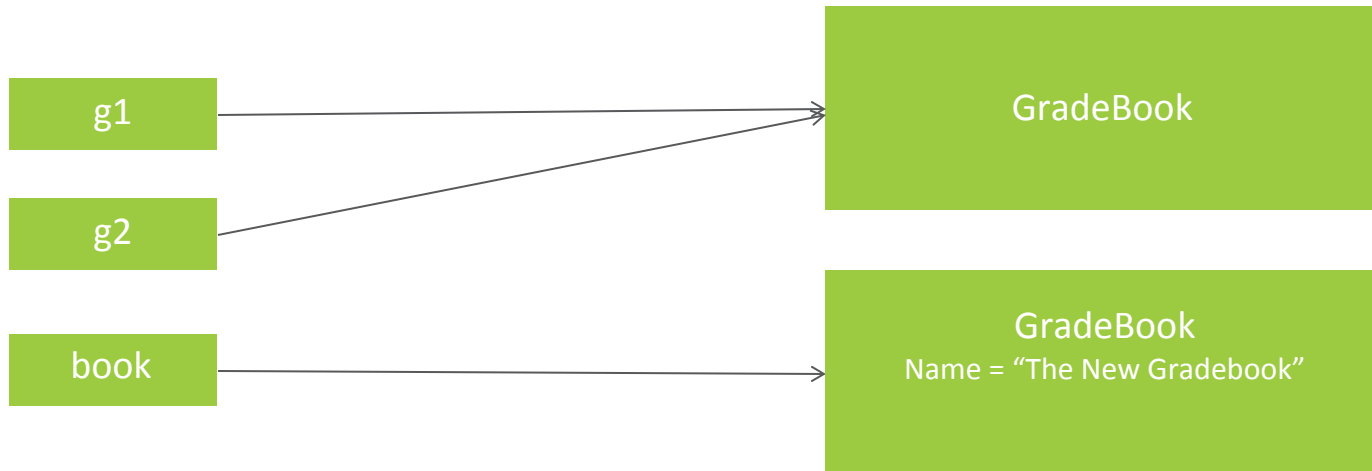
- No pointers, no references

- Many built-in primitives are value types

- int, double, float







# Creating Value Types

## .. struct definitions create value types

- Should represent a single value
- Should be small

```
public struct DateTime
{
    // ...
}
```

# Enumerations

## .. An enum creates a value type

- A set of named constants
- Underlying data type is int by default

```
public enum PayrollType
{
    Contractor = 1, Salaried,
    Executive,

    Hourly
}
```

```
if(employee.Role == PayrollType.Hourly)
{
    // ...
}
```

# Method Parameters

## -- Parameters pass “by value”

- Reference types pass a copy of the reference
- Value types pass a copy of the value

```
public void DoWork(GradeBook book)
{
    book.Name = "Grades";
}
```

# Immutability

## .. Value types are usually immutable

- Can not change the value of 4
- Can not change the value of August 9<sup>th</sup>, 2002

```
DateTime date = new DateTime(2002, 8, 11);  
date.AddDays(1)  
  
string name = " Scott "; name.Trim();
```



# Arrays

## .. Manage a collection of variables

- Fixed size
- Always 0 indexed

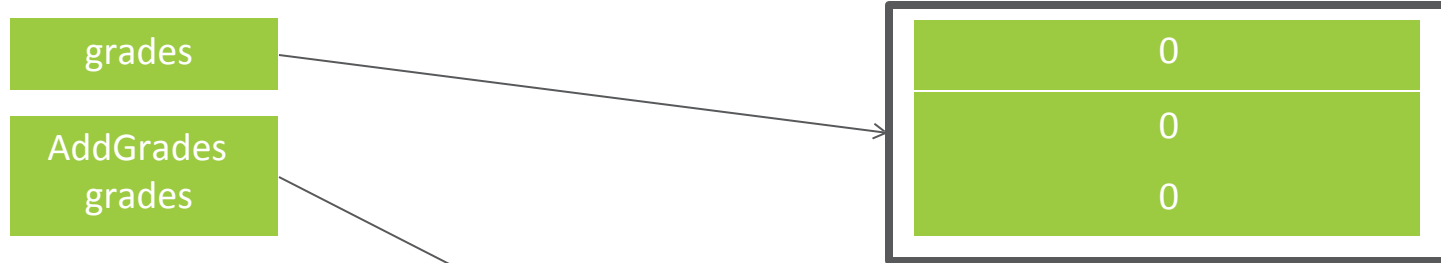
```
const int numberOfStudents = 4;
int[] scores = new int[numberOfStudents];

int totalScore = 0; foreach(int score in scores)
{
    totalScore += score;
}

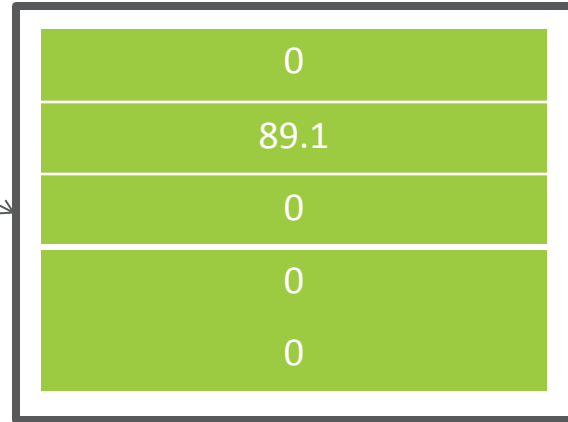
double averageScore = (double)totalScore / scores.Length;
```

grades

AddGrades  
grades



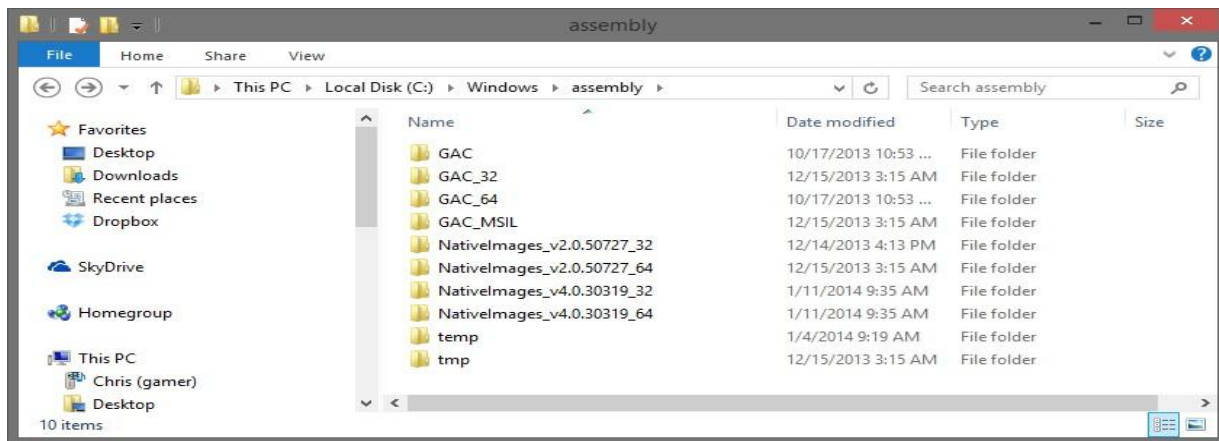
0
0
0



0
89.1
0
0

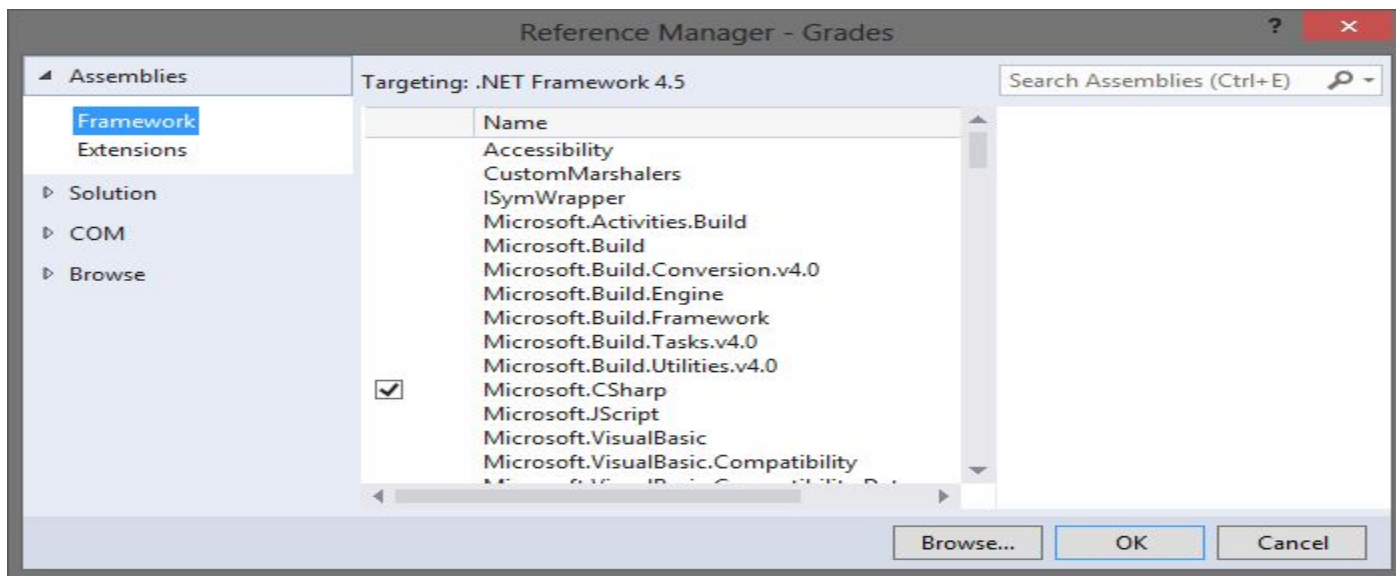
# Assemblies

- .. **Assemblies are .exe or .dll files**
  - Contain metadata about all types inside
- .. **Global Assembly Cache**
  - A central location to store assemblies for a machine



# References

- .. **Must load assembly into memory before using types inside**
  - Easy approach – reference the assembly in Visual Studio



# Summary

- .. **Every type is a value type or reference type**

- Use struct to create a value type
- Use class to create a reference type

- .. **Arrays and strings are reference types**

- Strings behave like a value type

# Methods, Fields, Events, and Properties



# Methods

- .. **Methods define behavior**
- .. **Every method has a return type**
  - `void` if no value returned
- .. **Every method has zero or more parameters**
  - Use `params` keyword to accept a variable number of parameters
- .. **Every method has a signature**
  - Name of method + parameters

```
public void WriteAsBytes(int value)
{
    byte[] bytes = BitConverter.GetBytes(value);

    foreach(byte b in bytes)
    {
        Console.Write("0x{0:X2} ", b);
    }
}
```

# Method Overloading

- .. Define multiple methods with the same name in a single class

- Methods require a unique signature

- .. Compiler finds and invokes the best match

```
public void WriteAsBytes(int value)
{
    // ...
}

public void WriteAsBytes(double value)
{
    // ...
}
```



# Methods - Review

## .. **Instance methods versus static methods**

- Instance methods invoked via object, static methods via type

## .. **Abstract methods**

- Provide no implementation, implicitly virtual

## .. **Virtual methods**

- Can override in a derived class

## .. **Partial methods**

- Part of a partial class

## .. **Extension methods**

- Described in the LINQ module

# Fields

## .. Fields are variables of a class

- Can be read-only

```
public class Animal
{
    private readonly string _name;

    public Animal(string name)
    {
        _name = name;
    }
}
```

# Properties

- .. **A property can define a get and/or set accessor**
  - Often used to expose and control fields
- .. **Auto-implemented properties use a hidden field**

```
public string Name
{
    get; set;
}
```

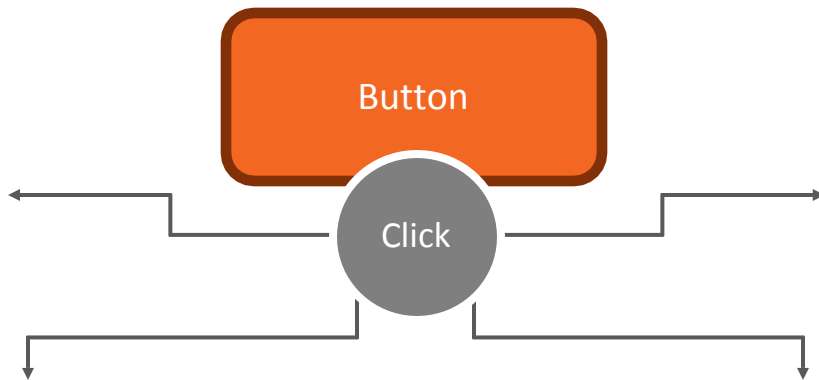
```
private string _name;

public string Name
{
    get { return _name; } set
    {
        if(!String.IsNullOrEmpty(value))
        {
            _name = value;
        }
    }
}
```

# Events

## .. Allows a class to send notifications to other classes or objects

- Publisher raises the event
- One or more subscribers process the event



# Delegates

- .. I need a variable that references a method
- .. A delegate is a type that references methods

```
public delegate void Writer(string message);
```

```
Logger logger = new Logger();  
Writer writer = new Writer(logger.WriteMessage);  
writer("Success!!");
```

```
{  
    public void WriteMessage(String message)  
    {  
        Console.WriteLine(message);  
    }  
}
```

# Subscribing To Events

## .. Use the += and -= to attach and detach event handlers

- Can attached named or anonymous methods

```
public static void Initialize()
{
    _submitButton.Click += new RoutedEventHandler(_submitButton_Click);
}

static void _submitButton_Click(object sender, RoutedEventArgs e)
{
    // ... respond to event
}
```

# Publishing Events

- .. Create custom event arguments (or use a built-in type)
  - Always derive from the base EventArgs class
- .. Define a delegate (or use a built-in delegate)
- .. Define an event in your class
- .. Raise the event at the appropriate time

```
public event NameChangingEventHandler NameChanging;

private bool OnNameChanging(string oldName, string newName)
{
    if(NameChanging != null)
    {
        NameChangingEventArgs args = new NameChangingEventArgs(); args.Cancel =
        false;
        args.NewName = newName; args.OldName = oldName;
        NameChanging(this, args);
    }
}
```

# Summary

## .. **Members are used to craft an abstraction**

- Fields and properties for state
- Methods for behavior
- Events for notification



# Control Flow



# Overview

- Branching
- Iterating
- Jumping
- Exceptions



# Branching

```
if (age <= 2)
    ServeMilk(); else if
(age < 21)
    ServeSoda();
else
{
    ServeDrink();
}
```

```
if (age <= 2)
{
    if(name == "Scott")
    {
        // ...
    }
}
```

```
string pass = age > 20 ? "pass" : "nopass";
```

# Switching

## .. Restricted to integers, characters, strings, and enums

- Case labels are constants

- Default label is optional

```
switch(name) {  case "Scott":  
                ServeSoda(); break;  
  case "Alex":  
                ServeMilk();  
                ServeDrink(); break;  
  default:  
                ServeMilk(); break;  
}
```

# Iterating

```
for(int i = 0; i < age; i++)  
{  
    Console.WriteLine(i);  
}
```

```
while(age > 0)  
{  
    age ---=  
    Console.WriteLine(age);  
}
```

```
do  
{  
    age++;  
    Console.WriteLine(age);  
} while (age < 100);
```

```
= {2, 21, 40, 72, 100};  
int value in ages)  
e.WriteLine(value);  
}
```

# Iterating with foreach

## .. Iterates a collection of items

- Uses the collection's GetEnumerator method

```
int[] ages = {2, 21, 40, 72, 100};  
foreach (int value in ages)  
{  
    Console.WriteLine(value);  
}
```

```
int[] ages = {2, 21, 40, 72, 100};  
IEnumerator enumerator = ages.GetEnumerator();  
while(enumerator.MoveNext())  
{  
    Console.WriteLine((int)enumerator.Current);  
}
```

# Jumping

- .. break
- .. continue
- .. goto
- .. return
- .. throw

```
foreach(int age in ages) { if(age
    == 2) {
        continue;
    }
    if(age == 21) { break;
}
}
```

```
foreach(int age in
ages) if(age == 2)
{
    goto skip;
}
// ... skip:
Console.WriteLine("Hello!");
}
```

# Returning

- .. You can use return in a void method

```
void CheckAges()  
{  
    foreach (int age in ComputeAges())  
    {  
        if (age == 21) return;  
    }  
}
```



# Throwing

- .. **Use throw to raise an exception**

- Exceptions provide type safe and structured error handling in .NET

- .. **Runtime unwinds the stack until it finds a handler**

- Unhandled exception will terminate an application

```
if(age == 21)
{
    throw new ArgumentException("21 is not a legal value");
}
```

# Built-in Exceptions

## .. Dozens of exceptions already defined in the BCL

- All derive from `System.Exception`

Type	Description
<code>System.DivideByZeroException</code>	Attempt to divide an integral value by zero occurs.
<code>System.IndexOutOfRangeException</code>	Attempt to index an array via an index that is outside the bounds of the array.
<code>System.InvalidCastException</code>	Thrown when an explicit conversion from a base type or interface to a derived type fails at run time.
<code>System.NullReferenceException</code>	Thrown when a null reference is used in a way that causes the referenced object to be required.
<code>System.StackOverflowException</code>	Thrown when the execution stack is exhausted by having too many pending method calls.
<code>System.TypeInitializationException</code>	Thrown when a static constructor throws an exception, and no catch clauses exist to catch it.

# Handling Exceptions

## .. Handle exceptions using a try block

- Runtime will search for the closest matching catch statement

```
try
{
    ComputeStatistics();
}
catch(DivideByZeroException ex)
{
    Console.WriteLine(ex.Message);
    Console.WriteLine(ex.StackTrace);
}
```

# Chaining Catch Blocks

- .. Place most specific type in the first catch clause
- .. Catching a `System.Exception` catches everything
  - ... except for a few “special” exceptions

```
try {  
    // ...  
}  
catch(DivideByZeroException ex)  
{  
    // ...  
}  
catch(Exception ex)  
{  
    // ...  
}
```

# Finally

## .. Finally clause adds finalization code

- Executes even when control jumps out of scope

```
FileStream file = new FileStream("file.txt", FileMode.Open);  
try  
{  
}  
finally  
{  
    file.Close();  
}
```

```
using(FileStream file1 = new FileStream("in.txt", FileMode.Open))  
using(FileStream file2 = new FileStream("out.txt", FileMode.Create))  
{  
    // ...  
}
```

# Re-throwing Exceptions

- .. **For logging scenarios**

- Catch and re-throw the original exception

- .. **For the security sensitive**

- Hide the original exception and throw a new, general error

- .. **For business logic**

- Useful to wrap the original exception in a meaningful

```
try exception
{
    // ...
}
catch(Exception ex)
{
    // log the error ...
    throw;
}
```

```
try
{
    // ...
}
catch(DivideByZeroException ex)
{
    throw new
        InvalidAccountValueException("...", ex);
}
```

# Custom Exceptions

- .. Derive from a common base exception
- .. Use an Exception suffix on the class name

```
public class InvalidAccountException : Exception
{
    public InvalidAccountException() { }
    public InvalidAccountException(string message) : base(message) { }
    public InvalidAccountException(string
message, Exception inner)
        : base(message, inner) { }
}
```

# Letter Grades

Letter Grades	
90-100	A
80-89	B
70-79	C
60-69	D
0-59	F



```
public interface IWindow
{
    string Title { get; set; }
    void Draw();
    void Open();
}
```

# Summary

- **Flow control statements fall into three categories**
  - Branching
  - Looping
  - Jumping
- **Exceptions provide structured error handling**
  - Throw exceptions (built-in or custom)
  - Catch exceptions

# Object Oriented Programming

---

# Pillars of OOP

Encapsulation

Inheritance

Polymorphism

Abstraction

# Encapsulation

```
public class GradeBook
{
    public GradeBook()...

    public GradeStatistics ComputeStatistics()...

    public void WriteGrades(TextWriter destination)...)

    public void AddGrade(float grade)...

    public string Name...

    public event NameChangedDelegate NameChanged;

    private string _name;
    private List<float> grades;
}
```

```
public void WriteGrades(TextWriter destination)
{
    for (int i = grades.Count; i > 0; i--)
    {
        destination.WriteLine(grades[i-1]);
    }
}
```

# Inheritance

```
public class NameChangedEventArgs : EventArgs
{
    public string ExistingName { get; set; }
    public string NewName { get; set; }
}
```

```
public class A
{
    public void DoWork()
    {
        // ... work!
    }
}

public class B : A
{
}

public class C : B
{
}
```

# Polymorphism

## .. Polymorphism == “many shapes”

- One variable can point to different types of objects
- Objects can behave differently depending on their type

```
public class A : Object
{
    public virtual void DoWork()
    {
        // ...
    }
}

public class B : A
{
    public override void DoWork()
    {
        // optionally call into base...
        base.DoWork();
    }
}
```

# Abstract Classes

- .. **Abstract classes cannot be instantiated**

- Can contain abstract members

```
public abstract class Window
{
    public virtual string Title { get; set; }

    public virtual void Draw()
    {
        // ... drawing code
    }

    public abstract void Open();
}
```



# Interfaces

- .. **Interfaces contain no implementation details**
  - Defines only the signatures of methods, events, and properties
- .. **A type can implement multiple interfaces**

```
public interface Window
{
    string Title { get; set; }
    void Draw();
    void Open();
}
```

# Important Interfaces

Name	Description
IDisposable	Release resources (files, connections)
IEnumerable	Supports iteration (foreach)
INotifyPropertyChanged	Raises events when properties change
IComparable	Compares for sorting

# Summary

```
internal interface IGradeTracker : IEnumerable
{
    void AddGrade(float grade);
    GradeStatistics ComputeStatistics();
    void WriteGrades(TextWriter destination);
    string Name { get; set; }
}
```