IPCA
INSTITUTO POLITÉCNICO
DO CÁVADO E DO AVE

WORKSHOP C# AVANÇADO

3 de Abril
10 de Abril

Alberto Simões
Daniela da Cruz

Day 1

Contents

Day 1

Speaker
Alberto Simões

- Nulls and Nullable Types
- Variadic and Optional Parameters
- Operator Overloading
- Extension Methods
- Generic Types
- Delegates (Actions e Funcs)
- Lambda Expressions
- LINQ

Nulls and Nullable Types

Day 1

Nullable Types (1)

Suppose the following method:

```
int CountValues(int[] haystack, int needle) {
    int c = 0;
    for (int i = 0; i < haystack.Length; i++)
        if (haystack[i] == needle) c++;
    return c;
}
```

How to differentiate from no occurrences, or **null** haystack?

Nullable Types (2)

Solution 1: using **out** parameters:

```
bool CountValues(int[] haystack, int needle, out int c) {
    c = 0;
    if (haystack == null) return false;
    for (int i = 0; i < haystack.Length; i++)
        if (haystack[i] == needle) c++;
    return true;
}
```

Sure, but **out** parameters are an hassle to use!

Nullable Types (3)

Solution 1: using **out** parameters:

```
int[] list = {1, 3, 5, 7};
int c;
bool hasElements = CountValues(list, 3, out c);
if (!hasElements) {
    Console.WriteLine("No elements to search for");
} else {
    Console.WriteLine("Found {0} elements", c);
}
```

Nullable Types (4)

Solution 2: using **nullable types**:

```
int? CountValues(int[] haystack, int needle) {
    int c = 0;
    if (haystack == null) return null;
    for (int i = 0; i < haystack.Length; i++)
        if (haystack[i] == needle) c++;
    return c;
}
```

How to use it?

Nullable Types (5)

Solution 2: using **nullable types**:

```
int[] list = {1, 3, 5, 7};

int? c = CountValues(list, 3);

if (c == null) {

    Console.WriteLine(“No elements to search for”);

} else {

    Console.WriteLine(“Found {0} elements”, c);

}
```

Nullable Types (6)

Other nullable tests:

```
int? variable = null;

if (variable.HasValue) {

    // not null, so we can access value

    int value = (int) variable;

    // or

    int value = variable.Value;

}
```

Better Nullable Tests

Checking for null variables can be easy

```
int result = countValues(array, value) ?? -1;
```

And calling methods on nulls, or accessing members of null variables:

```
int? x = possibleNullObject?.doGreatThings();
```

- In this case, returns null if the object is null, else calls method

```
int? value = possibleNullDictionary?[99];
```

- Returns value with key 99, or null if dictionary not initialized

Function Parameters

Day 1

Function Parameters (1)

- **Standard parameters:** all required, order relevant;

```
count = CountValues(array, 3);
```

- **Named parameters:** all required, order interchangeable;

```
count = CountValues(needle: 3, haystack: array);
```

- **Optional parameters:** some can be omitted, default values are used;

```
price = ApplyTax(3.99);  // uses standard 23% tax

price = ApplyTax(3.99, 0.13); // Uses 13% tax
```

- **Parameters as array:** variable number of arguments

```
concat = Concat("concatenate","these","strings");

concat = Concat(new string[] {"concatenate","these","strings"});
```

Function Parameters (2)

- **Standard parameters and Named Parameters:**
  - No special requirement
  - User can use varnames used in the function definition as argument names

```
void Func(int foo, string bar) { /* do something */ }
```

- **Using Standard parameters**

```
Func(42, “Hello World”);
```

- **Using Named parameters**

```
Func(foo: 42, bar: “Hello World”);

Func(bar: “Hello World!”, foo: 42);
```

Function Parameters (3)

- **Optional parameters**
  - Either at the end, or named parameters are required;
  - Each optional parameter need to define a default value.

```
float ApplyTax(float value, float tax = 0.23f) {

    return value + value * tax;

}
```

- **Use either as:**

```
price = ApplyTax(20.00f);

price = ApplyTax(20.00f, 0.13f);
```

Function Parameters (4)

- **Optional parameters and Named Parameters:**
  - When more than one parameter is optional;
  - And user might specify one, the other, or both.

```
float ComputePrice(float price, float tax = 0.23f, float discount = 0f)
{
    float value = price - price * discount;
    return value + value * tax;
}
```

- Use it with positional parameters:

```
float std = ComputePrice(29.3f);

float food = ComputePrice(29.3f, 0.13f);

float discounted = ComputePrice(23.2f. 0.23f, 0.10f);
```

- Use it with named parameters:

```
float discounted = ComputePrice(23.2f, discount: 0.10f);
```

Function Parameters (5)

- **Parameters as arrays:**
  - Use method with an array
  - Use method with some arguments as well
- Example:

```
string Concat(params String[] words) {

     return String.Join(" ", words);

 }
```

- Use both as:

```
string[] sentence = {"Hello", "cruel", "World!"};

Concat(sentence); // returns “Hello cruel World!”

Concat("Goodbye", "dear", "friend!"); // "Goodbye dear friend!”
```

Extension Methods

Day 1

Extension Methods (1)

- There is a class for some specific task;
- But the class misses a method you find useful;
- Instead of defining it somewhere else, can we extend the original class?

YES!

Process:

1. Create a static class inside your Namespace (or other you “use”);
2. Add public static methods;
3. Add as first argument to the method: `this Type var`

Example:

Allow a string to have a **NumberOfVowels** method.

Extension Methods (2)

```
using System;

namespace MyExtensionMethods {

    public static class MyStringExtensions {

        public static int NumberOfVowels(this String s) {

            int total = 0;

            for (int i = 0; i < s.Length; i++)

                if (“aeiouAEIOU”.contains(s.Substring(i,1)) total++;

            return total;

        }

    } }
```

Extension Methods (3)

```
using MyExtensionMethods;

using System;

namespace MyProgram {

    public class Program {

        public static void Main(String[] args) {

            string name = “IPCA”;

            int total = name.NumberOfVowels();

            Console.WriteLine(“{0} Vowels”, total);

        }

    } }
```

Extension Methods (4)

Other example: Convert string to int, with default value.

```
using System;

namespace MyExtensionMethods {

  public static class MyStringExtensions {

    public static int ToInteger(this String s, int defaultVal = 0)    {

      int outVal;

      return int.TryParse(s, out outVal) ? outVal : defaultValue;

     }

} }
```

Extension Methods (5)

Other example: Convert string to int, with default value.

```
using MyExtensionMethods;

using System;

namespace MyProgram {

    public class Program {

    public static void Main(string[] args) {

        string name = "IPCA";

        Console.WriteLine(name.ToInteger());   // prints 0

        Console.WriteLine(name.ToInteger(defaultVal: 10)); // prints 10

} } }
```

Operator Overloading

Day 1

Operator Overloading (1)

Suppose we did not have a Vector2 class available...

```
public class Vector2 {

    public float x, y;

    /* Standard Constructor */

    public Vector2(float x, float y) { this.x = x; this.y = y; }

    /* Clone-like Constructor */

    public Vector2(Vector2 o) { this.x = o.x; this.y = o.y; }
```

Operator Overloading (2)

Now we need to have Vector2 mathematics...

```
/* Add two vectors */

public void Add(Vector2 o) { this.x += o.x; this.y += o.y; }

/* Negate vector */

public void Neg() { this.x = -this.x; this.y = -this.y; }

/* Subtract vectors */

public void Sub(Vector2 o) { this.x -= o.x; this.y -= o.y; }
```

The usage of these methods is not very practical...

Operator Overloading (3)

Make standard math operators work on vectors…

```
public static Vector2 operator + (Vector2 left, Vector2 right) {
    return new Vector2(left.x + right.x, left.y + right.y);
}
public static Vector2 operator - (Vector2 v) {
    return new Vector2(-v.x, -v.y);
}
public static Vector2 operator - (Vector2 left, Vector2 right) =>
    new vector2(left.x - left.y, right.x - right.y);
```

Operator Overloading (4)

And comparisons?

```
public static bool operator == (Vector2 left, Vector2 right) =>
    left.x == right.x && left.y == right.y;
public static bool operator != (Vector2 left, Vector2 right) =>
    left.x != right.x || left.y != right.y;
```

Operator Overloading (5)

Operators that can be overloaded:

Unary Operators:

+, -, !, ~, ++, --, true, false

Binary Operators:

+, -, *, /, %, &, |, ^, <<, >>

Comparison Operators:

==, !=, <, >, <=, >= (in pairs)

You can define your own semantics, but try to be consistent with the operators' regular meaning.

Generics

Day 1

Generics (1)

Allows generic classes or methods, that work for different data types.

Examples are the Collection classes available in C#:

- List<GameObject> objects;
- Dictionary<string, string> map;
- Tuple <string, int, float> employee;

Generics (2)

Example 1: DIY array resizer:

```
T[] EnlargeArray<T>(T[] original) {
    T[] newArray = new T[original.Length + 10];
    for (int i = 0; i < original.Length; i++) {
        newArray[i] = original[i];
    }
    return newArray;
}
```

Generics (3)

Example 2: Pair

```
class Pair<T1,T2> {
  public T1 First { get; set; }
  public T2 Second { get; set; }
  public Pair(T1 f, T2 s) {
    First = f;
    Second = s;
  }
  public override string ToString() {
    return String.Format("({0},{1})", First, Second);
  }
}
```

Usage:

```
Pair<int,int> pt = new Pair<int,int>(10,20);

pt.First = 30;

Console.WriteLine(pt);
```

Delegates (Action and Func)

Day 1

Action and Func (1)

Generic Data types to store references to

- routines (**Action**)
- functions (**Func**)

Action and Func (2)

References to Routines:

```
public static void Hello(string who, int times) {
    for (int i = 0; i < times; i++)
        Console.WriteLine($"Hello, {who}");
}
public static void Main(string[] args) {
        Action<string> show = Console.WriteLine;
        show("Hello, World!");
        Action<string, int> hi = Hello;
        hi("IPCA", 10);
}
```

Action and Func (2)

References to Functions:

```
public static int Comprimento(string str)

{ return str.Length(); }


public static void Main(string[] args) {

        Func<int, int> absolute = Math.Abs;

        int dez = absolute(-10);

        Func<string, int> comp = Comprimento;

        int vinte = comp(“Internationalization”);

}
```

Action and Func (3)

Real World Example: Consider a class to represent a Game Object.

```
class GameObject {

    public void Update(GameTime gameTime) {

        // updates the object

    }

}
```

We need to add user-code to the update method.

Action and Func (4)

```
class GameObject {
    List<Action<GameTime>> updateActions = new List<Action<GameTime>>();


    public void Update(GameTime) {
        foreach (var action in updateActions) action(GameTime);
    }
    public void OnUpdate(Action<GameTime> action) {
        updateActions.Add(action);
    }
}
```

Action and Func (5)

```
GameObject car = new GameObject();

car.OnUpdate((GameTime x) => Console.WriteLine(“I’m in car update!”));

car.OnUpdate(

     (GameTime t) => Console.WriteLine(“Elapsed ” + t.totalSeconds));


// In the game main cycle

car.Update(gameTime); // will call both methods defined above;
```

Lambda Expressions

Day 1

Lambda Expressions (1)

Derived from Lambda Calculus formalism;

In Programming, a quick way to represent functions;

In C#, the easiest way to create an anonymous function.

Anonymous Function is a function without a name, stored in a variable.

Lambda Expressions (2)

Syntax:

<parameters> => *<expression>*

```
Func<int, bool> Even = x => x % 2 == 0;
```

Or

<parameters> => { *<statements>* }

```
Func<int, int> ToLowestEven = x => {
    if (Even(x)) return x; else return x-1;
};
```

Lambda Expressions (3)

Syntax:

Methods with no parameters:

```
Action Copyright = () => Console.WriteLine("(C) IPCA");
Copyright();
```

Explicit types:

```
Action<string> CR = (string n) => Console.WriteLine($“(C) {n}”);
CR(“Alberto”);
```

Lambda Expressions (3)

Define a function to square a float number.

```
Func<float, float> Square = x => x * x;

float y = Square(4);

Console.WriteLine(y);
```

When does this gets useful? See Linq, in the next slides for examples.

LINQ: Language Integrated Query

Day 1

LINQ: Language Integrated Query
(1)

Functional approach to manage collections.

```
using System.Linq;
```

Now, lists, arrays, and other generic collections have cow-powers!

Two different approaches:

- Declarative (kind of SQL-like)
- Explicit (call C# methods on collections)

LINQ: Language Integrated Query

(2)

Example 1: print sorted array of strings.

Declarative approach:

```
string[] names = {"João","Maria","Marcos","Joaquim"};

IEnumerable<string> query = from s in names orderby s select s;

foreach (var name in query) {

    Console.WriteLine(name);

}
```

LINQ: Language Integrated Query

(2)

Example 1: print sorted array of strings.

Explicit Approach:

```
string[] names = {"João","Maria","Marcos","Joaquim"};

IEnumerable<string> query = names.OrderBy(s => s);

foreach (var name in query) {

    Console.WriteLine(name);

}
```

LINQ: Language Integrated Query
(3)

Example 2: given an array of students, compute the grade average.

```
Student[] students = LoadSomeStudents();
```

Declarative Approach:

```
float mean = (from s in students select s.Grade).Average();
```

Explicit Approach:

```
float mean = students.Select(s => s.Grade).Average();
```

LINQ: Language Integrated Query

(4)

Example 3: grade average of passing students.

Declarative Approach:

```
float mean = (from s in students
              where s.Grade >= 9.5f
              select s.Grade).Average();
```

Explicit Approach:

```
float mean = students.Where(s => s.Grade >= 9.5f).
                      Select(s => s.Grade).Average();
```

LINQ: Language Integrated Query

(5)

Example 4: given an array of students, compute the grade average by genre.

Declarative Approach:

```
var divisao = from s in students
              group s.Grade by s.Gender into groups
              select groups;
foreach (var g in divisao) {
    Console.WriteLine("Gender: " + g.Key);
    Console.WriteLine("Average: " + g.Average());
}
```

LINQ: Language Integrated Query

(6)

Example 4: given an array of students, compute the grade average by genre.

Explicit Approach:

```
var divisao = students.GroupBy(s => s.Gender, s => s.Grade);

 foreach (var g in divisao) {

     Console.WriteLine("Gender: " + g.Key);

     Console.WriteLine("Average: " + g.Average());

 }
```

Q&A

Thank you!
See you next week!

IPCA
INSTITUTO POLITÉCNICO
DO CÁVADO E DO AVE

WORKSHOP C# AVANÇADO

3 de Abril
10 de Abril

Alberto Simões
Daniela da Cruz

Day 2

Contents
Day 2

Speaker
Daniela da Cruz

- Event-driven Programming
- Asynchronous Programming
- Reflection
- Attributes
- Dynamic Variables
- Memory Management

Event Driven Programming

Day 2

Event-driven programming

Event-driven programming responds to **events**.

An event is generated (or raised) when "something happens", such as the user pressing a button.

Often, events are generated by user action, but events can also be generated by the system starting or finishing work.

For example, the system might raise an event when a file that you open for reading has been read into memory or when your battery's power is running low.

Windows is an event-driven program. The OS waits until it detects an event such as the user clicking the mouse on a button. The click raises an event, which must be handled. The method that responds to the event is called the event handler. When the event is raised, the event handler, if one exists, is automatically executed by Windows.

Event-driven programming

Delegate

A delegate is very similar to a function pointer in C++. It is a reference type that encapsulates a method which has a specific signature and a return type.

Event

An event allows a class (or other object) to send notifications to other classes (or objects) that something has occurred. In simple terms an event is the outcome of a specific action.

Event-driven programming

In event driven programming you have a **publisher** (a class that exposes the event) and at least one **subscriber** (a class that subscribes to your event through the use of a delegate).

So, given that context, your publisher raises an event, more than likely through some user interaction or user choice, and your subscriber makes a decision on what to execute based on what event was raised, thus the term **event driven programming**.

To demonstrate the use of delegates & events in C# we're going to create an application that is driven by events.

Exercise

Create a class called *NumbersSequence* which basically consists on a list of random numbers.

Also create a method that traverses such list and prints a message each time two consecutive numbers are equal.

After that we will make this method to raise an event.

Asynchronous Programming

Day 2

Asynchronous programming

Asynchronous programming and threading is an important feature for concurrent or parallel programming.

Asynchronous programming may or may not use threading.

Asynchronous operation means that the operation runs independent of main or other process flow.

In general C# program starts executing from the Main method and ends when the Main method returns. In between all the operations runs sequentially one after another. One operation must wait until its previous operation finishes.

Asynchronous programming

```
static void Main(string[] args)
{
    DoTaskOne();
    DoTaskTwo();
}
```

Method “DoTaskTwo” would not be started until “DoTaskOne” finishes.

In other words method “DoTaskOne” blocks the execution as long it takes to finish.

In asynchronous programming a method called that runs in the background and the calling thread is not blocked.

After calling the method the execution flow immediately backs to calling thread and performs other tasks. Normally it uses Thread or Task classes.

Async programming - Thread

The constructor of Thread class accepts a delegate parameter of type

1. ThreadStart: This delegate defines a method with a void return type and no parameter.

2. And ParameterizedThreadStart: This delegate defines a method with a void return type and one object type parameter.

Async programming - Thread

CASE 1. How we can start a new thread with Start method:

```
static void Main(string[] args)
{
    Thread thread = new Thread(DoTask);
    thread.Start(); // Start DoTask method in a new thread

    // Do other tasks in main thread

}


static public void DoTask() {
    //do something in a new thread
}
```

Async programming - Thread

CASE 2. Use lambda expression instead of named method:

```
static void Main(string[] args)
{
    Thread thread = new Thread(() => {
        //do something in a new thread
    });

    thread.Start();

    // Do other tasks in main thread

}
```

Async programming - Thread

CASE 3. We don’t require the variable reference we can even start the thread directly like:

```
 static void Main(string[] args)
{
        new Thread(() => {
            //do something in a new thread
        }).Start();// Start a new thread

      //Do other tasks in main thread

}
```

Async programming - Thread

CASE 4. But if we want to control the tread object after it is created we require the variable reference. We can assign different values to the different properties of the object like:

```
static void Main(string[] args)
{
    Thread thread = new Thread(DoTask);
    thread.Name = "My new thread";// Assigning name to the thread
    thread.Priority = ThreadPriority.AboveNormal;// thread priority
    thread.Start();// Start DoTask method in a new thread

    //Do other tasks in main thread

}
```

Async programming - Thread

CASE 5. If we want to pass some data to the method we can pass it as a parameter of Start method. As the method parameter is object type we need to cast it properly.

```
static void Main(string[] args)
{
        Thread thread = new Thread(DoTaskWithParm);
        thread.Start("Passing string");// Start DoTaskWithParm method

        //Do other tasks in main thread

}

static public void DoTaskWithParm(object data)
{
        //we need to cast the data to appropriate object
}
```

Async programming - async/await

.NET framework introduced two new keywords to perform asynchronous programing: “**async**” and “**await**”.

To use “await” keyword within a method we need to declare the method with “async” modifier.

Async programming - async/await

await is used before a method call.

It suspends execution of `CallerWithAsync()` method and control returns to the calling thread that can perform other task.

```
private async static void CallerWithAsync()
{
    string result = await GetSomethingAsync();

    Console.WriteLine(result); // this line would not be executed
before GetSomethingAsync method completes
}
```

Async programming - async/await

The **async** modifier can only be used with methods returning a Task or void. It cannot be used with the entry point of a program, the Main method.

We cannot use ***await*** keyword before all the methods.

To use “await” the method must have to return “awaitable” type. Following are the types that are “awaitable”:

1. Task
2. Task<T>
3. Custom “awaitable” type. (Using a custom type is a rare and advanced scenario; we will not discuss it here)

Let’s go for some practice.

Exercise

- Create an web form and add 2 labels (one for async content and other for non asyn content)

- Create a class called ContentManagement with 6 methods:
  - `public string GetContent()`
  - `public int GetCount()`
  - `public string GetName()`
  - `public async Task<string> GetContentAsync()`
  - `public async Task<int> GetCountAsync()`
  - `public async Task<string> GetNameAsync()`

- Create 2 methods with the following signatures:
  - `public void NonAsync()`
  - `public async Task<long> Async()`

Reflection

Day 2

Reflection

.NET Framework's Reflection API allows you to fetch Type (Assembly) information at runtime programmatically.

At runtime, Reflection mechanism uses the Portable Executable (PE) file to read the information about the assembly.

Reflection enables you to use code that was not available at compile time.

Reflection

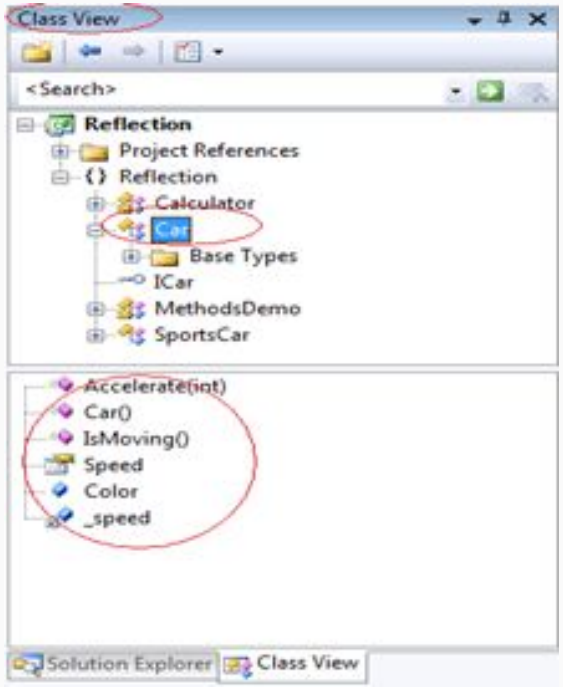
.NET Reflection allows application to collect information about itself and also manipulate on itself.

It can be used effectively to find all the types in an assembly and/or dynamically invoke methods in an assembly.

This includes information about the type, properties, methods and events of an object. With reflection we can dynamically create an instance of a type, bind the type to an existing object, or get the type from an existing object and invoke its methods or access its fields and properties.

Reflection

Using reflection, you can get any kind of information which you can see in class viewer, for example information of methods, properties, fields, and events of an object.



Reflection - System.Type

System.Reflection namespace and System.Type class plays very important role in .NET Reflection, these two works together and allow you to reflect over many other aspects of a type

The System.Type class defines a number of members that can be used to examine a type's metadata, a great number of which return types from the System.Reflection namespace.

Property	Returns
Name	The name of the data type.
FullName	The fully qualified name of the data type (including the namespace name).
Namespace	The name of the namespace in which the data type is defined.

Reflection - System.Type

It is also possible to retrieve references to further type objects that represent related classes, as shown in the following table:

Property	Returns Type Reference Corresponding To
BaseType	The name of the data type.
UnderlyingSystemType	The type that this type maps to in the .NET runtime (recall that certain .NET base types actually map to specific predefined types recognized by IL).

A number of Boolean properties indicating whether this type is, for example, a class, an enum, and so on.

```
IsAbstract, IsArray, IsClass, IsCOMObject, IsEnum, IsGenericTypeDefinition,
IsGenericParameter, IsInterface, IsPrimitive, IsPublic, IsNestedPrivate,
IsNestedPublic, IsSealed, IsValueType, IsPointer
```

Reflection - System.Type

Most of the methods of System.Type are used to obtain details of the members of the corresponding data type - the constructors, properties, methods, events, and so on. Quite a large number of methods exist, but they all follow the same pattern.

Return type	Methods (the method with the plural name returns an Array)
ConstructorInfo	GetConstructor(), GetConstructors()
EventInfo	GetEvent(), GetEvents()
FieldInfo	GetField(), GetFields()
InterfaceInfo	GetInterface(), GetInterfaces()
MemberInfo	GetMember(), GetMembers()
MethodInfo	GetMethod(), GetMethods()
PropertyInfo	GetProperty(), GetProperties()

Reflection - Assembly

The System.Reflection namespace provides a class called Assembly. We can use this Assembly class to fetch information about the assembly and manipulate the provided assembly; this class allows us to load modules and assemblies at run time.

The Assembly class provides the following methods to load an assembly at runtime:

- **Load()**: This static overloaded method takes the assembly name as input parameter and searches the given assembly name in the system.
- **LoadFrom()**: This static overloaded method takes the complete path of an assembly, it will directly look into that particular location instead of searching in the system.
- **GetExecutingAssembly()**: The Assembly class also provides another method to obtain the currently running assembly information using the GetExecutingAssembly() methods. This method is not overloaded.
- **GetTypes()**: The Assembly class also provides a nice feature called the GetTypes method which allows you to obtain the details of all the types that are defined in the corresponding assembly.

Exercise

- Download the DLL located at:
  - https://tinyurl.com/reflectionDLL

- Create a folder called “DLL” under your web project and add this dll in such folder.

- Create a new web form.

Attributes

Day 2

Attributes

- An **attribute** is a declarative tag that is used to convey information to runtime about the behaviors of various elements like classes, methods, structures, enumerators, assemblies etc. in your program.
- You can add declarative information to a program by using an attribute.
- A declarative tag is depicted by square ([ ]) brackets placed above the element it is used for.
- Attributes are used for adding metadata, such as compiler instruction and other information such as comments, description, methods and classes to a program.

Attributes

- The .Net Framework provides two types of attributes:
  - the pre-defined attributes
  - and custom built attributes.

- The three pre-defined attributes are:
  - Conditional
  - Obsolete
  - AttributeUsage

Attributes - Conditional

- This predefined attribute marks a conditional method whose execution depends on a specified preprocessing identifier.

- It causes conditional compilation of method calls, depending on the specified value such as Debug or Trace.

- For example,

```
[Conditional("DEBUG")]
```

Attributes - Obsolete

- This predefined attribute marks a program entity that should not be used. It enables you to inform the compiler to discard a particular target element.

- Syntax for specifying this attribute is as follows:

```
[Obsolete( message )]
[Obsolete( message, iserror)]
```

Attributes - AttributeUsage

The pre-defined attribute **AttributeUsage** describes how a custom attribute class can be used. It specifies the types of items to which the attribute can be applied.

Syntax for specifying this attribute is as follows:

```
[AttributeUsage( validon,  AllowMultiple=allowmultiple, Inherited=inherited)]
```

- The parameter *validon* specifies the language elements on which the attribute can be placed. It is a combination of the value of an enumerator AttributeTargets. The default value is AttributeTargets.All.
- The parameter *allowmultiple* (optional) provides value for the AllowMultiple property of this attribute, a Boolean value. If the argument is set to true, then the resulting attribute can be applied more than once to a single entity. The default is false (single-use).
- The parameter *inherited* (optional) provides value for the Inherited property of this attribute, a Boolean value. If it is true, the attribute is inherited by derived classes. The default value is false (not inherited).

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Constructor |
AttributeTargets.Field | AttributeTargets.Method | AttributeTargets.Property,
AllowMultiple = true)]
```

Dynamic variables

Day 2

Dynamic variables

A dynamic variable can have any type.

Its type can change during runtime.

The downside is that performance suffers and you lose compile-time checking.

The dynamic keyword is used to replace other types such as int, bool or string.

Dynamic variables

```
dynamic a;

a = 1;

a = “Hi”;

a = Test();


dynamic Test()
{
        return 1;
}
```

Dynamic variables

```
class Program
{
    static void PrintValue(dynamic val)
    {
        Console.WriteLine(val);
    }

    static void Main(string[] args)
    {

        PrintValue("Hello World!!");
        PrintValue(100);
        PrintValue(100.50);
        PrintValue(true);
        PrintValue(DateTime.Now);
    }
}
```

Dynamic variables

Points to Remember :

- The dynamic types are resolved at runtime instead of compile time.
- The compiler skips the type checking for dynamic type. So it doesn't give any error about dynamic types at compile time.
- The dynamic types do not have intellisense support in visual studio.
- A method can have parameters of the dynamic type.
- An exception is thrown at runtime if a method or property is not compatible.

Memory Management

Day 2

Memory Management

.NET memory management is designed so that the programmer is freed from the task of consciously having to allocate and dispose of memory resources.

An application is made up of two things:

- The code itself, and
- The data that stores the state of the application during execution.

Memory Management

When a .NET application runs, four sections of memory (heaps) are created to be used for storage:

1. The Code Heap stores the actual native code instructions after they have been compiled.

2. The Small Object Heap (SOH) stores allocated objects that are less than 85K in size

Memory Management

3. The Large Object Heap (LOH) stores allocated objects greater than 85K (although there are some exceptions, which we won't discuss here)

4 . Finally, there's the Process Heap, but let's not go there just yet

Memory Management

Applications are usually written to encapsulate code into methods and classes, so .NET has to keep track of chains of method calls as well as the data state held within each of those method calls.

When a method is called, it has its own environment where any data variables it creates exist only for the lifetime of the call.

A method can also get data from globals/static objects, and from the parameters passed to it.

Memory Management

Stack

The stack is used to keep track of a method's data from every other method call.

When a method is called, .NET creates a container (a stack frame) that contains all of the data necessary to complete the call, including parameters, locally declared variables and the address of the line of code to execute after the method finishes.

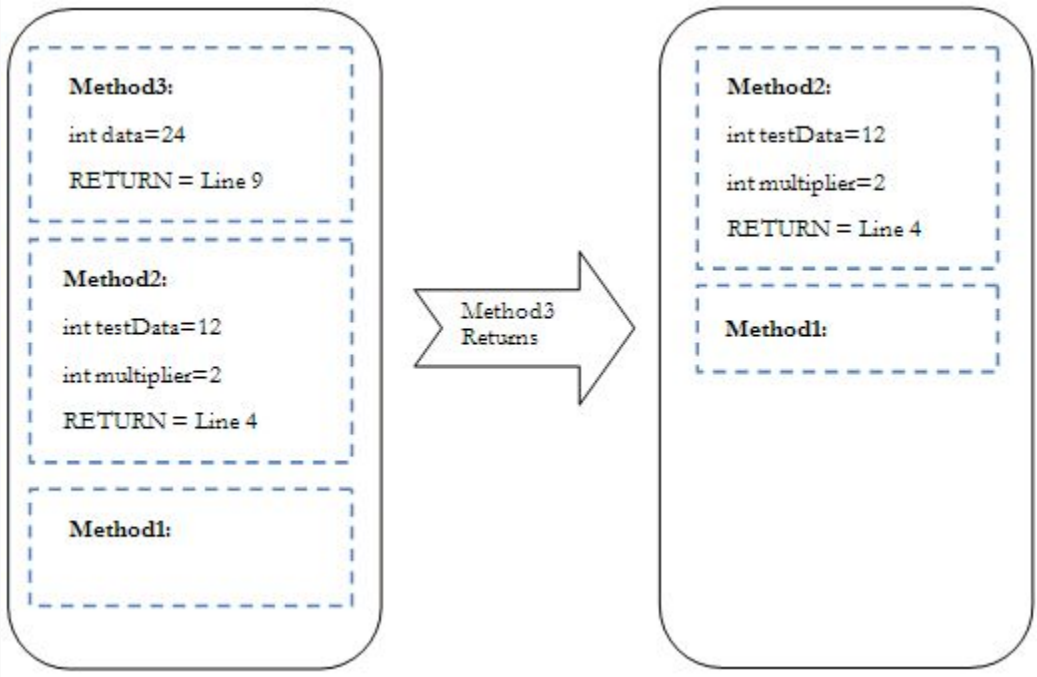
For every method call made in a call tree (i.e. one method that calls another, which calls another… etc.), stack containers are stacked on top of each other.

When a method completes, its' container is removed from the top of the stack and the execution returns to the next line of code within the calling method (with its own stack frame).

The frame at the top of the stack is always the one used by the current executing method.

Memory Management

```
1 void Method1 ()
2 {
3     Method2(12);
4     Console.WriteLine("Goodbye");
5 }
6 void Method2(int testData)
7 {
8     int multiplier=2;
9     Console.WriteLine("Value is " +
            testData.ToString());
10    Method3(testData * multplier) ;
11 }
12 void Method3(int data)
13 {
14    Console.WriteLine("Double " +
            testData.ToString());
15 }
```



Memory Management

Heap

The stack can store variables that are the primitive data types defined by .NET.

These include the following types: Byte, SByte, Int16, Int32, Int64, UInt16, UInt32, UInt64, Single, Double, Boolean, Char, Decimal, IntPtr, UIntPtr, Structs

These are primitive data types and are collectively called Value Types.

Any of these data types or struct definitions are usually stored on the stack.

On the other hand, instances of everything you have defined, including: **Classes, Interfaces, Delegates, Strings, Instances of "object"**

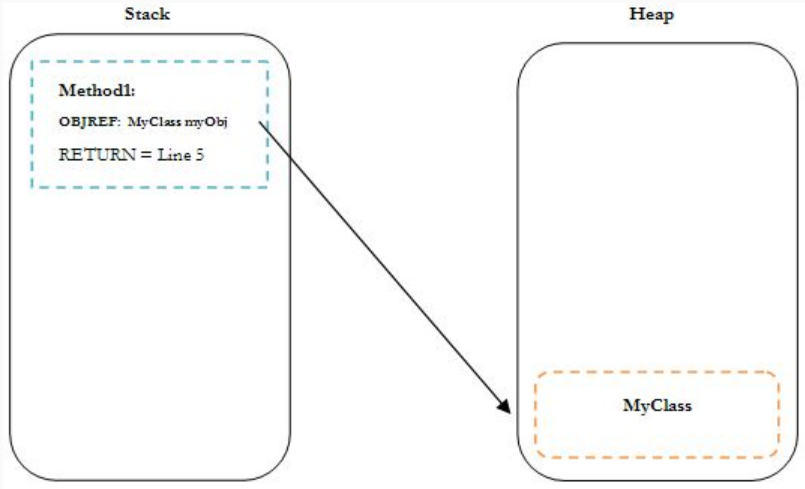
... are all referred to as "reference types", and are stored on the heap

Memory Management

When an instance of a reference type is created (usually involving the new keyword), only an object reference is stored on stack.

The actual instance itself is created on the heap, and its' address held on the stack.

```
1 void Method1()
2 {
3       MyClass myObj=new MyClass();
4       Console.WriteLine(myObj.Text);
5 }
```



Memory Management

Recalling:

When a .NET application runs, two sections of memory are reserved for storing objects allocated during execution (we'll ignore the other two sections of the heap for now).

The memory sections are called the **Small Object Heap (SOH)** and the **Large Object Heap (LOH)**.

The SOH is used to store objects smaller than 85 K, and the LOH for all larger objects.

Memory Management

Consider the following code:

```
class MyClass
{
    string Test="Hello world Wazzup!";
    byte[] data=new byte[86000];
}
```

The string will be allocated on the SOH, and its object reference held by the instance of the class; the byte array will be allocated onto the LOH, as it's bigger than 85 KB.

Memory Management

Automatic garbage collection is just a bunch of code that runs periodically, looking for allocated objects that are no longer being used by the application.

It frees developers from the responsibility of explicitly destroying objects they create, avoiding
the problem of objects being left behind and building up as classic memory leaks.

The GC runs on a separate thread when certain memory conditions are reached (which we'll discuss in a little while) or when the application begins to run out of memory.

The developer can also explicitly force the GC to run using the following line of code:

GC.Collect();

Forcing the GC to collect.

This is never really a good idea because it can cause performance and scalability problems.

C# 7

Near Future... well, Present!

C#7

(Available Now)

- Out variables declaration;
- Tuples Syntax
- Pattern Matching
- Pattern based Switch command
- Local Functions
- More expression-bodied members
- Throw Expressions
- Generalized async return types
- Numeric literal syntax improvements

Q&A

Thank you!