C# Course:

Introduction:  
  
[C# vs .NET](#_C#_vs_.NET:)

[CLR](#_CLR:)

[Architecture of .NET Applications](#_Architecture_of_.NET)

Our First C# Application

C# vs .NET:

C# is a programming language

.NET Framework:

.Net is a Framework for building applications on Windows (VB.NET, F#, C#)

2 components:  
1) CLR (Common Language Runtime)

2) Class Library

CLR:

History of C#  
Before C# two languages  
C/C++ code compiled to the native code for the machine it was to be run on.  
Can’t run on computer with different architecture.

Java is translated to bite code Same as C#  
  
C# compiled to IL or Intermediate Language.

CLR application in memory to translate IL code to Machine Code, Just in Time Compilation (JIT)

Architecture of .NET Applications:

Application is made of Classes

Classes are building blocks of application.

**Class: Container**

Data (property, Attributes)

Methods (Functions)

**Methods/Functions** have behavior, they execute code, do things for us

**Data** represents the state of the application

Example:

Class Car:

Data:  
Make

Model

Color

Functions:  
Start()

Move()

**Namespace:**

As the number of Classes grows in your application we need a way to organize these classes

Similar classes belong to the same namespace

**Assembly (DLL/EXE)**:  
container for related namespaces.

Single unit of deployment of .NET Applications.

Compiler builds one or more Assemblies depending on how you built your code.

**Solution:**  
Can have one or more Projects

VS:  
Properties:  
AssemblyInfo.cs (information for the executable)

References:  
what you may need to use in your project

App.config (xml)  
Configuration for the application

Connection strings to database

Program.cs:  
Using Statements (use a class that is not in the application)

Systems = primitives

Generics = Collections

All solutions have a class called program

Main is the entry point to the application

Methods have input and output  
input = parameter/arguments

C# is a case sensitive language

Variables and Constants

Variable: a name given to a storage location in memory

Constant: an immutable value

Constants create safety in the application.

Pi = 3.14 would be a constant in a mathematical application

**Declaring Variables**  
  
Type, Identifier, ;

int number = 3;

int num1;

can’t use a variable unless you initialize it.

**Declaring constants:**  
  
const, Type, Identifier, value, ;

Must set a value when declaring a constant

**Rules for Variable Identifiers:**

Can’t start with a number: 1route would be oneroute

Can’t have whitespace: first name would be firstName

Can’t be reserved keyword: int can be @int

Use meaningful names to make it easier to read and maintain

**Naming conventions:**  
  
Camel Case: firstName

Pascal Case FirstName

Hungarian Notation: strFirstName

For local Variables:  
Use CamelCase (string firstName = “”;)

For Constants:

Use Pascal case ( cons tint MaxZoom = 5;)

**Primitive Types in C#:**  
  
Integral Numbers: Byte, short, int, long,

Real Numbers: float, double, decimal

Character: char

Boolean: bool

**Real Number considerations:**  
float Single Bytes = 4

double Double Bytes = 8

decimal Decimal Bytes = 16

float number = 1.2f;

decimal num2 = 1.2m;

**Non-Primitive Types:**

String

Array

Enum

Class

Overflowing:

Exceeding the allowed maximum for a type variable storage, C# doesn’t have built in Overflow checking

Byte number = 255;

number = number +1; //0

Use checked to help avoid overflowing:  
checked  
{

byte number = 255;  
 number = number + 1;

}

this will cause an exception that should be caught and handled.  
do we need it? never have used it at all  
  
Scope:

Where a variable/constant has meaning

{

byte a = 1;

{

byte b = 2;

{

byte c = 3;

}

}

}

variable is only available in the block of code.

Type Conversion

Implicit Type Conversion

Explicit type Conversion (casting)

Conversion between non-compatible types

**Implicit:**

byte b = 1; = 00000001

int i = b; = 00000000 00000000 00000000 00000001

100 % types are compatible then types can be converted implicitly

int I = 1;

float f = I;

Implicit conversion

Non Implicit:  
int I = 1;

byte b = I;

Data loss because integer is 4 bytes, when there is a chance for data loss the compiler will not allow implicit conversion.

**Explicit:**  
int i = 1;

byte b = (byte)i; (Casting)

float f = 1.0f;

int I = (int)f;

Non-compatible types:

string s = “1”;

int I = (int)s; This will not compile because a string is not compatible with Integer

will not be handled by the Explicit casting

to handle these non-compatible types we need to use: Convert Class (.NET)  
Convert.ToInt32();

or

int.Parse();

all primitive types have a parse method. Takes a string and tries to convert it to the type.

ToByte

ToInt15

ToInt32

ToInt64

### C# Operators

Arithmetic Operators

Comparison Operators

Assignment Operators

Logical Operators

Bitwise Operators

Arithmetic Operators:

Add: a + b

Subtract: a - c

Multiply: a \* d

Divide: a / b

Remainder: a % c

Increment: i++: i = i + 1

Decrement: i--: I = i – 1

**Postfix increment:**

a = 1;

var b = a++;

result: b = 1, a = 2

**Prefix increment:**a = 1;

var b = ++a;

a is incremented first so the result is

b = 2 and a = 2

**Comparison Operators:**  
Equal ==

Not Equal !=

Greater Than >

Less Than <

Greater or Equal >=

Less than or equal <=

object.equals

**Assignment Operators:**

assignment: = a = 1;

addition assignment += a += 3; a = a + 3;

subtraction assignment -= a -= 3; a = a - 3;

multiplication assignment \*= a \*= 3; a = a \* 3;

division assignment /= a /= 3; a = a / 3;

**Logical Operators: (Boolean expressions)**

And && a && e;

Or || a || d;

Not ! !a;

**Bitwise Operators (low level programming)**sockets, encryption

And & a&b;

Or | a|b;

### Commenting in Code:

Single-line Comment: // Here is a single-line comment

Multi-line Comment: /\* Here is mutli-line comment \*/

keep comments to minimum, only when required:  
Whys, how’s and constrains.

not what the code is doing.

# Non - Primitive Types

### Classes

Classes are the building blocks of the application

Combine related variables (fields) and functions (methods)

Example:  
  
Person Class:  
Fields:

Name: String

Age: Integer

Height: Float

Weight: byte

Methods:

Walk()

Talk()

Eat()

Sleep()

Class is a blueprint to create objects, objects are instances of a class:  
  
Objects talk to each other and collaborating to provide some functionality

How to create a class?  
  
public class Person (access modifier, Keyword, NameOfClass)

variables = fields with access modifier, type and Name;

Methods = access modifier, type returned, Name

Create an Object (instances of class)

Person newPerson = new Person();

or

var newPerson = new Person ();

reference type must have memory allocated for the object.

newPerson.Method();

Static Modifier:

public class Calculator.

Static members can be access from class itself.

Calculator.Add();

only one instance of Calculator should exist in memory. only one entry point

Date/Time class is static as well.

### STRUCTS

similar to classes, syntax is similar to a class, just use struct key word.

combined related fields and methods together.

public struct RgbColor

{

public int Red;

public int Green;

public int Blue;

}

99% will create new types using classes not structures.

Structure for small light weight objects, RBG or Point with X and y

### Arrays

What is an Array

A data structure to store a collection of variables of the same type, with a fixed size.

Declaring Arrays

int [] numbers = new int[3];

numbers.Add(2);

numbers

this also allocates memory for the array at the time of initialization, it becomes and object.

Initializing Arrays

int [] numbers = new int[3];

this also allocates memory for the array at the time of initialization, it becomes and object.

Access Array Elements

arrays are 0 based index starting with 0 instead of 1

### Strings

String is a sequence of characters “Hello World”

**Creating Strings:**

string filename = “TEST”

Concatenate Strings  
string fullName = firstName + “ “ + lastName

string name = string.Format(“{0} {1}”, firstName, lastName);

joining array of objects:  
var numbers = new int[3]{1,2,3};

string list = string.Join(“,” numbers);

**String Elements:**  
can be accessed by index:  
  
var name = “Angel”;

char firstChar = name[0];

Strings are immutable, once you create them you can’t change them.

**Special characters:**

\n == new line

\t == tab

\\ == blackslash

\’ == single quotation mark

\” == double quotation mark

Verbatim Strings

string path = “C:\\projects\\project1\\folder1”;

string path (@”C\projects\project1\folder1);

### Enums

Enum: A set of Name/Value pairs (constants)

Enum Shipping methods:

public enum ShippingMethod

{

RegularAirMail = 1,

RegisterAirMail = 2,

Express = 3

}

dot notation:  
var method = ShippingMethod.Express;

### Reference Types and Value Types

2 main types which we use to create other types:  
  
Structures: example int32, int16

Primitive Types, Custom structures

small types, no more than 8 bytes

Classes: = example string, array

Custom classes.

Memory management is different for both structures and classes

Structures: = Value Types   
Allocated on stack

Memory allocation done automatically

Immediately removed when out of scope  
  
Classes = Reference types

You the program need to allocate memory yourself

Class class = new Class():

this tells the run time to create a memory allocated on heap.

when it goes out of scope it will not go off the heap immediately

Garbage collected by CLR. (automatic removal of things out of scope)

### Conditional Statements

Types of Conditional Statements:  
1) If/Else statements

2) Switch/case Statements

3) Conditional Operator a ?b:c

if(condition)

{

SomethingtoDo

}

else if (condition)

{

SomethingToDo

}

else

{

SomethingToDo

}

switch(role)

{

case Role.Admin:

codetodo;

break;

case Role.Moderator:

codetodo;

break;

default:

codetodo

break;

}

tertiary:  
array.count == 10 ? thisiftrue : thisiffalse;

### Iteration Statements:

For Loops

Foreach Loops

While Loops

Do-While Loops

**For Loops:**  
for(i = 0; I < 10; i++){}

**Foreach Loops** (enumerable objects)

foreach (var item in items){}

**While Loops:**  
while ( i<10){ i++}

**Do While loops** are executed at least once. because the condition is at the end of the execution.

do

{

//code to do at least once

i++ counter or bool you can use to determine condition

}while (i < 10);

**Breaks and Continue**

**break:** jumps out of the loop

**continue:** jumps into the next iteration

### Arrays and Lists

Multidimensional Arrays

Generic Lists

Types of Arrays in C#

**Single Dimension**: var array = new int[6]{ 1, 2, 3, 4, 5, 6}

Multi Dimension Array: like a matrix

Rectangular 3x5 Jagged (array of arrays)

0, 1, 2, 3, 4 0, 1, 2, 3

0, 1, 2, 3, 4 0, 1, 2, 3, 4

0, 1, 2, 3, 4 0, 1, 2

CLR is optimized around single Dimension arrays

to model a matrix is faster to do Jagged versus a Rectangular

**Rectangular array syntax**:  
**var matrix = new int[3, 5];**

need to specify the dimensions of the array

this one has 3 rows and 5 columns

Initialize with values:

var matrix = new int[3,5]

{

{1, 2, 3, 4, 5},

{2, 3, 4, 5, 7},

{12, 13, 14, 15, 16)

};

var element = matrix[0, 0] = 1

rectangular 3D

var colors = new matrix int[3, 5, 4];

**Jagged array syntax:**  
0 1 2 3

0 1 2 3 4

0 1 2

**var jagged = new int[3][ ];**

first a single Dimension array is created showing the number of elements at the top level.

Then initialize the elements of the array:

array[0] = new int[4];

array[1] = new int[5];

array[2] = new int[3];

access an element of this array we use two square brackets:  
array[0][0] = 1;

**Array Class**Length

**-------------**

Clear()

Copy()

IndexOf()

Reverse()

Sort()

Array.Clear() called on the array class itself not on the literal array  
 These are static methods

numbers.Length this is not a static method

### Lists

Lists are mutable (can be changed)

Array: fixed size

List: dynamic size of same type

Initialize:  
var numbers = new List<int>();

Generic type must determine the type in the <>

var numbers = new List<int> { 1, 2, 3, 4 };

Methods:  
Add()

AddRange()

Remove()

RemoveAll()

IndexOf()

Contains()

Count