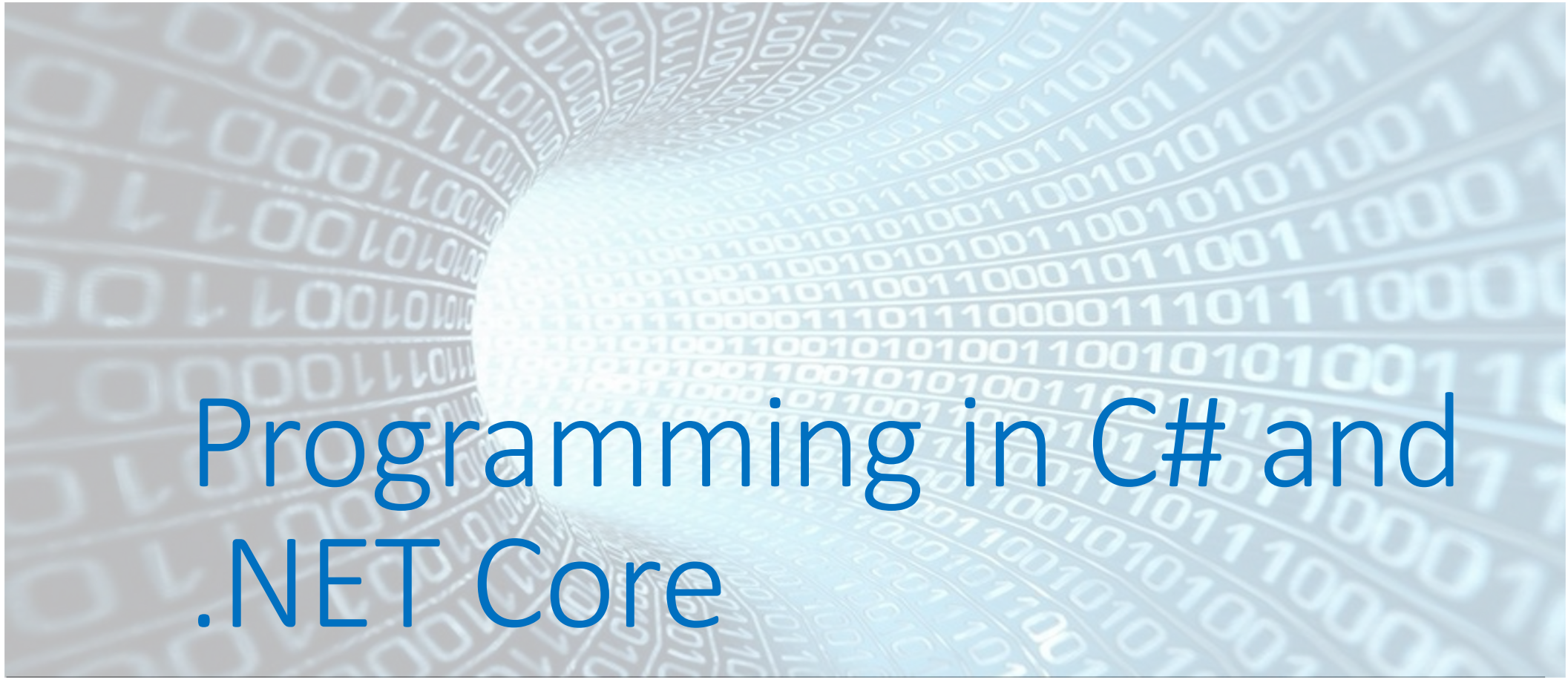


Programming in C# and .NET Core

RICCARDO TERRELL

Computer Programming in C# and .NET Core

Agenda

- Basic of Programming language
- What C# and .NET Core
- Install C# and .NET Core
- C#
  - Syntax
  - Working with Objects
  - Value vs Reference types
  - Control Flow
  - Collections
  - Loops
  - Generics
  - LINQ
  - ….

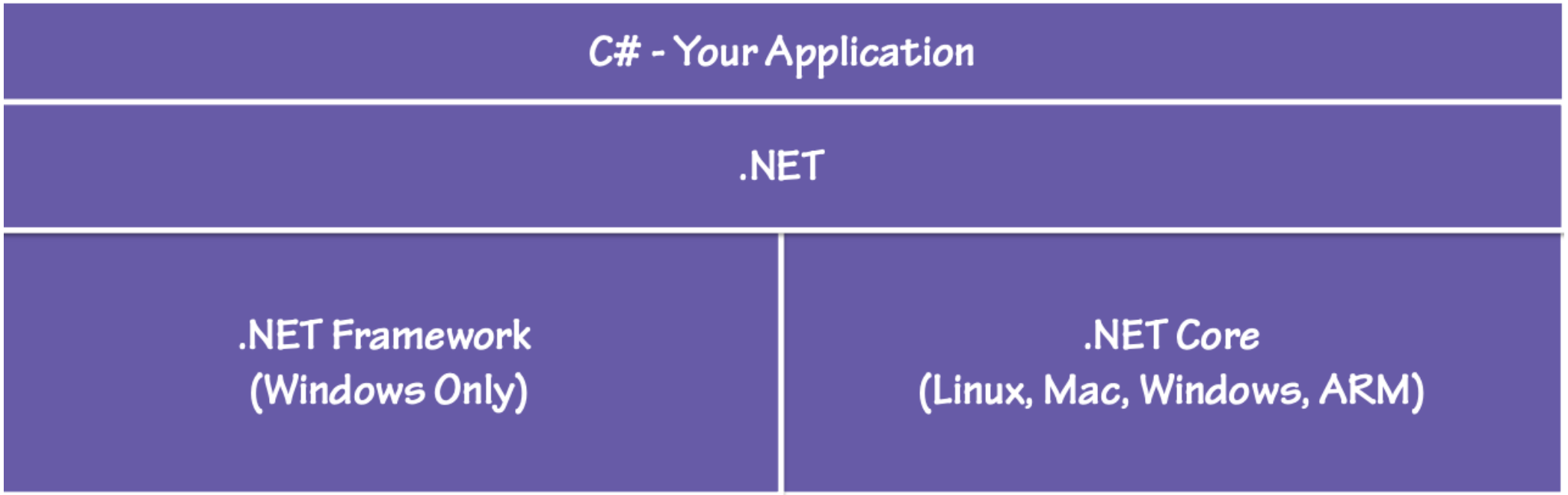
C# Fundamentals

Basic Concepts

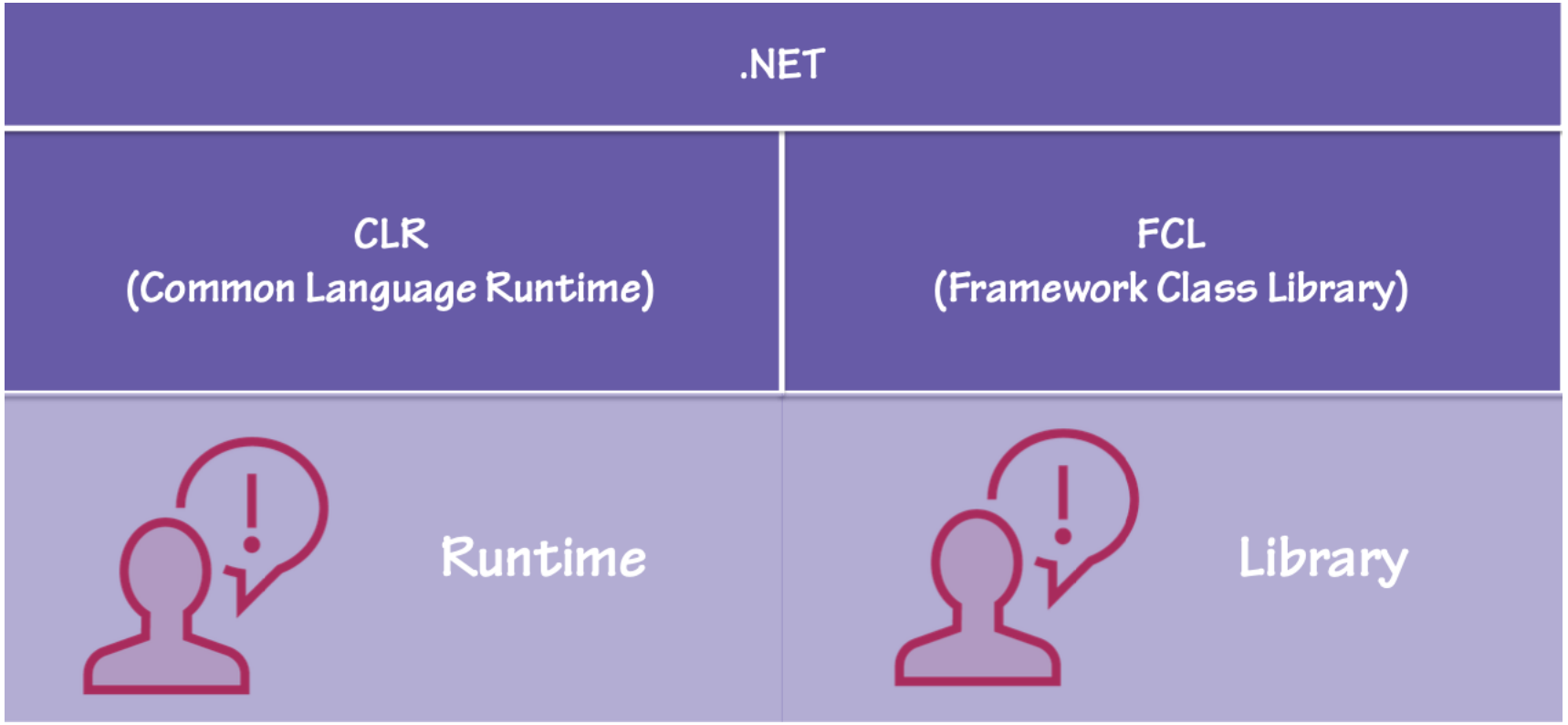
Why to learn C# ?

- simple and easy to learn
- modern and for general purpose
- structured language
- object oriented
- component oriented
- produces an efficient programs
- part of .Net Core
- can be compiled on a variety of computer platforms

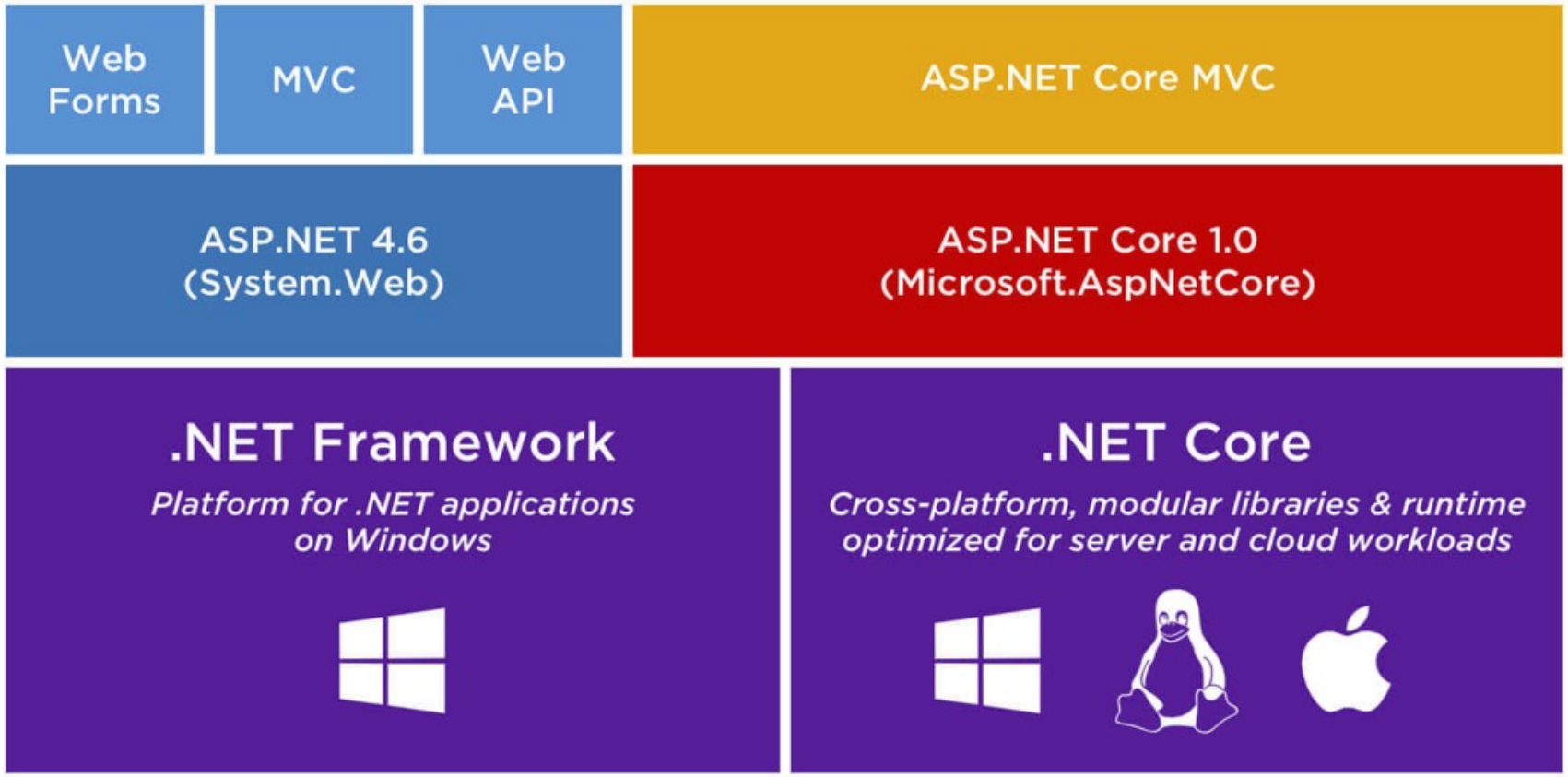
.NET Frameworks



.NET Frameworks



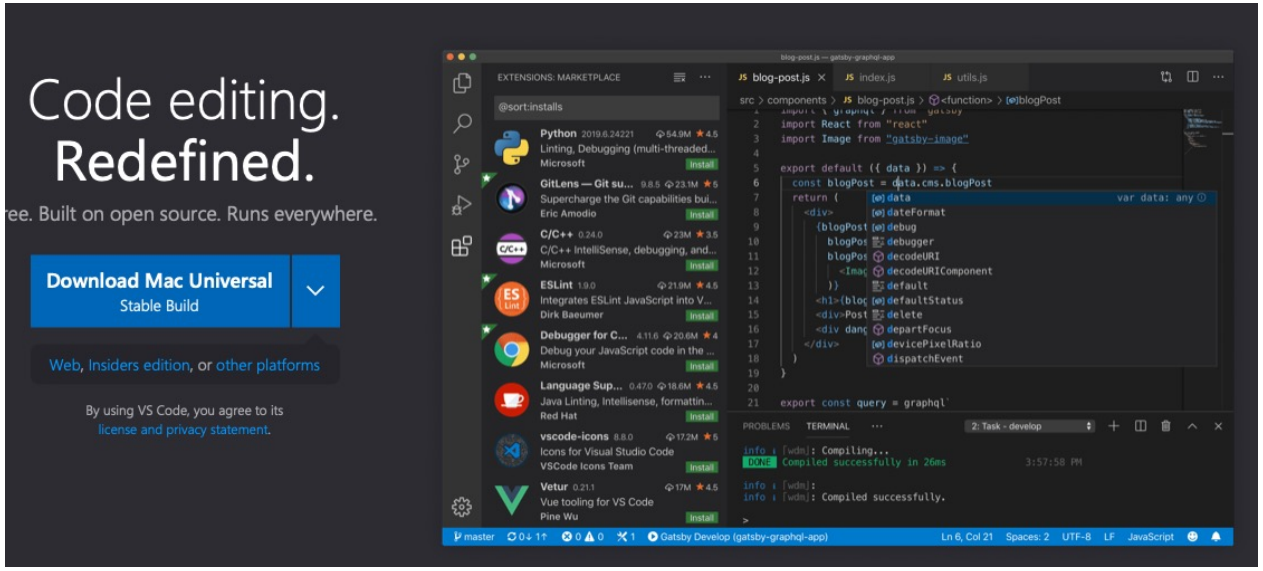
.NET Frameworks



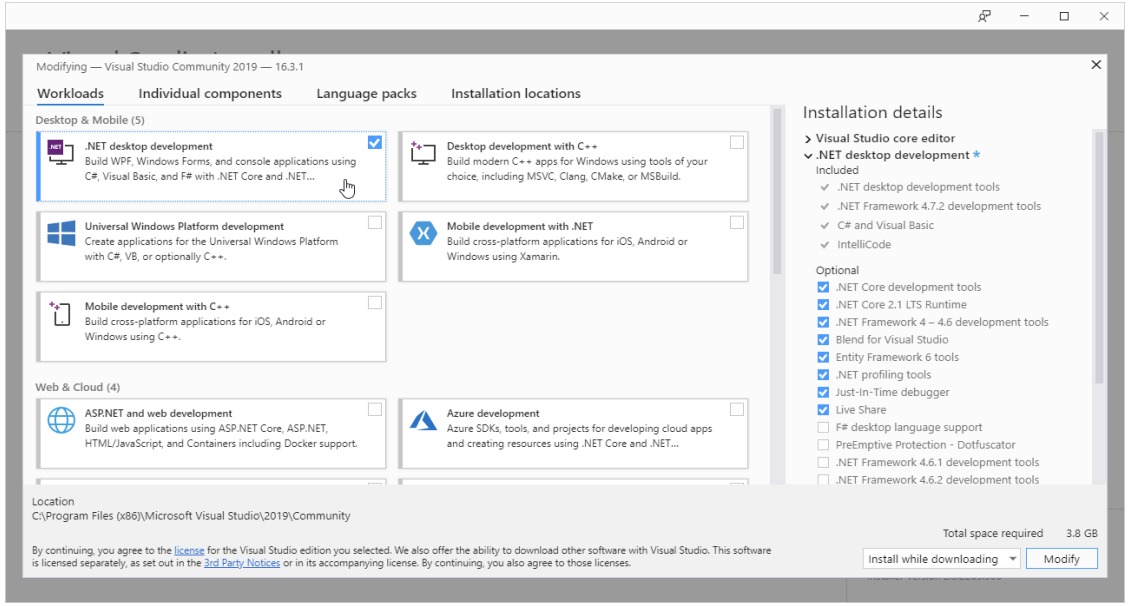
Install .NET Core and IDE

C# .NET Core IDE

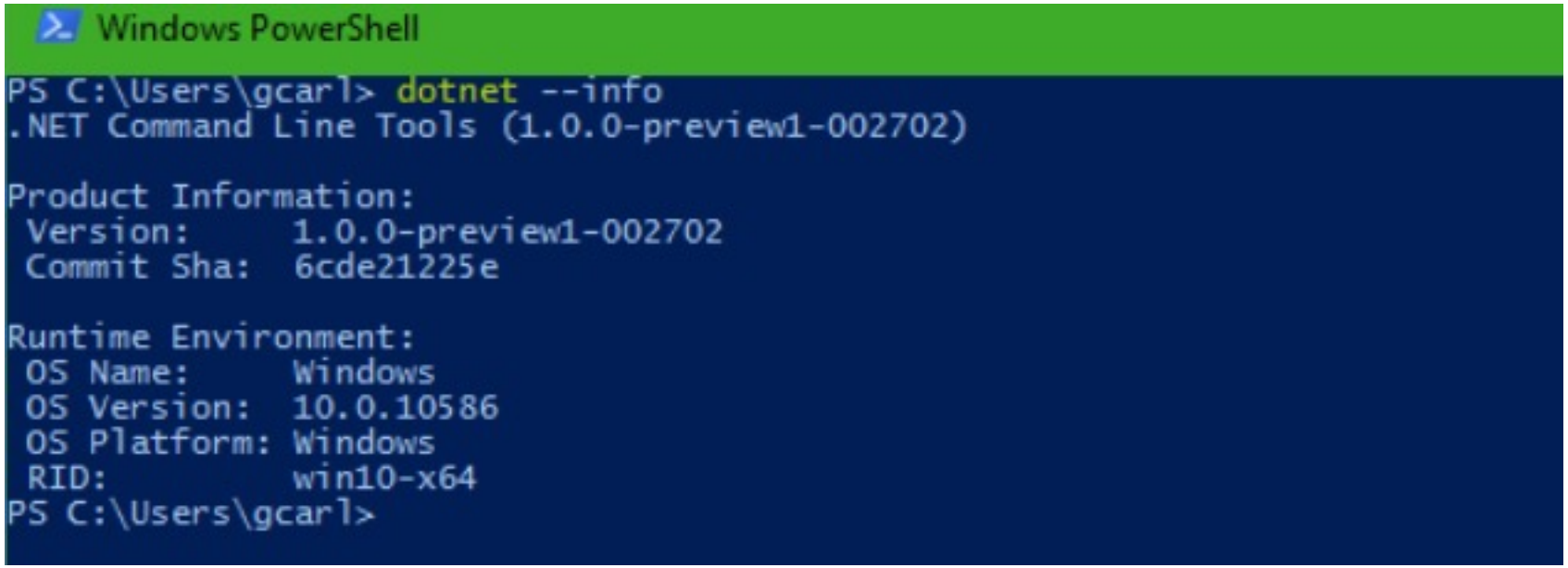
https://code.visualstudio.com



https://visualstudio.microsoft.com/vs/community/



.NET Core Installation

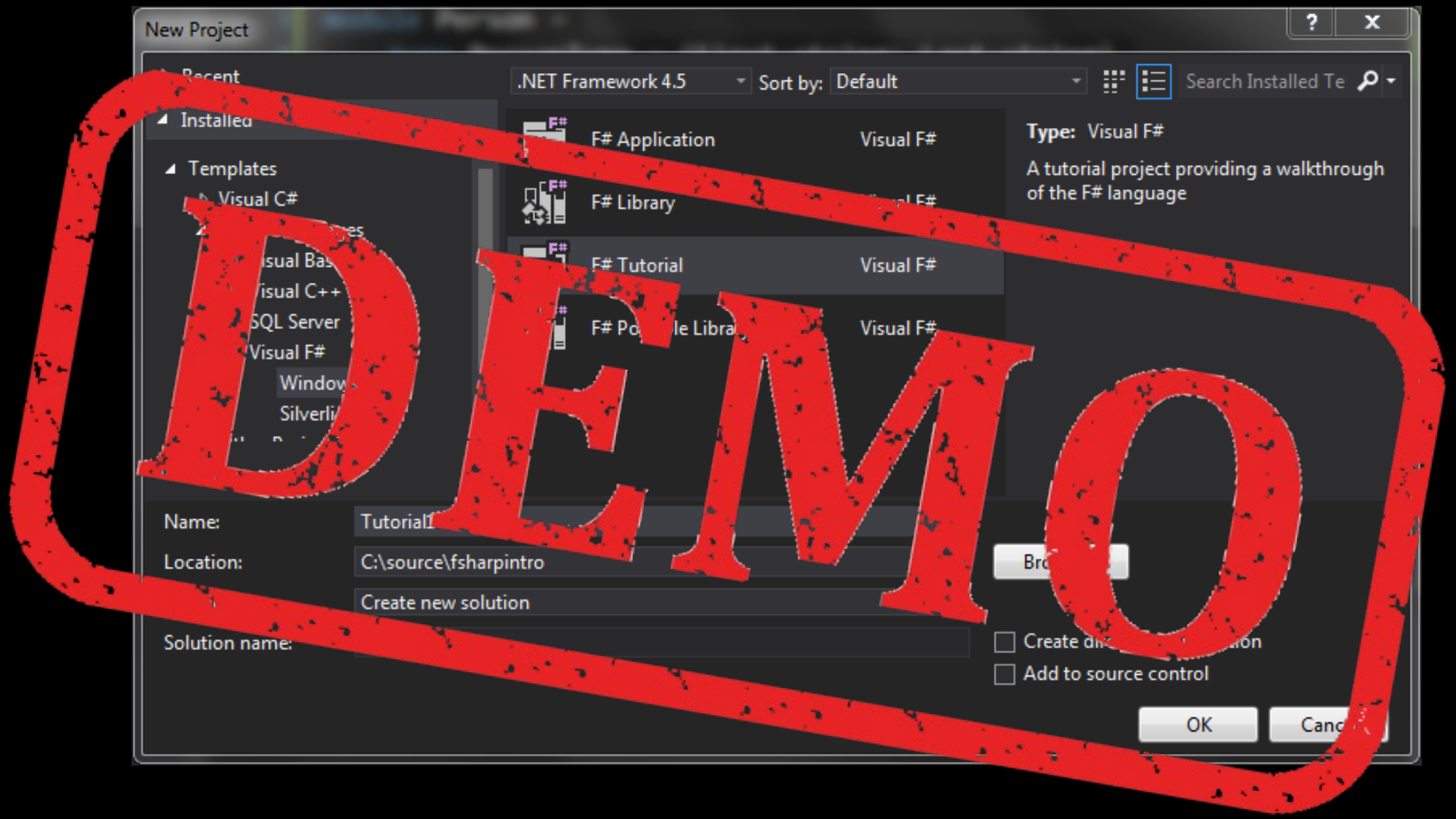


.NET Core project

.NET Core CLI

Basic Concepts

Command Syntax	Description
`new`	Initialize a basic .NET project
`restore`	Restore dependencies specified in the .NET project
`build`	Build a .NET project
`publish`	Publish a .NET project for deployment (including the runtime)
`run`	Compile and immediately execute a .NET project
`test`	Run unit tests using the test runner specified in the project
`pack`	Create a NuGet package


New Project
.NET Framework 4.5
Sort by:
Default
Search Installed Te
Installed
Templates
Visual C#
Visual C++
SQL Server
Visual F#
F# Application
Visual F#
F# Library
F# Tutorial
Visual F#
Type: Visual F#
A tutorial project providing a walkthrough of the F# language
Name:
Tutorial1
Location:
C:\source\fsharpintro
Create new solution
Solution name:
Add to source control
OK
DEMO

C# Fundamentals

Basic Concepts

Program Structure

C# program structure comprises from these parts:

- namespace declaration
- a class
- class attributes
- class methods
- a main method
- statements and expressions

C#
Writing and Running a Program

```
1   using System;
2
3   class Greetings
4   {
5       static void Main()
6       {
7           Console.WriteLine("Greetings!");
8       }
9   }
```

C#
Writing and Running a Program

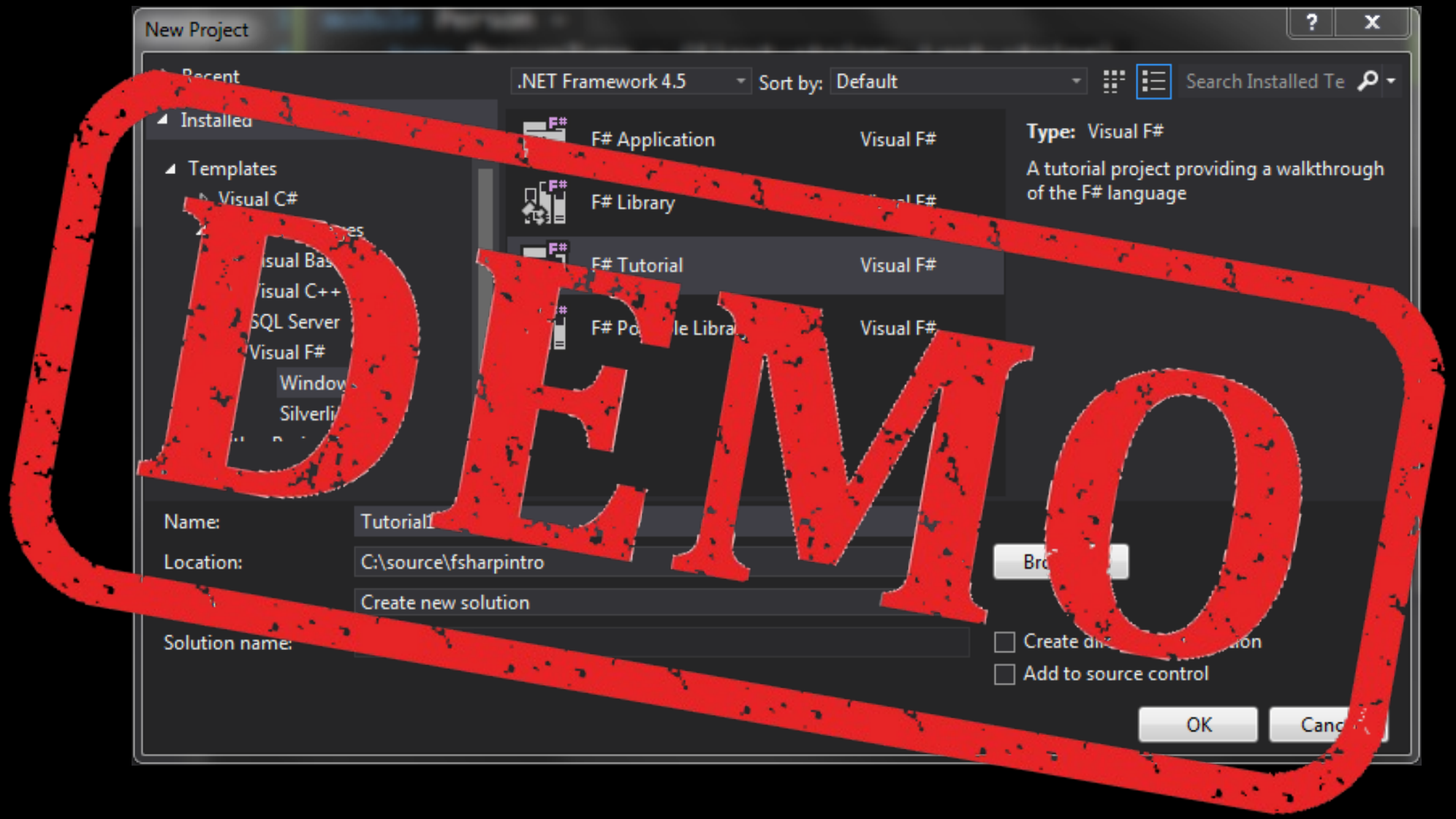
Namespaces and Code Organization

```
1   using static System.Math;
2
3   namespace Utilities
4   {
5       public class Calc
6       {
7           public static double Pythagorean(double a, double b)
8           {
9               double cSquared = Pow(a, 2) + Pow(b, 2);
10              return Sqrt(cSquared);
11          }
12      }
13  }
```

C#
Writing and Running a Program

Namespaces and Code Organization

```
1   using Utilities;
2   using System;
3
4   using Crypto = System.Security.Cryptography;
5
6   namespace NamespaceDemo
7   {
8       class Program
9       {
10          static void Main()
11          {
12              double hypotenuse = Calc.Pythagorean(2, 3);
13              Console.WriteLine("Hypotenuse: " + hypotenuse);
14
15              Crypto.AesManaged aes = new Crypto.AesManaged();
16
17              Console.ReadKey();
18          }
19      }
20  }
```

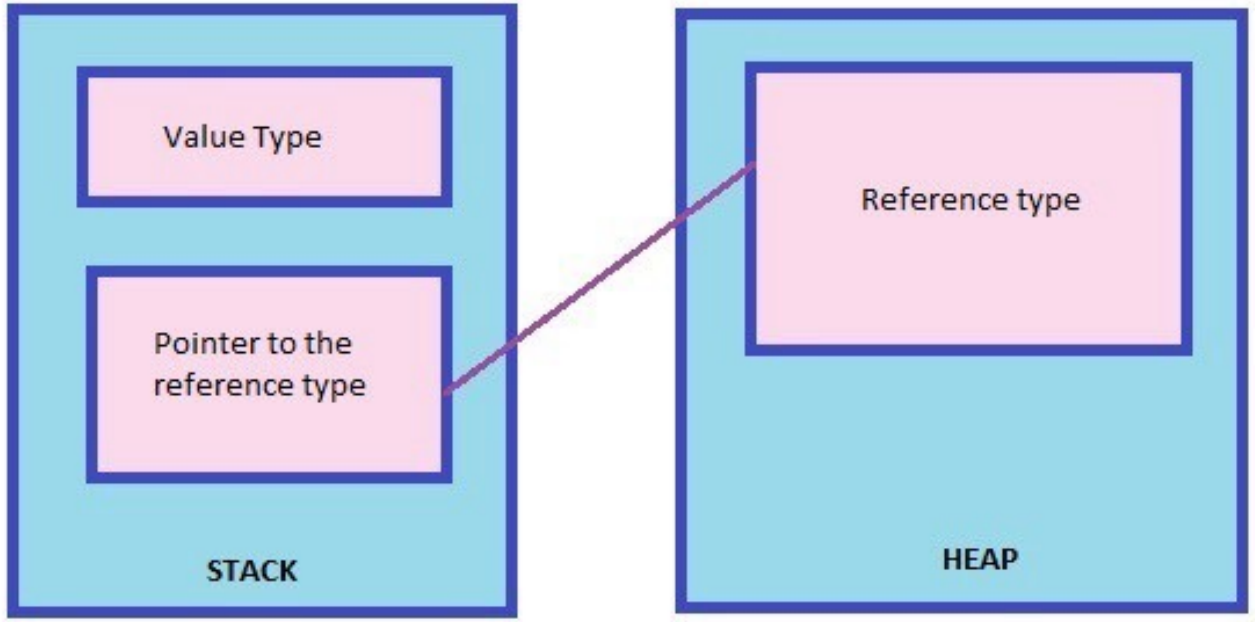

New Project
.NET Framework 4.5
Sort by:
Default
Search Installed Te
Installed
Templates
Visual C#
Visual C++
SQL Server
Visual F#
F# Application
Visual F#
F# Library
F# Tutorial
Visual F#
Type: Visual F#
A tutorial project providing a walkthrough of the F# language
Name:
Tutorial1
Location:
C:\source\fsharpintro
Create new solution
Solution name:
Add to source control
OK
DEMO

C# Fundamentals

Basic Concepts

C# Data Types

- reference types
- value types



C# Fundamentals

Basic Concepts

Reference Type Variables in C#

- object type
- string type

Value Type

Type	Represents	Range	Default Value
char	16-bit Unicode character	U +0000 to U +ffff	'\0'
byte	8-bit unsigned integer	0 to 255	0
bool	Boolean value	True or False	False
uint	32-bit unsigned integer type	0 to 4,294,967,295	0
int	32-bit signed integer type	-2,147,483,648 to 2,147,483,647	0
decimal	128-bit precise decimal values with 28-29 significant digits	(-7.9 x 1028 to 7.9 x 1028) / 100 to 28	0.0M
double	64-bit double-precision floating point type	(+/-)5.0 x 10-324 to (+/-)1.7 x 10308	0.0D
float	32-bit single-precision floating point type	-3.4 x 1038 to + 3.4 x 1038	0.0F
ushort	16-bit unsigned integer type	0 to 65,535	0
short	16-bit signed integer type	-32,768 to 32,767	0
long	64-bit signed integer type	-923,372,036,854,775,808 to 9,223,372,036,854,775,807	0L
sbyte	8-bit signed integer type	-128 to 127	0
ulong	64-bit unsigned integer type	0 to 18,446,744,073,709,551,615	0

C# Expression and Statements

Basic Concepts

Variables in C#

```
int count = 7;

char keyPressed = 'Q';

string title = "Weekly Report";

float fNumber = 3.19
```

C# Types and Operators

Basic Concepts

Variables in C#

```
int total = “777”;

string message = 7;
```

C# Types and Operators

Basic Concepts

Variables in C#

```
int total = “777”;
string message = 7;
```

```
int total = int.Parse(”777");

string message = 7.ToString();
```

C# Types and Operators

Basic Concepts

Variables in C#

```
int total = “777”;
string message = 7;

int total = int.Parse(”777");
string message = 7.ToString();
```

```
double preciseLength = 5.61;
int roundedLength = (int)preciseLength;
```

C# Types and Operators

Basic Concepts

Variables in C#

```
int total = “777”;
string message = 7;

int total = int.Parse(”777");
string message = 7.ToString();

double preciseLength = 5.61;
int roundedLength = (int)preciseLength;

decimal price = 9.95m;
char cr = '\u0013’;
int crUnicode = (int)cr;
```

C# Types and Operators

Basic Concepts

Category	Description
Primary	`x.y  x?.y  f(x)  a[x]  x++  x--  new  typeof  default  checked  unchecked  nameof`
Unary	`+  -  !  ~  ++x  --x  (T)x  await x`
Multiplicative	`*  /  %`
Additive	`+  -`
Shift	`<<  >>`
Relational and Type Testing	`<  >  <=  >=  is  as`
Equality	`==  !=`
Logical AND	`&`
Logical XOR	`^`
Logical OR	`|`
Conditional AND	`&&`
Conditional OR	`||`
Null Coalescing	`??`
Conditional	`?:`
Assignment	`=  *=  /=  %=  +=  -=  <<=  >>=  &=  ^=  |=  =>`

C# Types and Operators

Basic Concepts

String Formatting

```
string name = "Joe";

// Option 1
string helloViaConcatenation = "Hello, " + name + "!";
Console.WriteLine(helloViaConcatenation);

// Option 2
string helloViaStringFormat = string.Format("Hello,
{0}!", name);
Console.WriteLine(helloViaStringFormat);

// Option 3
Console.WriteLine("Hello, {0}!", name);

// Option 4
Console.WriteLine($"Hello, {name}!");
```

C# Types and Operators

Basic Concepts

Branching Statements (if/else)

```
string action = null;

if (priceGain <= 2m)
{
	action = "Sell";
}
else if (priceGain > 2m && priceGain <= 3m)
{
	action = "Do Nothing";
}
else
{
	action = "Sell";
}
```

C# Types and Operators

Basic Concepts

Branching Statements (switch)

```
string currentWeather = "rain";
string equipment = null;

switch (currentWeather)
{
        case "sunny":
                equipment = "sunglasses";
                break;
        case "rain":
                equipment = "umbrella";
                break;
        case "cold":
        default:
                equipment = "jacket";
                break;
}
```

C# Types and Operators

Basic Concepts

Collections and Arrays

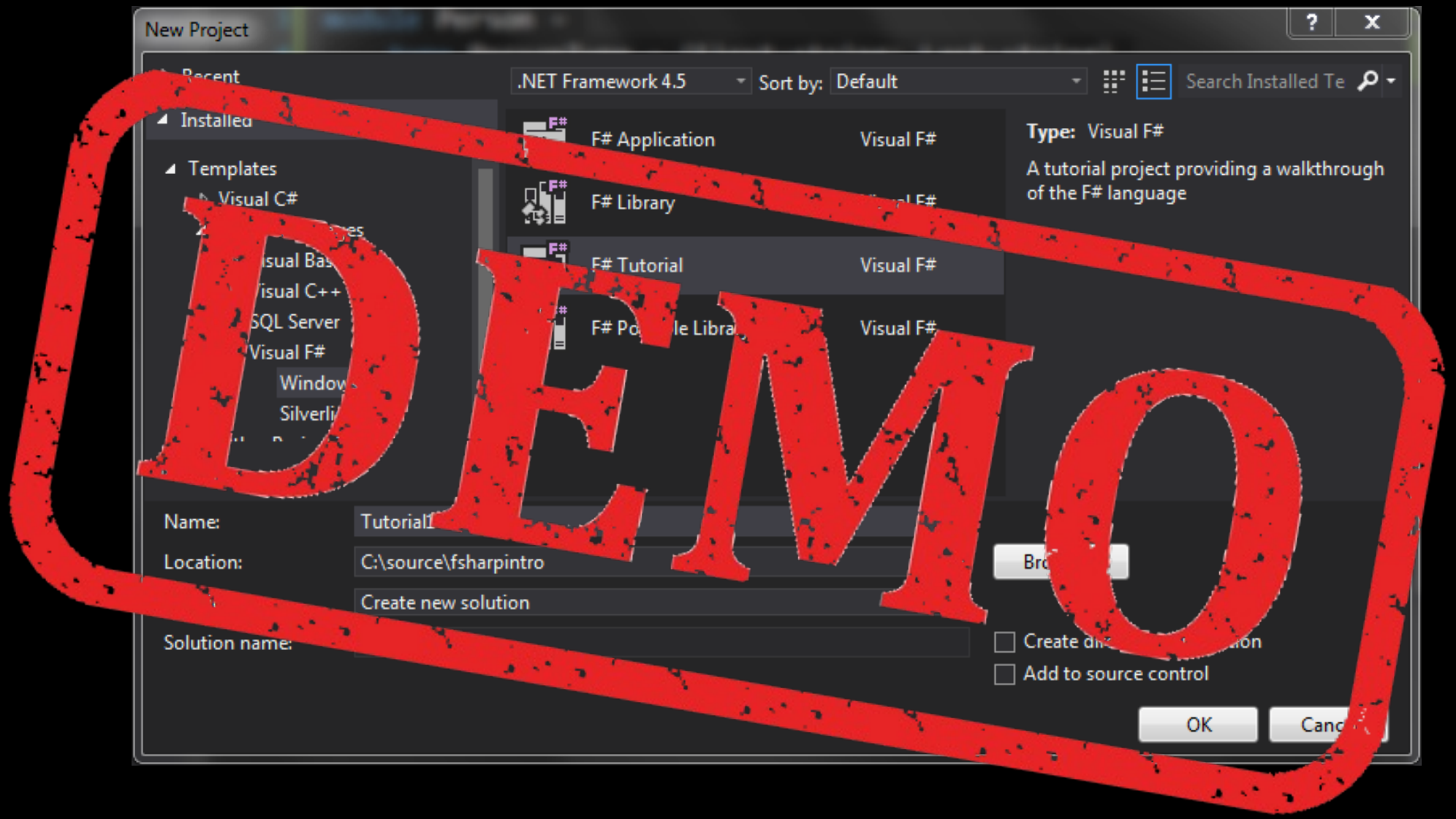
```
// Example 1
int[] oddNumbers = { 1, 3, 5 };
int firstOdd = oddNumbers[0];
int lastOdd = oddNumbers[2];

// Example 2
string[] names = new string[3];
names[1] = "Joe";

// Example 3
List<decimal> stockPrices = new
List<decimal>();
stockPrices.Add(56.23m);
stockPrices.Add(72.80m);
decimal secondStockPrice = stockPrices[1];
```

C# Types and Operators

Basic Concepts

Looping Statements

```
// Example 1
double[] temperatures = { 72.3, 73.8, 75.1,
74.9 };
for (int i = 0; i < temperatures.Length; i++)
      Console.WriteLine(i);


// Example 2
foreach (int temperature in temperatures)
      Console.WriteLine(temperature);

// Example 3
int tempCount = 0;
while (tempCount < temperatures.Length) {
      Console.WriteLine(tempCount);
      tempCount++;
}
```


New Project
.NET Framework 4.5
Sort by:
Default
Search Installed Te
Installed
Templates
Visual C#
Visual C++
SQL Server
Visual F#
F# Application
Visual F#
F# Library
F# Tutorial
Visual F#
Type: Visual F#
A tutorial project providing a walkthrough of the F# language
Name:
Tutorial1
Location:
C:\source\fsharpintro
Create new solution
Solution name:
Add to source control
OK
DEMO

C# Methods and Properties

Basic Concepts

```
sing System;
class Calculator1
{
    static void Main() {

        Console.Write("First Number: ");
        string firstNumberInput = Console.ReadLine();
        double firstNumber = double.Parse(firstNumberInput);

        Console.Write("Second Number: ");
        string secondNumberInput = Console.ReadLine();
        double secondNumber = double.Parse(secondNumberInput);

        double result = firstNumber + secondNumber;
        Console.WriteLine($"\n\tYour result is {result}.");
        Console.ReadKey();
    }
}
```

C# Methods and Properties

Basic Concepts

```
using System;

class Calculator2
{
    static void Main()
    {
        double firstNumber = GetFirstNumber();

        double secondNumber = GetSecondNumber();

        double result = AddNumbers(firstNumber, secondNumber);

        PrintResult(result);

        Console.ReadKey();
    }

    static double GetFirstNumber()
    {
        Console.Write("First Number: ");
        string firstNumberInput = Console.ReadLine();
        double firstNumber = double.Parse(firstNumberInput);
        return firstNumber;
    }

    static double GetSecondNumber()
    {
        Console.Write("Second Number: ");
        string secondNumberInput = Console.ReadLine();
        double secondNumber = double.Parse(secondNumberInput);
        return secondNumber;
    }

    static double AddNumbers(double firstNumber, double secondNumber)
    {
        return firstNumber + secondNumber;
    }

    static void PrintResult(double result)
    {
        Console.WriteLine($"\nYour result is {result}.");
    }
}
```

C# Methods and Properties

Basic Concepts

```
static double GetFirstNumber()
{
        Console.Write("First Number: ");
        string firstNumberInput = Console.ReadLine();
        double firstNumber = double.Parse(firstNumberInput);
        return firstNumber;
}
```

```
static double AddNumbers(double firstNumber, double secondNumbe
{
        return firstNumber + secondNumber;
}
```

```
static void PrintResult(double result)
{
        Console.WriteLine($"\nYour result is {result}.");
}
```

C# Methods and Properties

Basic Concepts

Simplifying the Code with Methods

```
class Calculator3
{
    static void Main()
    {
        double firstNumber = GetNumber("First");
        double secondNumber = GetNumber("Second");

        double result = AddNumbers(firstNumber, secondNumber);

        PrintResult(result);

        Console.ReadKey();
    }

    static double GetNumber(string whichNumber)
    {
        Console.Write($"{whichNumber} Number: ");
        string numberInput = Console.ReadLine();
        double number = double.Parse(numberInput);
        return number;
    }

    static double AddNumbers(double firstNumber, double secondNumber)
    {
        return firstNumber + secondNumber;
    }

    static void PrintResult(double result)
    {
        Console.WriteLine($"\nYour result is {result}.");
    }
}
```

C# Methods and Properties

Basic Concepts

Adding Properties

```
public class Calculator
{
        double[] numbers = new double[2];
        public double First {
                get { return numbers[0]; }
        }
        public double Second {
                get { return numbers[1]; }
        }
        double result;
        public double Result {
                get { return result; }
                set { result = value; }
        }
        public void GetNumber(string whichNumber) {
                Console.Write($"{whichNumber} Number: ");
                string numberInput = Console.ReadLine();
                double number = double.Parse(numberInput);
                if (whichNumber == "First")
                        numbers[0] = number;
                else
                        numbers[1] = number;
        }
        public void AddNumbers() {
                Result = First + Second;
        }
        public void PrintResult() {
                Console.WriteLine($"\nYour result is {result}.");
        }
```

C# Methods and Properties

Basic Concepts

Adding Properties

```
double result;
public double Result
{
        get { return result; }
        set { result = value; }
}


public double Result
{
        get;
        set;
}
```

Code Katas

Code katas

What is a code kata? Code katas are nothing more than repeatable exercises. Generally, these exercises are meant to take no more than 30 minutes to complete. Most code katas are directed at a specific classification of a problem to solve.

GITHUB

Exception Handling

```
static void HandleNullReference() {
    Program prog = null;
    try
    {
       Console.WriteLine(prog.ToString());
    }
    catch (NullReferenceException ex)
    {
       Console.WriteLine(ex.Message);
     }
}
```

Exception Handling

```
static void HandleUncaughtException()
{
    Program prog = null;
    try
    {
       Console.WriteLine(prog.ToString());
     }
     catch (ArgumentNullException ex) {
            Console.WriteLine("From ArgumentNullException: " +
ex.Message);
     }
     catch (ArgumentException ex) {
           Console.WriteLine("From ArgumentException: " + ex.Message);
     }
     catch (Exception ex) {
           Console.WriteLine("From Exception: " + ex.Message);
     }
     finally {
        Console.WriteLine("Finally always executes.");
     }
}
```

Object-Oriented Code in C#

Inheritance

- **Calculator**
  - **ScientificCalculator**
  - **ProgrammerCalculator**

In C#, you would express this relationship as follows.

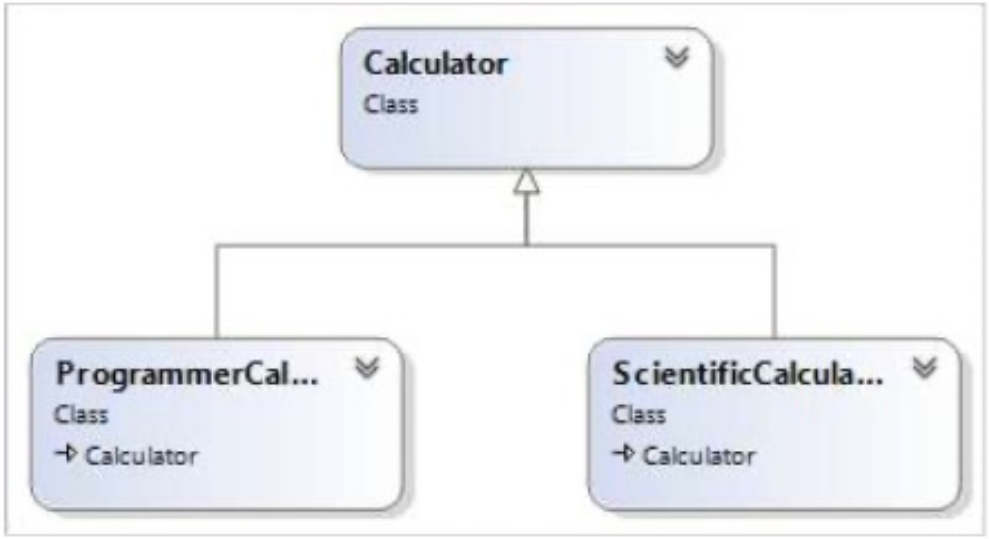
```
public class Calculator { }
public class ScientificCalculator : Calculator { }
public class ProgrammerCalculator : Calculator { }
```

Inheritance

- **Calculator**
  - **ScientificCalculator**
  - **ProgrammerCalculator**

In C#, you would express this relationship as follows.

```
public class Calculator { }
public class ScientificCalculator : Calculator { }
public class ProgrammerCalculator : Calculator { }
```



Inheritance

```
using System;

public class Calculator
{
    public double Add(double num1, double num2)
    {
        return num1 + num2;
    }
}

public class ScientificCalculator : Calculator
{
    public double Power(double num, double power)
    {
        return Math.Pow(num, power);
    }
}

public class ProgrammerCalculator : Calculator
{
    public int Or(int num1, int num2)
    {
        return num1 | num2;
    }
}
```

Inheritance

```
using System;

public class Program
{
    public static void Main()
    {
        ScientificCalculator sciCalc = new ScientificCalculator();
        double powerResult = sciCalc.Power(2, 5);
        Console.WriteLine($"Scientific Calculator 2**5: {powerResult}");
        double sciSum = sciCalc.Add(3, 3);
        Console.WriteLine($"Scientific Calculator 3 + 3: {sciSum}");

        ProgrammerCalculator prgCalc = new ProgrammerCalculator();
        double orResult = prgCalc.Or(5, 10);
        Console.WriteLine($"Programmer Calculator 5 | 10: {orResult}");

        double prgSum = prgCalc.Add(3, 3);
        Console.WriteLine($"Programmer Calculator 3 + 3: {prgSum}");

        Console.ReadKey();
    }
}
```

Access Modifiers and Encapsulation

```
internal class Calculator
{
    private double Add(double num1, double num2) {
        return num1 + num2;
    }
}

internal class ScientificCalculator : Calculator
{
    public double Power(double num, double power) {
        return Math.Pow(num, power);
    }
}

public class ProgrammerCalculator : Calculator
{
    protected int Or(int num1, int num2) {
        return num1 | num2;
    }
}
```

Class vs. Struct

```
public struct Complex
{
    public Complex(double real, double imaginary)
    {
        Real = real;
        Imaginary = imaginary;
    }

    public double Real { get; set; }

    public double Imaginary { get; set; }

    public static Complex operator +(Complex complex1, Complex complex2)
    {
        Complex complexSum = new Complex();
        complexSum.Real = complex1.Real + complex2.Real;
        complexSum.Imaginary = complex1.Imaginary + complex2.Imaginary;
        return complexSum;
    }
    public static implicit operator Complex(double dbl)
    {
        Complex cmplx = new Complex();
        cmplx.Real = dbl;
        return cmplx;
    }
    // This is not a safe operation.
    public static explicit operator double(Complex cmplx)
    {
        return cmplx.Real;
    }
}
```

Class vs. Struct

```
public struct Complex
{
    public Complex(double real, double imaginary)
    {
        Real = real;
        Imaginary = imaginary;
    }

    public double Real { get; set; }

    public double Imaginary { get; set; }

    public static Complex operator +(Complex complex1, Complex complex2)
    {
        Complex complexSum = new Complex();
        complexSum.Real = complex1.Real + complex2.Real;
        complexSum.Imaginary = complex1.Imaginary + complex2.Imaginary;
        return complexSum;
    }
    public static implicit operator Complex(double dbl)
    {
        Complex cmplx = new Complex();
        cmplx.Real = dbl;
        return cmplx;
    }
    // This is not a safe operation.
    public static explicit operator double(Complex cmplx)
    {
        return cmplx.Real;
    }
}
```

Class vs. Struct

```
using System;

class Program
{
    static void Main()
    {
        Complex complex1 = new Complex();
        complex1.Real = 3;
        complex1.Imaginary = 1;

        Complex complex2 = new Complex(7, 5);

        Complex complexSum = complex1 + complex2;

        Console.WriteLine(
            $"Complex sum - Real: {complexSum.Real}, " +
            $"Imaginary: {complexSum.Imaginary}");

        Complex complex3 = 9;

        double realPart = (double)complex3;

        Console.ReadKey();
    }
}
```

Boxing vs Unboxing

```
ArrayList intCollection = new ArrayList();
intCollection.Add(7);
int number = (int)intCollection[0];
```

Make the code predictable

```
// create two customers
var cust1 = new Customer(99, "J Smith");
var cust2 = new Customer(99, "J Smith");


// true or false?
cust1 == cust2;
```

You cannot tell!

Enums

```
public enum MathOperation
{
    Add,
    Subtract,
    Multiply,
    Divide
}
```

Enums

```
static void Main()
{
    string[] possibleOperations = Enum.GetNames(typeof(MathOperation));

    Console.Write($"Please select ({string.Join(", ", possibleOperations)}): ");

    string operationString = Console.ReadLine();

    MathOperation selectedOperation;

    if (!Enum.TryParse<MathOperation>(operationString, out selectedOperation))
        selectedOperation = MathOperation.Add;

    switch (selectedOperation)
    {
        case MathOperation.Add:
            Console.WriteLine($"You selected {nameof(Add)}");
            break;
        case MathOperation.Subtract:
            Console.WriteLine($"You selected {nameof(Subtract)}");
            break;
        case MathOperation.Multiply:
            Console.WriteLine($"You selected {nameof(Multiply)}");
            break;
        case MathOperation.Divide:
            Console.WriteLine($"You selected {nameof(Divide)}");
            break;
    }
```

Polymorphism

```
using System;

public class Calculator
{
    public virtual double Add(double num1, double num2)
    {
        Console.WriteLine("Calculator Add called.");
        return num1 + num2;
    }
}

public class ProgrammerCalculator : Calculator
{
    public override double Add(double num1, double num2)
    {
        Console.WriteLine("ProgrammerCalculator Add called.");
        return MyMathLib.Add(num1, num2);

    }
}

public class MyMathLib
{
    public static double Add(double num1, double num2)
    {
        return num1 + num2;
    }
}

public class ScientificCalculator : Calculator
{
    public override double Add(double num1, double num2)
    {
        Console.WriteLine("ScientificCalculator Add called.");
        return base.Add(num1, num2);
    }
}
```

Polymorphism

```
using System;

public class Program
{
    public static void Main()
    {
        Calculator sciCalc = new ScientificCalculator();
        double sciCalcResult = sciCalc.Add(2, 5);
        Console.WriteLine($"Scientific Calculator 2 + 5: {sciCalcResult}");

        Calculator prgCalc = new ProgrammerCalculator();
        double prgCalcResult = prgCalc.Add(5, 10);
        Console.WriteLine($"Programmer Calculator 5 + 10: {prgCalcResult}");

        Console.ReadKey();
    }
}
```

Output:

- ScientificCalculator Add called.
- Calculator Add called.
- Scientific Calculator 2 + 5: 7
- ProgrammerCalculator Add called.
- Programmer Calculator 5 + 10: 15

Abstract Classes

```
public abstract class Calculator
{
    public abstract double Add(double num1, double num2);
}
```

Interfaces

```
public interface ICalculator
{
    double Add(double num1, double num2);
}
```

Interfaces

```
public interface ICalculator
{
    double Add(double num1, double num2);
}
```

```
public class ScientificCalculator : ICalculator
{
    public double Add(double num1, double num2)
    {
        return num1 + num2;
    }
}

public class ProgrammerCalculator : ICalculator
{
    public double Add(double num1, double num2)
    {
        return MyMathLib.Add(num1, num2);
    }
}

public class MyMathLib
{
    public static double Add(double num1, double num2)
    {
        return num1 + num2;
    }
}
```

Interfaces

```
public class Program
{
    public static void Main()
    {
        ICalculator sciCalc = new ScientificCalculator();
        double sciCalcResult = sciCalc.Add(2, 5);
        Console.WriteLine($"Scientific Calculator 2 + 5: {sciCalcResult}");

        ICalculator prgCalc = new ProgrammerCalculator();
        double prgCalcResult = prgCalc.Add(5, 10);
        Console.WriteLine($"Programmer Calculator 5 + 10: {prgCalcResult}");


          Console.ReadKey();
    }
}
```

Interfaces

```
public interface ICalculator { }
public interface IMath { }

public class ScientificCalculator : ICalculator, IMath
{
    public double Add(double num1, double num2)
    {
        return num1 + num2;
    }
}
```

Object Lifetime

```
using System;

public class Calculator
{
    static double pi = Math.PI;

    double startAngle = 0;

    public DateTime Created { get; } = DateTime.Now;

    static Calculator()
    {
        Console.WriteLine("static Calculator()");
    }

    public Calculator()
    {
        Console.WriteLine("public Calculator()");
    }

    public Calculator(int val)
    {
        Console.WriteLine("public Calculator(int)");
    }
}
```

Object Lifetime

```
using System;

public class ScientificCalculator : Calculator
{
    static double pi = Math.PI;
    double startAngle = 0;

    static ScientificCalculator()
    {
        Console.WriteLine("static ScientificCalculator()");
    }

    public ScientificCalculator() : this(0)
    {
        Console.WriteLine("public ScientificCalculator()");
    }

    public ScientificCalculator(int val)
    {
        Console.WriteLine("public ScientificCalculator(int)");
    }

    public ScientificCalculator(int val, string word) : base(val)
    {
        Console.WriteLine("public ScientificCalculator(int, string)");
    }


    public double EndAngle { get; set; }
}
```

Object Lifetime

```
using System;

class Program
{
    static void Main()
    {
        var calc1 = new ScientificCalculator();

        var calc2 = new ScientificCalculator(0, "x")
        {
            EndAngle = 360
        };

        Console.ReadKey();
    }
}
```

And here is the program's output:

- static ScientificCalculator()
- static Calculator()
- public Calculator()
- public ScientificCalculator(int)
- public ScientificCalculator()
- public Calculator(int)
- public ScientificCalculator(int, string)

Object Lifetime

IDisposable

```
public class Calculator : IDisposable
{
    static Calculator()
    {
        // Initialize log file stream.
    }

    #region IDisposable Support
    private bool disposedValue = false; // To detect redundant calls.

    protected virtual void Dispose(bool disposing)
    {
        if (!disposedValue)
        {
            if (disposing)
            {
                // TODO: dispose managed state (managed objects).
                // Close log file stream.
            }

            // TODO: free unmanaged resources (unmanaged objects) and override a
finalizer below.
            // TODO: set large fields to null.

            disposedValue = true;
        }
    }

    // TODO: override a finalizer only if Dispose(bool disposing) above has code to
free unmanaged resources.
    // ~Calculator() {
    //   // Do not change this code. Put cleanup code in Dispose(bool disposing) above.
    //   Dispose(false);
    // }

    // This code added to correctly implement the disposable pattern.
    public void Dispose()
    {
        // Do not change this code. Put cleanup code in Dispose(bool disposing) above.
        Dispose(true);
        // TODO: uncomment the following line if the finalizer is overridden above.
        // GC.SuppressFinalize(this);
    }
    #endregion
}
```

Object Lifetime

IDisposable

```
ScientificCalculator calc3 = null;
try
{
    calc3 = new ScientificCalculator();
    // Do stuff.
}
finally
{

    if (calc3 != null)
        calc3.Dispose();
}
```

Object Lifetime

IDisposable

```
ScientificCalculator calc3 = null;
try
{
    calc3 = new ScientificCalculator();
    // Do stuff.
}
finally
{
    if (calc3 != null)
        calc3.Dispose();
}
```

```
using (var calc4 = new ScientificCalculator())
{
    // Do stuff.
}
```

S.O.L.I.D.

- Single Responsibility Principle (SRP)
- Open Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)

Single Responsibility Principle (SRP)

```
public interface IDatabase
{
    void Connect(string connectionString);
    void Close();
    object GetData();
    void SendData(object data);
}
```

Single Responsibility Principle (SRP)

```
public interface IConnectionManager
{
    void Connect(string connectionString);
    void Close();
}

public interface IDataManager
{
    object GetData(IConnectionManager connManager);
    void SendData(IConnectionManager connManager, object data);
}
```

Open Closed Principle (OCP)

```
public interface IDatabase
{
    void Connect(string connectionString);
    void Close();
    object GetData();
    void SendData(object data);
}
```

```
public interface IConnectionManager
{
    void Connect(string connectionString);
    void Close();
}

public interface IDataManager
{
    object GetData(IConnectionManager connManager);
    void SendData(IConnectionManager connManager, object data);
}
```

Liskov Substitution Principle (LSP)

```
bool TestConnection(IConnectionManager connMngr)
{
    if (connMngr is SqlServerConnectionManager)
    {
        // Do something...
    }
    else if (connMngr is OracleConnectionManager)
    {
        // Do something else...
    }
    else
    {
        // ...
    }
}
```

Interface Segregation Principle (ISP)

```
public interface IDatabase
{
    void Connect(string connectionString);
    void Close();
    object GetData();
    void SendData(object data);
}
```

Interface Segregation Principle (ISP)

```
public class ConnectionManager
{
    public void Connect(string connectionString)
    {
        // ...
    }

    public void Close()
    {
        // ...
    }
}
```

```
public class DataManager : ConnectionManager
{
    public virtual object GetData()
    {
        // ...
    }

    public virtual void SendData(object data)
    {
        // ...
    }
}

public class DatabaseManager : DataManager
{
    // Needs ConnectionManager...
}
```

Dependency Inversion Principle (DIP)

```
class Program
{
     //...

    bool TestConnection(SqlConnectionManager connMngr)
    {
        // ...
    }

}

public interface IConnectionManager
{
    void Close();
    void Connect(string connectionString);
}

public class SqlConnectionManager : IConnectionManager
{
    // ...
}
```

Dependency Inversion Principle (DIP)

```
class Program
{
    //...

    bool TestConnection(SqlConnectionManager connMngr)
    {
        // ...
    }

}

public interface IConnectionManager
{
    void Close();
    void Connect(string connectionString);
}

public class SqlConnectionManager : IConnectionManager
{
    // ...
}
```

```
bool TestConnection(IConnectionManager connMngr)
{
    // ...
}
```

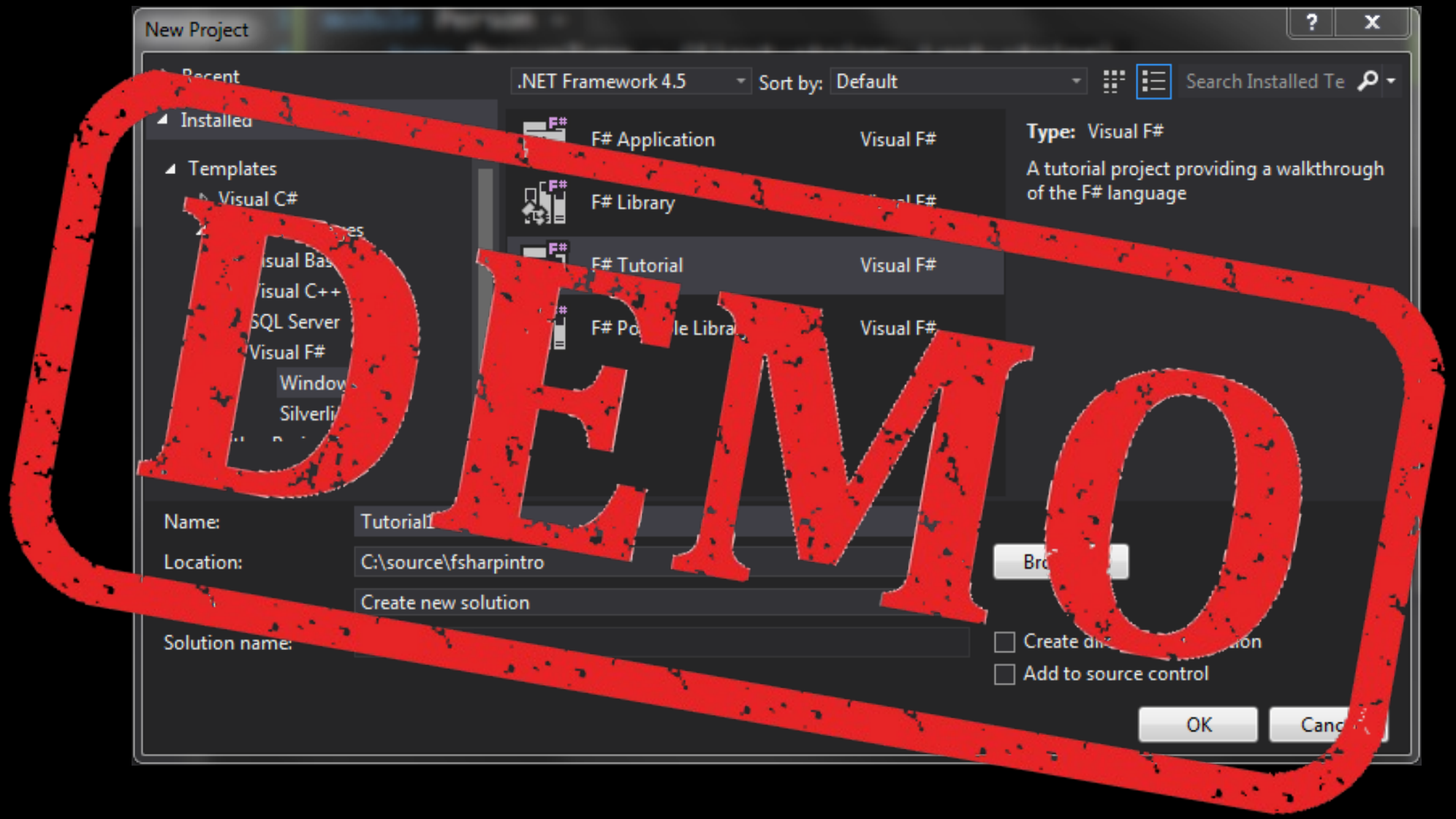
Dependency Injection (DI)

```
public interface IConnectionManager
{
    void Close();
    void Connect();
}

public class SqlConnectionManager : IConnectionManager
{
    public void Close()
    {
        Console.WriteLine("Closed SQL Server connection...");
    }

    public void Connect()
    {
        Console.WriteLine("Connected to SQL Server!");
    }
}

public class OracleConnectionManager : IConnectionManager
{
    public void Close()
    {
        Console.WriteLine("Closed Oracle connection...");
    }

    public void Connect()
    {
        Console.WriteLine("Connected to Oracle!");
    }
}
```

Dependency Injection (DI)

```
class Program
{
    static void Main(string[] args)
    {
        // Create a new DI Container...
        IUnityContainer container = new UnityContainer();
        // ...And register a type!
        container.RegisterType<IConnectionManager, SqlConnectionManager>();
    }
}
```

```
IConnectionManager connMngr = container.Resolve<IConnectionManager>();
bool success = TestConnection(connMngr);
// Do something with the result...
```


New Project
.NET Framework 4.5
Sort by:
Default
Search Installed Te
Installed
Templates
Visual C#
Visual C++
SQL Server
Visual F#
F# Application
Visual F#
F# Library
F# Tutorial
Visual F#
Type: Visual F#
A tutorial project providing a walkthrough of the F# language
Name:
Tutorial1
Location:
C:\source\fsharpintro
Create new solution
Solution name:
Add to source control
OK
DEMO

Resources

https://dotnet.microsoft.com/en-us/download

https://dotnet.microsoft.com/en-us/learn/csharp

https://code.visualstudio.com

https://visualstudio.microsoft.com/vs/community/

The tools we use have a profound (and devious!) influence on our thinking habits, and, therefore, on our thinking abilities.

-- Edsger Dijkstra


That's all Folks!