

C# Fundamentals

Viet Nguyen

Mar-2025



Back to basic

Viet Nguyen

Mar-2025



Learning Objectives

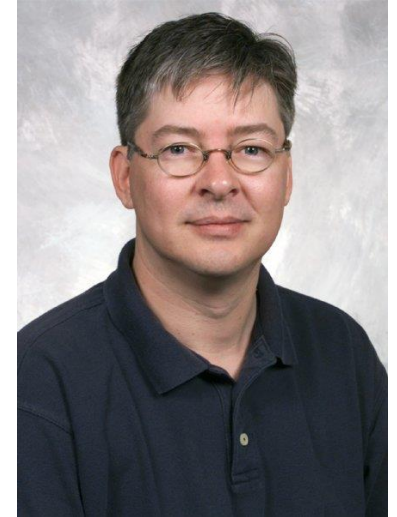
- Why we use C#?
- Fundamentals of the C# language
 - Types, program structure, statements, operators
- Be able to begin writing and debugging C# programs
- Be able to write individual C# methods

Agenda

- Why C#?
- C# and .NET Framework
- Data types and variables
- Statements – loop
- Logic and conditions
- Methods

Programming Language C#

C# (pronounced as C-sharp) is an object-oriented programming language. It was developed in 1998-2001 as a language for developing applications for the Microsoft .NET by a group of engineers led by Anders Hejlsberg.

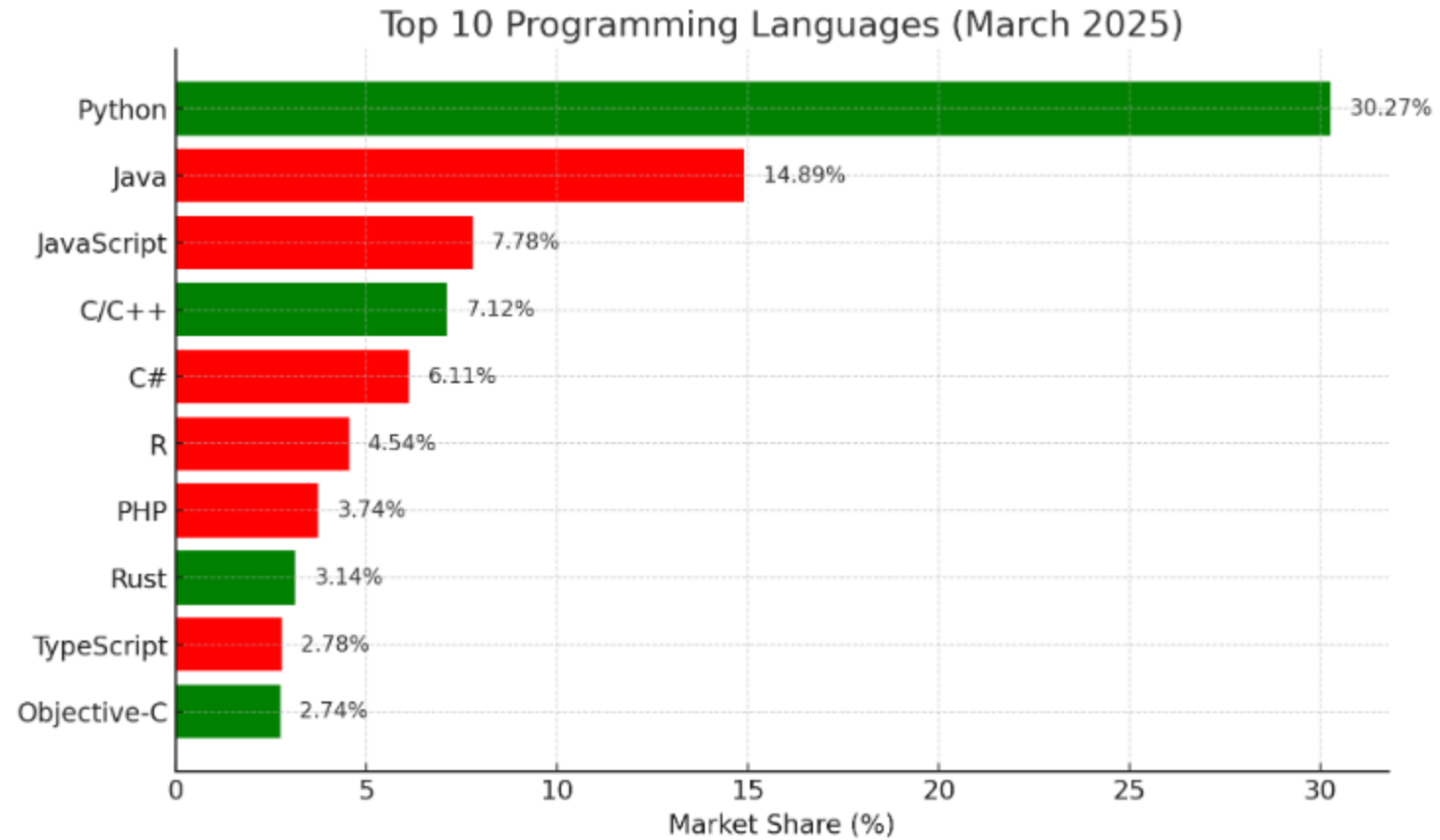


Anders Hejlsberg

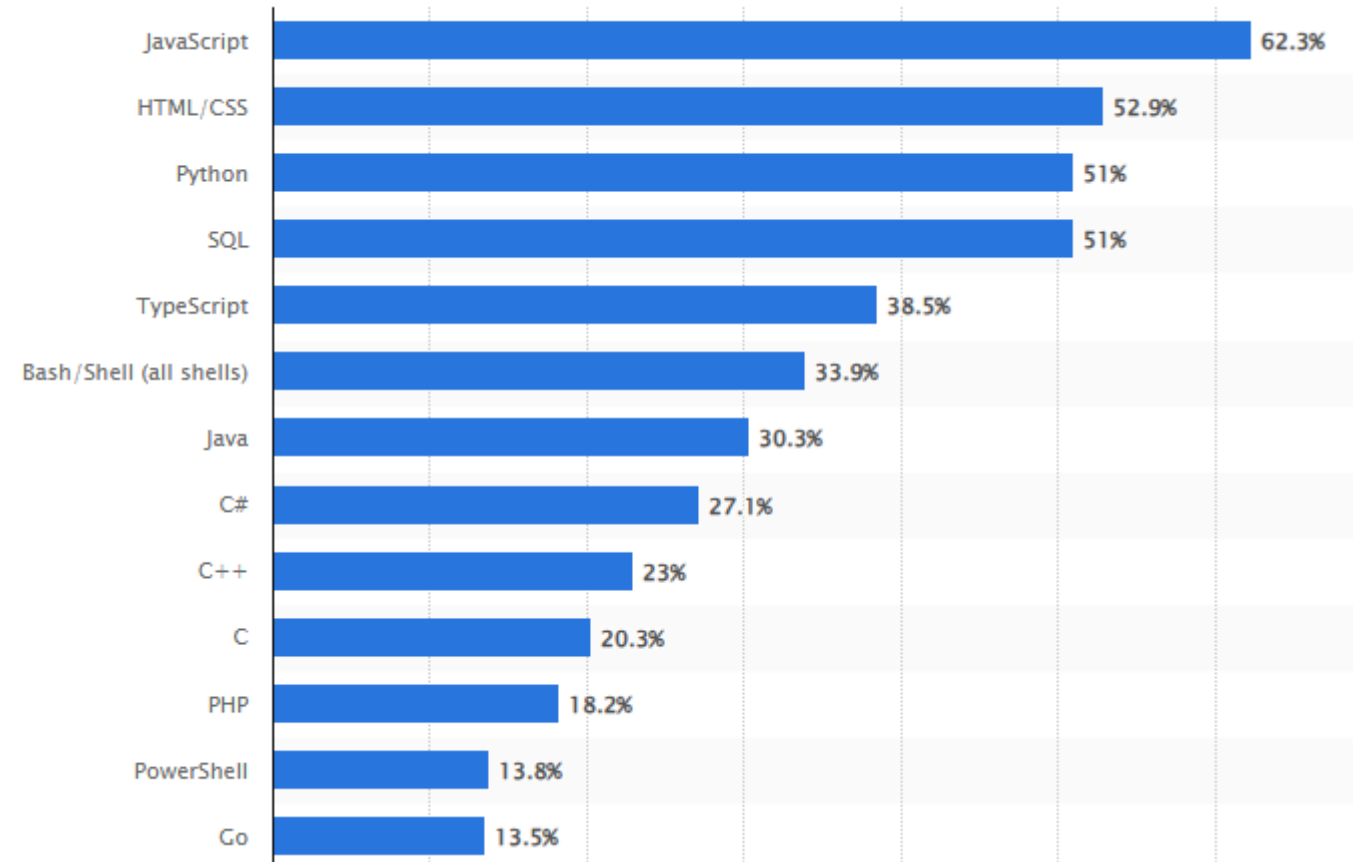


- ▶ C# was inspired by C, C++ and Java, however C# creators took best parts of those languages and innovated further by adding new concepts.
- ▶ The name “C sharp” was inspired by musical notation
- ▶ C# was designed with simplicity and readability in mind.

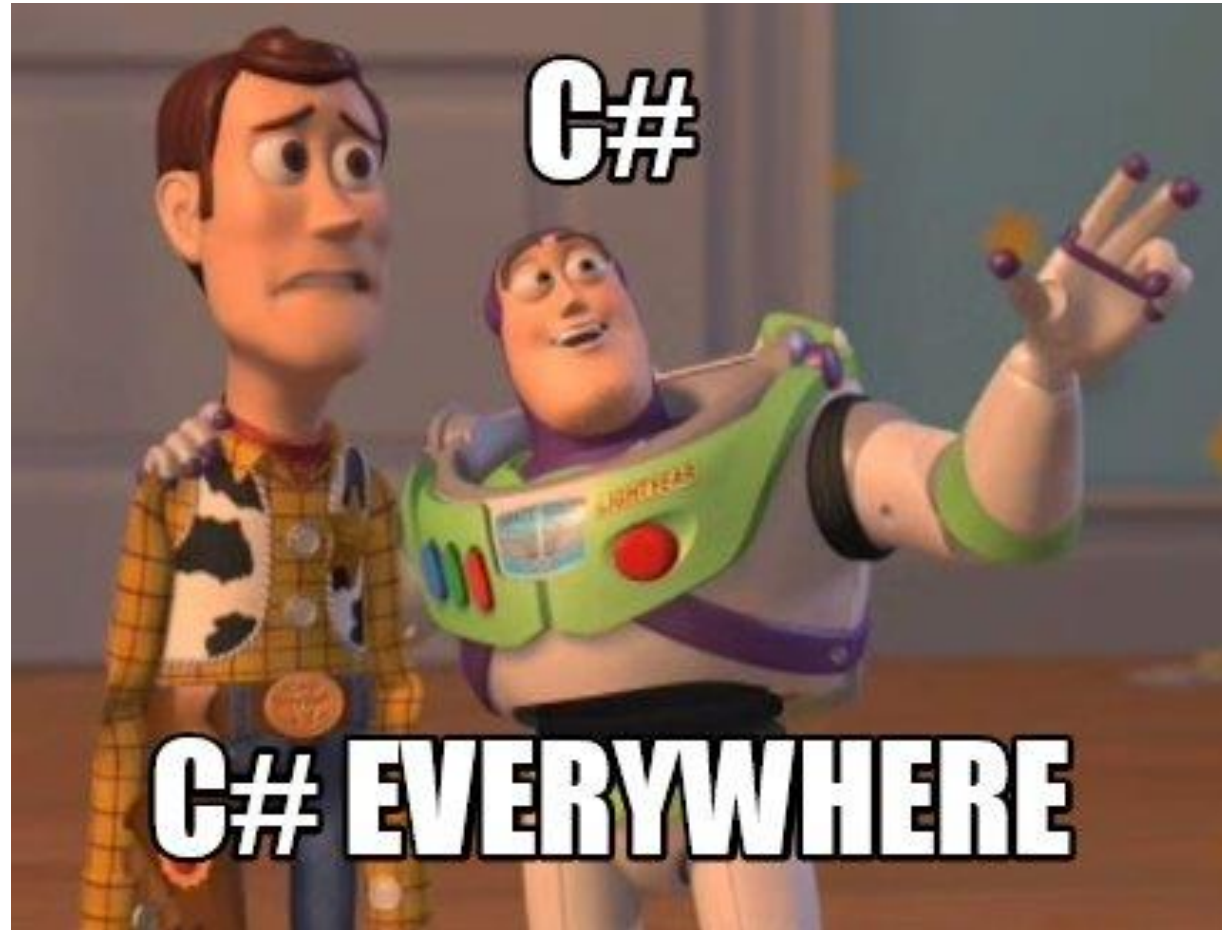
Top 10 Programming Language



Most used programming languages among dev 2024



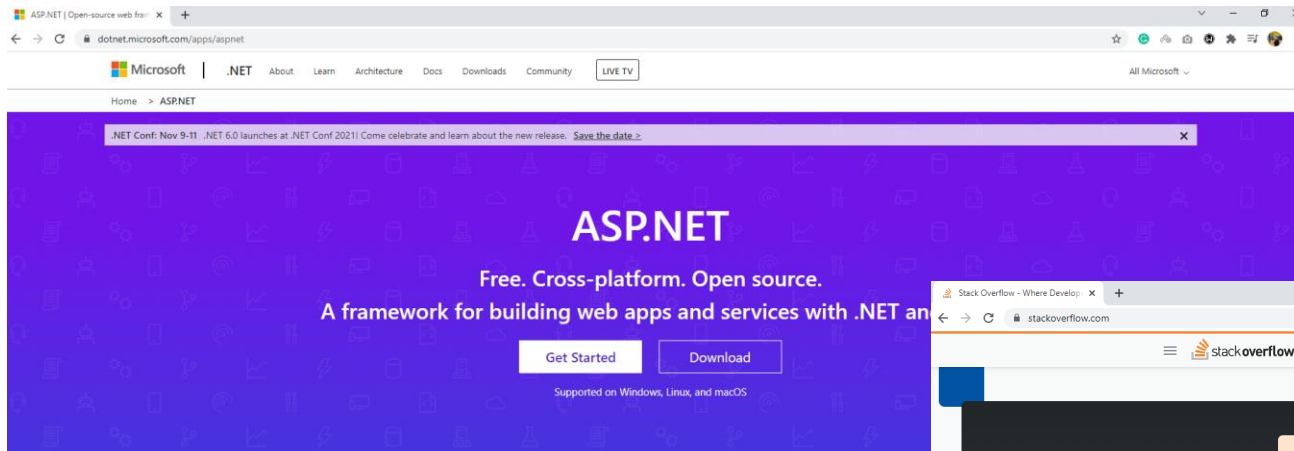
Why C#?



Why C#?

- Simple, readable and easy to use
- Multi-paradigm programming language
- Flexible
- Cross-platform
- Mature and Popular
- Well documented

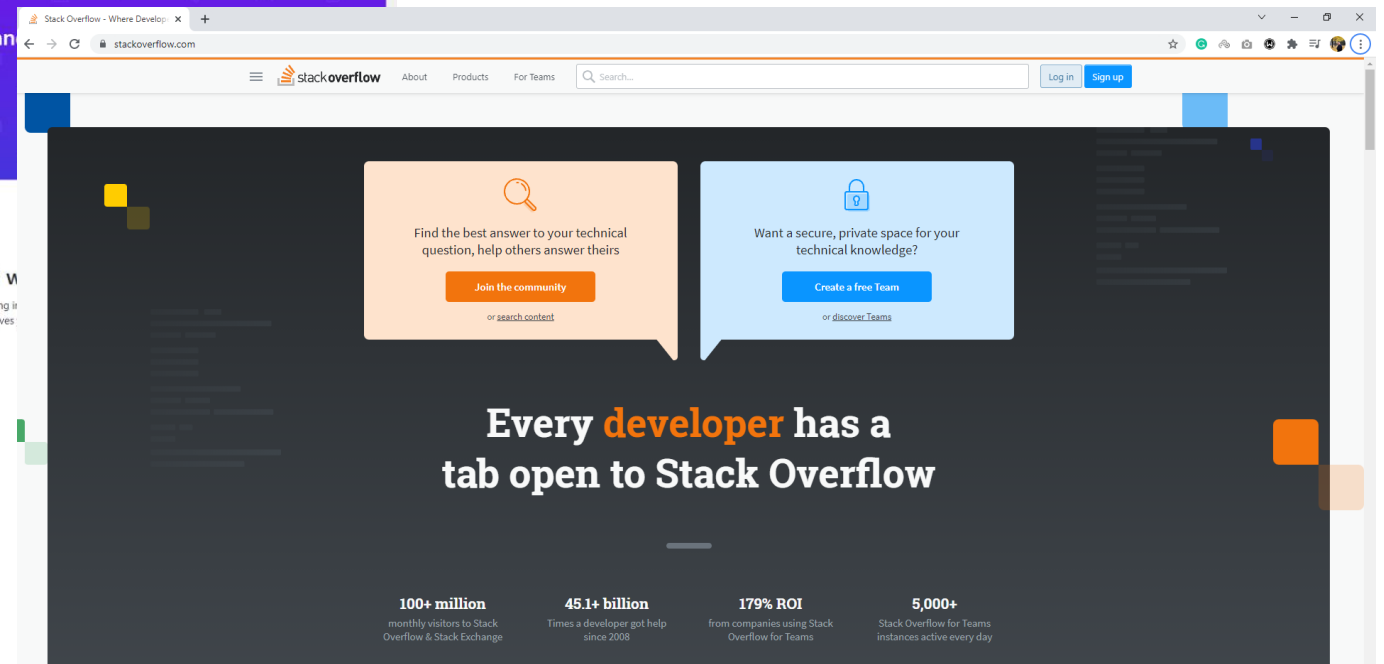
Web Application



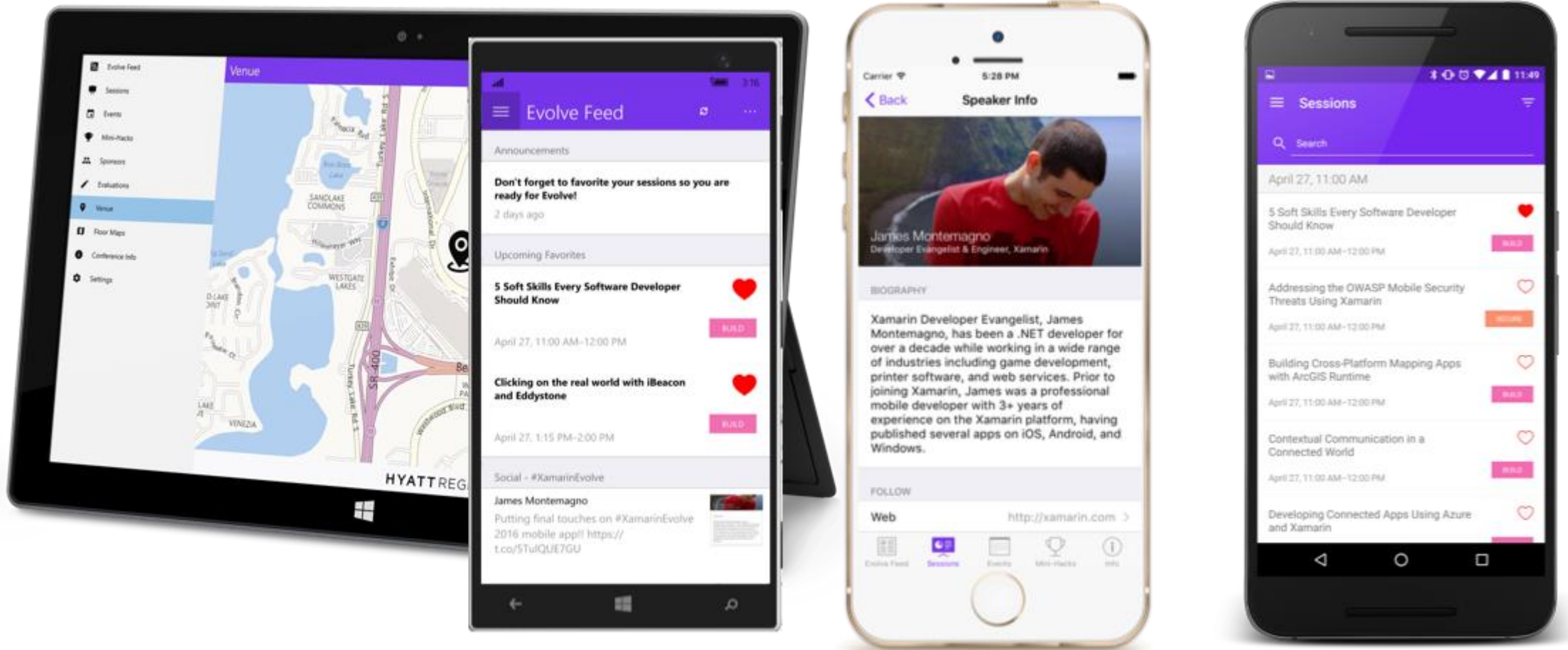
Interactive web UI with Blazor

Blazor is a feature of ASP.NET for building interactive web UIs using C# instead of JavaScript. Blazor gives you the ability to run your code in the browser on WebAssembly.

[Learn about Blazor](#)



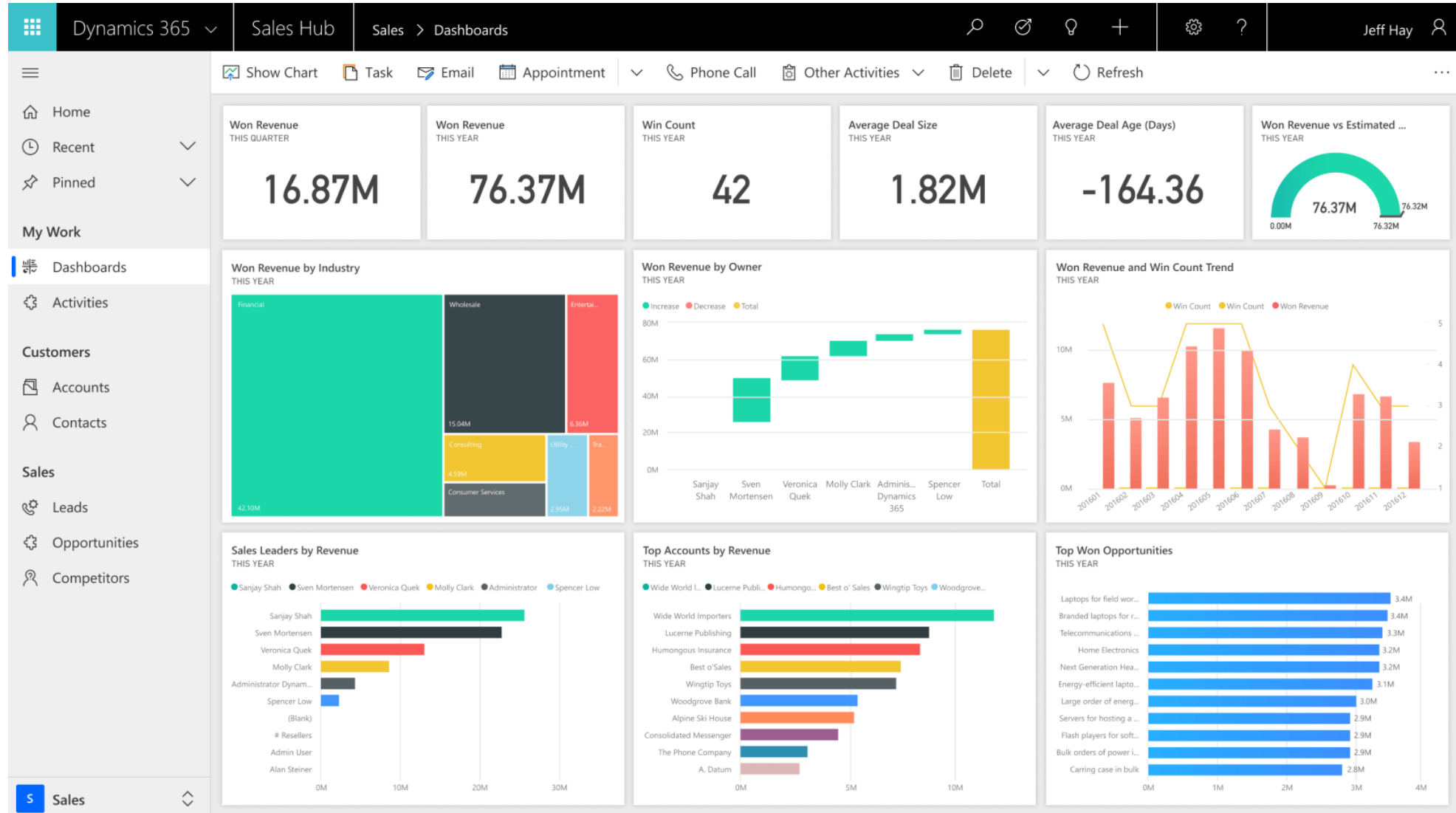
Mobile Application



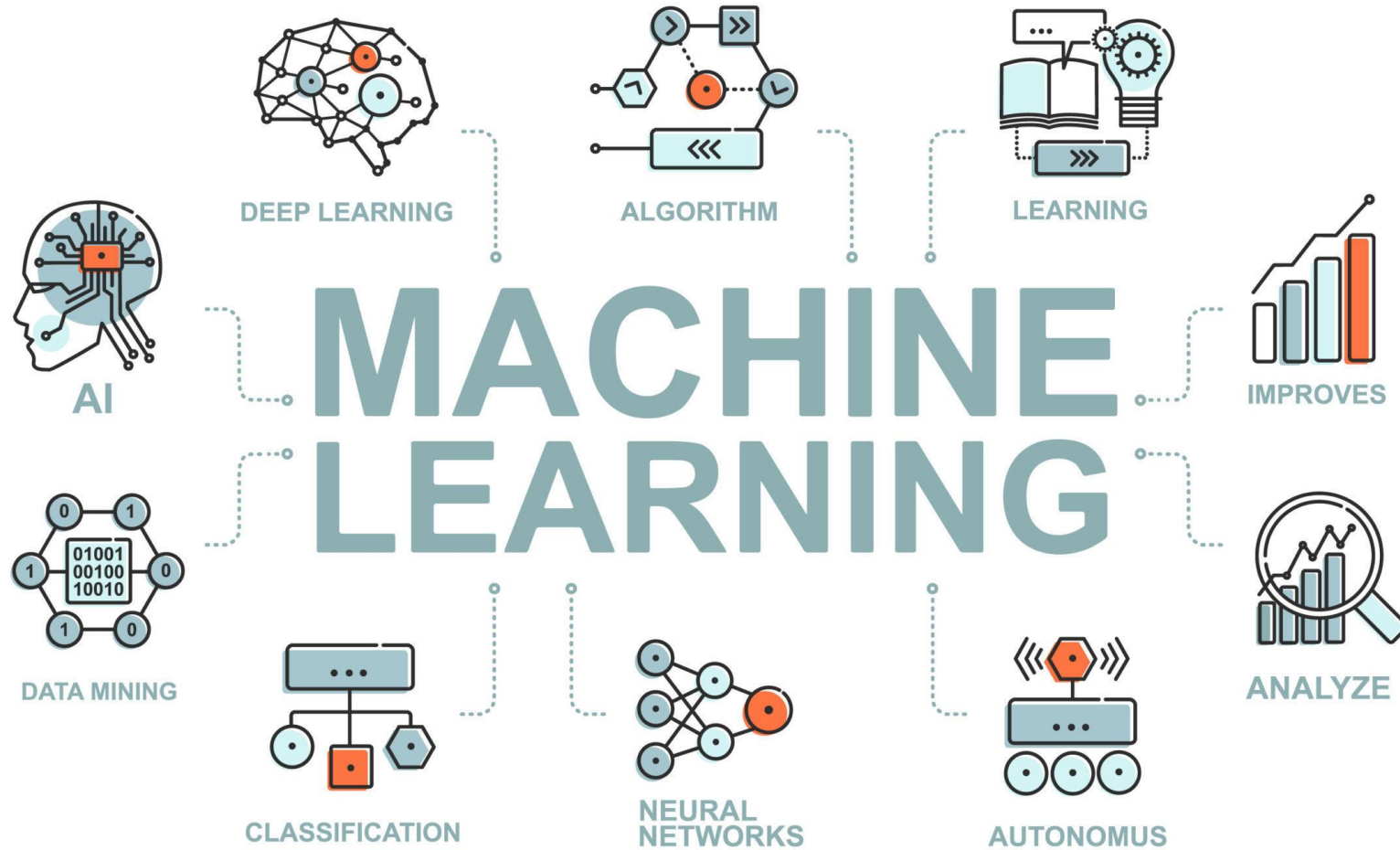
Game Development



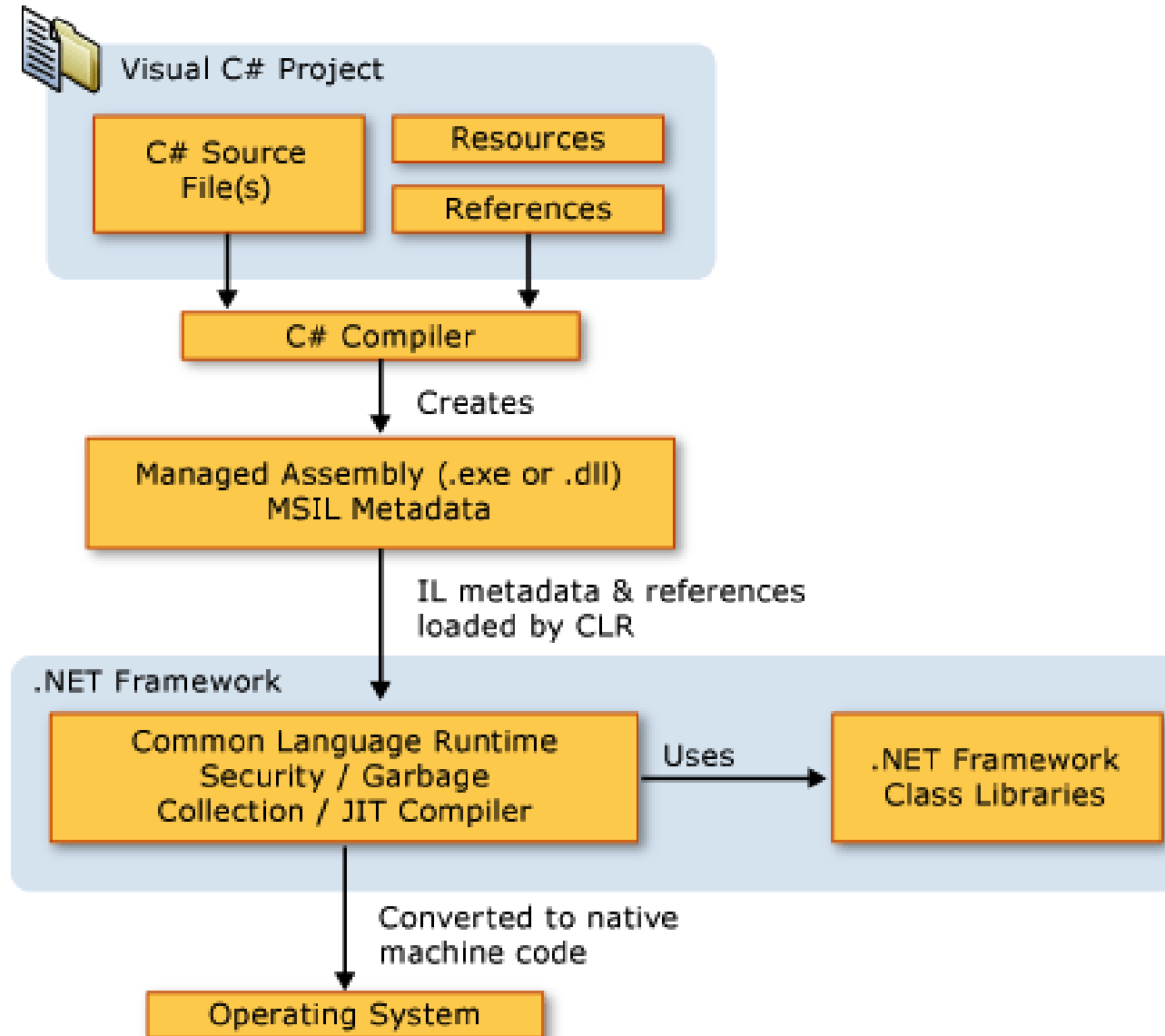
Business Applications



ML.NET



C# and .NET Framework



Data Types - Overview

- A C# program is a collection of types
 - Classes, structs, enums, interfaces, delegates
- C# provides a set of predefined types
 - E.g. `int`, `byte`, `char`, `string`, `object`, ...
- You can create your own types
- All data and code is defined within a type
 - No global variables, no global functions

Data Types - Overview

- A data type contain:
 - Data members (store data)
 - Fields
 - Properties
 - Constants
 - Readonly fields
 - Function members (define behavior)
 - Methods
 - Constructors
 - Destructors
 - Operators
 - Other members (control behavior and state)
 - Events
 - Indexers

Data Types - Overview

```
class Car
{
    public string Brand; // Field
    public int Speed { get; set; } // Property
    public const int MaxSpeed = 200; // Constant
    public readonly string Model; // Readonly Field

    public Car(string model) // Constructor
    {
        Model = model;
    }

    public void Drive() // Method
    {
        Console.WriteLine($"{Brand} is driving at {Speed} km/h.");
    }

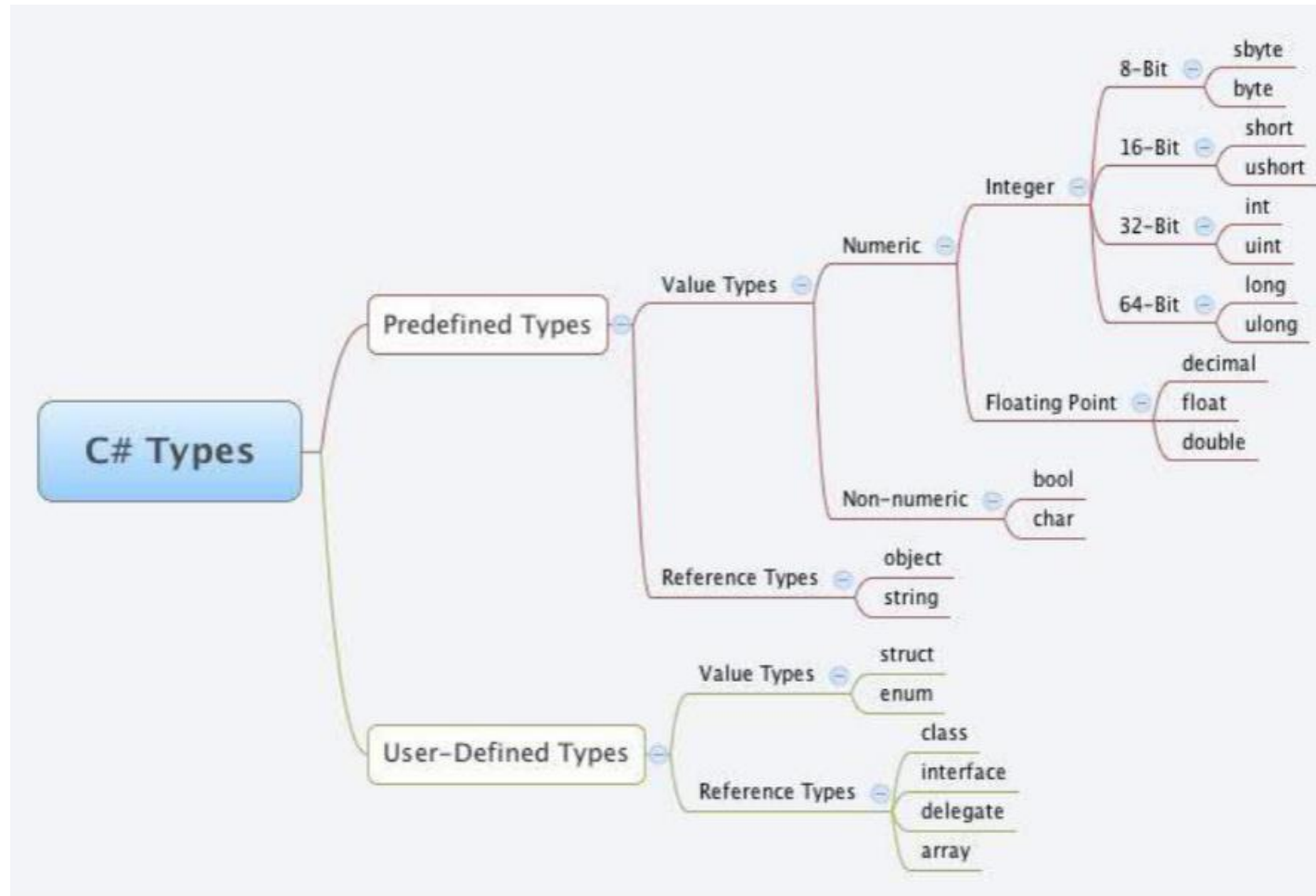
    public event Action EngineStarted; // Event

    public int this[int index] // Indexer
    {
        get { return index * Speed; }
    }
}
```

Data Types - Overview

- Types can be instantiated...
 - ...and then used: call methods, get and set properties, etc.
- Can convert from one type to another
 - Implicitly and explicitly
- Types are organized
 - Namespaces, files, assemblies
- There are two categories of types: value and reference
- Types are arranged in a hierarchy

Data Types - Overview







Value Type

- It holds a data value within its own memory space. It means the variables of these data types directly contain values.
- For example, consider integer variable “int i=100”
The system stores 100 in the memory space allocated for the variable “i” like this



Value Type – Primitive Types

Category	Type	Size	Range
 Integer	sbyte	8-bit	-128 to 127
	byte	8-bit	0 to 255
	short	16-bit	-32,768 to 32,767
	ushort	16-bit	0 to 65,535
	int	32-bit	-2,147,483,648 to 2,147,483,647
	uint	32-bit	0 to 4,294,967,295
	long	64-bit	-9 quintillion to 9 quintillion
	ulong	64-bit	0 to 18 quintillion
 Floating-Point	float	32-bit	$\pm 1.5 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$
	double	64-bit	$\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$
 Precise Decimal	decimal	128-bit	$\pm 1.0 \times 10^{-28}$ to $\pm 7.9 \times 10^{28}$
 Other	bool	1-bit	true or false
	char	16-bit	Unicode character (e.g., 'A', '√')

Value Type - Enum

- Defines a set of named constants
- Improves code readability and maintainability by replacing numeric values with meaningful names
- Example: `enum DaysOfWeek { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday }`

Value Type – Behavior example

```
int a = 10;
```

```
int b = a; // A COPY of 'a' is assigned to 'b'
```

```
b = 20; // Changing 'b' does NOT affect 'a'
```

```
Console.WriteLine(a); // Output: 10
```

```
Console.WriteLine(b); // Output: 20
```

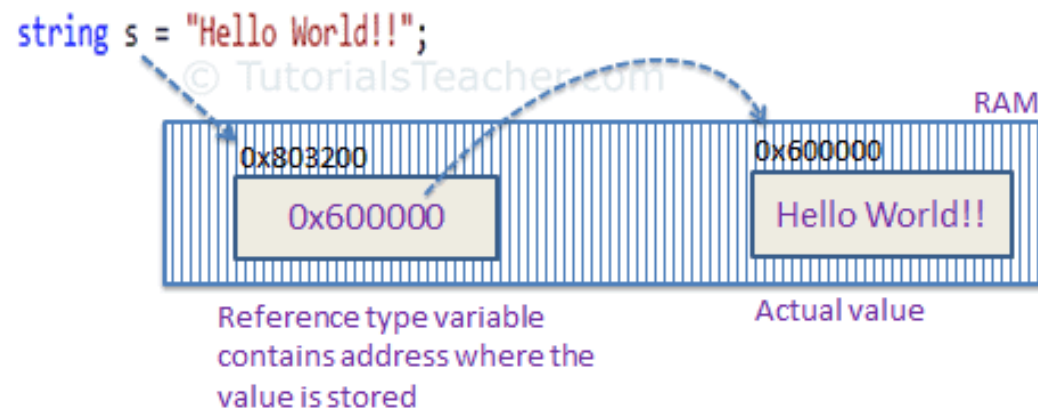

Value Type – Behavior example

```
void ChangeNumber(int num)
{
    num = 50; // Only the local copy changes
}

int x = 10;
ChangeNumber(x);
Console.WriteLine(x); // Output: 10 (original value remains unchanged)
```

Reference Type

- Unlike value types, a reference type doesn't store its value directly. Instead, it stores the address where the value is being stored. In other words, a reference type contains a pointer to another memory location that holds the data.
- For example, consider the following string variable:
“string s = "Hello World!!";”



Reference Type

- String
- Arrays (even if their elements are value types)
- Class
- Delegate

Reference Type – Behavior example

```
class Person
{
    public string Name;
}
```

```
Person person1 = new Person { Name = "Alice" };
```

```
Person person2 = person1; // person2 gets the REFERENCE to person1
```

```
person2.Name = "Bob"; // Modifying person2 also affects person1
```

```
Console.WriteLine(person1.Name); // Output: "Bob"
```

```
Console.WriteLine(person2.Name); // Output: "Bob"
```

Reference Type – Behavior example

```
void ChangePerson(Person p)
{
    p.Name = "Charlie"; // Modifies the original object
}
```

```
Person person = new Person { Name = "Alice" };
ChangePerson(person);
Console.WriteLine(person.Name); // Output: "Charlie"
```

Reference Type - String

- Reference type but behaves like a value type because it is immutable
- Immutable: Any modification creates a new string instead of modifying the existing one

```
string str1 = "Hello";  
str1 += " World"; // A new string is created, and str1 now points to it.
```

Reference Type - String - Ways to format

- String Concatenation: Simple but inefficient for multiple concatenations (creates new string instances)

```
string name = "Alice";  
int age = 25;  
string message = "Name: " + name + ", Age: " + age;  
Console.WriteLine(message);
```

- String Interpolation: Readable and efficient. Introduced in C# 6.

```
string name = "Alice";  
int age = 25;  
string message = $"Name: {name}, Age: {age}";  
Console.WriteLine(message);
```

Reference Type - String - Ways to format

- `string.Format` Method: Useful in older C# versions but replaced by interpolation in modern code

```
string message = string.Format("Name: {0}, Age: {1}", "Alice", 25);  
Console.WriteLine(message);
```

- `StringBuilder` (For Multiple Modifications): Efficient for large or frequent string modifications

```
StringBuilder sb = new StringBuilder();  
sb.Append("Name: Alice");  
sb.Append(", Age: 25");  
Console.WriteLine(sb.ToString());
```


Reference Type - String - Ways to format

■ string.Join (For Joining Collections)

```
string[] names = { "Alice", "Bob", "Charlie" };  
string result = string.Join(", ", names);  
Console.WriteLine(result); // Alice, Bob, Charlie
```

■ string.Concat (Concatenates Multiple Values)

```
string message = string.Concat("Name: ", "Alice", ", Age: ", 25);  
Console.WriteLine(message); // Output: Name: Alice, Age: 25
```

■ string.Replace (Replacing Text in Strings)

```
string text = "Hello, World!";  
string newText = text.Replace("World", "C#");  
Console.WriteLine(newText); // Hello, C#!
```

Reference Type - String - Ways to format

- `string.ToLower()` / `string.ToUpper()` (Case Conversion)

```
string input = "Hello";  
Console.WriteLine(input.ToUpper()); // HELLO  
Console.WriteLine(input.ToLower()); // hello
```

- `string.Trim()` (Removing Leading/Trailing Spaces)

```
string input = " C# ";  
Console.WriteLine(input.Trim()); // "C#"
```

Escaping Special Characters in C# Strings

- Using Escape Sequences (\)

Escape Sequence	Meaning	Example Output
<code>\"</code>	Double Quote	<code>"Hello \"John\"!"</code> → Hello "John"!
<code>\'</code>	Single Quote	<code>"It\'s a test"</code> → It's a test
<code>\\</code>	Backslash	<code>"C:\\Program Files\\"</code> → C:\Program Files\
<code>\n</code>	New Line	<code>"Hello\nworld"</code> → Hello (newline) world
<code>\t</code>	Tab	<code>"A\tB"</code> → A B
<code>\r</code>	Carriage Return	<code>"Hello\rworld"</code> → world (overwrites "Hello")
<code>\b</code>	Backspace	<code>"ABC\bD"</code> → ABD

Escaping Special Characters in C# Strings

- Using Verbatim Strings (@)

```
string filePath = @"C:\Program Files\MyApp\config.json";  
Console.WriteLine(filePath);  
// Output: C:\Program Files\MyApp\config.json
```

```
string multiLine = @"Hello,  
This is a  
Multi-line string."  
Console.WriteLine(multiLine);
```

Value Type vs Reference Type in C#

Feature	Value Type (<code>int</code> , <code>double</code> , <code>struct</code>)	Reference Type (<code>class</code> , <code>string</code> , <code>object</code>)
Memory Location	Stored in stack	Stored in heap (reference stored in stack)
Data Storage	Stores the actual value	Stores a reference (pointer) to the object
Nullability	Cannot be <code>null</code> (except <code>nullable</code> types, e.g., <code>int?</code>)	Can be <code>null</code>
Default Value	Default is <code>0</code> , <code>false</code> , etc.	Default is <code>null</code>
Assignment Behavior	Creates a copy of the value	Copies only the reference , not the object
Performance	Faster (stack allocation, no garbage collection)	Slightly slower (heap allocation, garbage collection)
Garbage Collection	Not applicable (stack variables are automatically cleaned)	Handled by garbage collector

Differences between Struct and Class

	Value (Struct)	Reference (Class)
Variable holds	Actual value	Memory location
Allocated on	Stack, member	Heap
Nullability	Always has value	May be null
Default value	0	null
Aliasing (in a scope)	No	Yes
Assignment means	Copy data	Copy reference

Boxing

- The process of Converting a Value Type (char, int etc.) to a Reference Type(object) is called **Boxing**.
- Boxing is implicit conversion process in which object type (super type) is used.

Unboxing

- The process of converting reference type into the value type is known as **Unboxing**.
- It is explicit conversion process.

Type Conversion

- Implicit conversions
 - Occur automatically
 - Guaranteed to succeed
 - No information (precision) loss
- Explicit conversions
 - Require a cast
 - May not succeed
 - Information (precision) might be lost

Type Conversion

- `int x = 123456;`
- `long y = x;` `// implicit`
- `short z = (short)x;` `// explicit`

- `double d = 1.2345678901234;`
- `float f = (float)d;` `// explicit`
- `long l = (long)d;` `// explicit`

Variables in C#

- We should create our variables by following examples:
 - `studentName`
 - `subject`
 - `work_day ...`
- The wrong examples would be
 - `student Name`
 - `work-day`
 - `1place`

Arrays

- Arrays allow a group of elements of a specific type to be stored in a contiguous block of memory
- Arrays are reference types
- Derived from `System.Array`
- Zero-based
- Can be multidimensional
 - Arrays know their length(s) and rank
- Bounds checking

Arrays

- Declare

```
int[] primes;
```

- Allocate

```
int[] primes = new int[9];
```

- Initialize

```
int[] primes = new int[] {1,2,3,5,7,11,13,17,19};  
int[] primes = {1,2,3,5,7,11,13,17,19};
```

- Access and assign

```
prime2[i] = prime[i];
```

- Enumerate

```
foreach (int i in prime) Console.WriteLine(i);
```

List

- Declare

```
List<string> foods = new List<string>();
```

- Add item to list

```
foods.Add("Cookie");
```

- Initialize

```
List<string> foods = new List<string>();  
List<string> foods1 = new List<string>(){ "Pizza", "Cake" };
```

- Access and assign

```
Foods1[0] = "Sushi";
```

- Enumerate

```
foreach (string i in food1) Console.WriteLine(i);
```

Statements - Overview

- Statement lists
- Block statements
- Declarations
 - Constants
 - Variables
- Expression statements
 - checked, unchecked
 - lock
 - using
- Conditionals
 - if
 - switch
- Loop Statements
 - while
 - do
 - for
 - foreach
- Jump Statements
 - break
 - continue
 - goto
 - return
 - throw
- Exception handling
 - try
 - throw

Statements - while

- Requires bool expression

```
int i = 0;  
while (i < 5) {  
    ...  
    i++;  
}
```

```
int i = 0;  
do {  
    ...  
    i++;  
}  
while (i < 5);
```

```
while (true) {  
    ...  
}
```


Statements - for

```
for (int i=0; i < 5; i++) {  
    ...  
}
```

```
for (;;) {  
    ...  
}
```

Statements - foreach

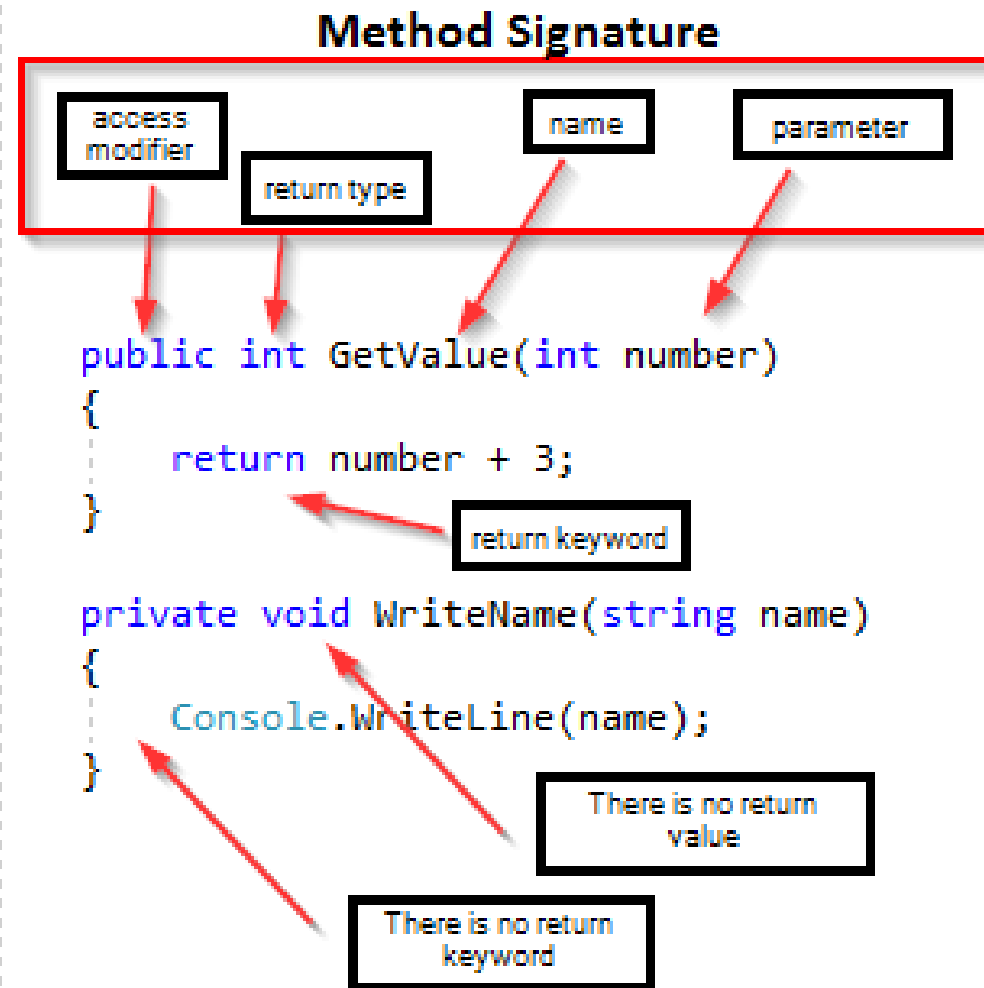
- Iteration of arrays

```
public static void Main(string[] args) {  
    foreach (string s in args)  
        Console.WriteLine(s);  
}
```

Conditions

- Basic Condition Statements
- Nested Conditional Statements
- Switch-Case Statements

Methods - Method Signatures



Methods

- Parameters and Arguments
- Optional Parameters

Nullable types

- Declaring
- Null-Coalescing Operator (??)
- Null-Conditional Operator (?.)

```
Person? person = null;  
string name = person?.Name ?? "Unknown"; // If person or Name is null, use "Unknown"  
Console.WriteLine(name); // Output: "Unknown"
```

Q&A

C# Intermediate

Viet Nguyen

Mar-2025

**Nash
Tech.**



Learning Objectives

- Be able to write C# code in an object-oriented manner
- Be able to use the object-oriented concepts while writing that code

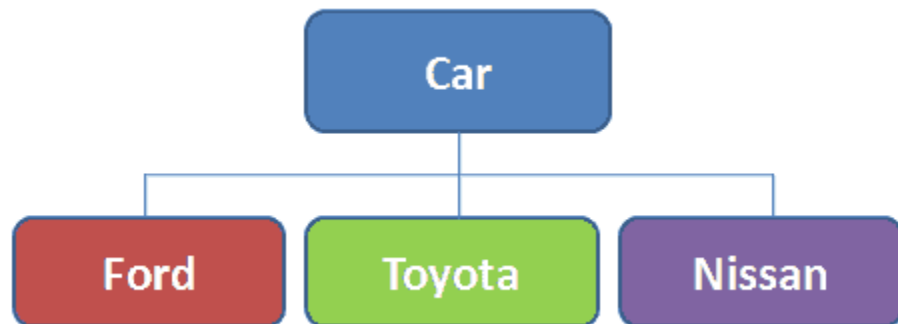
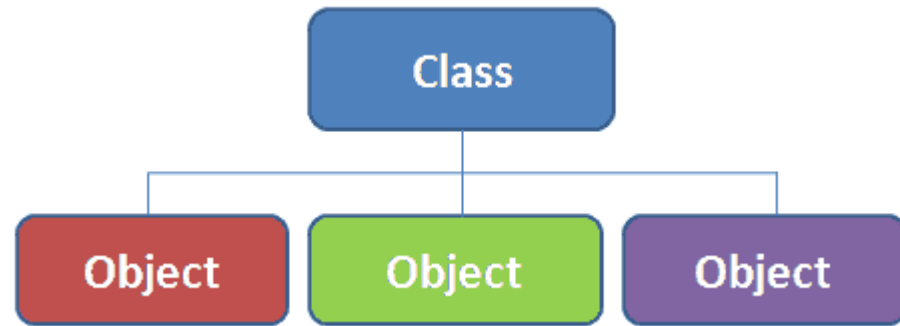
Agenda

- Introduction to Classes
- Association between Classes
- Interfaces
- Abstract Classes
- References
- LINQ
- Asynchronous C# programming
- Delegates & Events

Classes

- Anatomy of a class
 - Data (represented by fields)
 - Behaviour (represented by methods/ functions)
- Naming Convention
 - Pascal Case
 - camel Case
- What is an Object ?
 - An instance of a class

Classes and Objects



```
public class Car
{
    public string Tyres { get; set; } // Property
    public string Color { get; set; } // Property

    public void Driving() // Method
    {
        Console.WriteLine("The car is driving...");
    }

    public void Reverse() // Method
    {
        Console.WriteLine("The car is reversing...");
    }

    public static void Main() // Entry point
    {
        // Declare an instance.
        Car toyota = new Car();

        // Call the member.
        toyota.Reverse();
    }
}
```

Classes

- Constructors
 - To put an Object to early state
- Constructor Overloading
 - The use of more than one constructor in an instance class.
- Access modifier
 - To creates safety in our program
- Properties
 - Encapsulates getter/setter for accessing a field

Static Class

```
public static class TestClass
{
    private static int number;

    static TestClass()
    {
        number = 54;
    }
}
```

Anonymous Classes

- Can contain only public fields.
- Fields must all be initialized.
- Members never are static.
- Cannot specify any methods.
- Cannot implement interface

```
myAnonymousObj = new { Name = "John", Age = 32 };
```

Inheritance

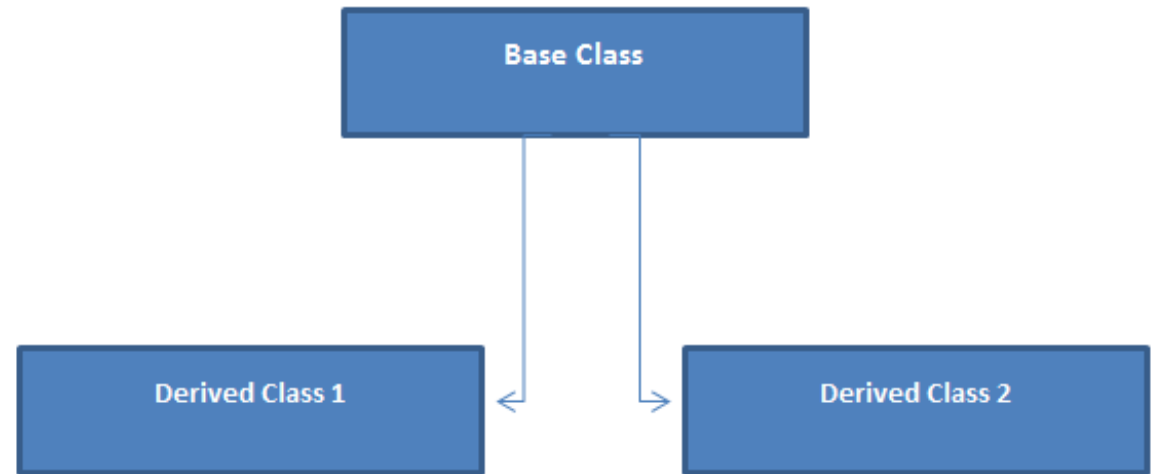
- A kind of relationship between two classes that allows one to inherit code from the other.
- Referred to as Is-A relationship - allows you to build new class definitions that extend the functionality of an existing class: A Car is a Vehicle
- Benefits: code re-use and polymorphic behavior.

```
public class Car : Vehicle  
{  
  
}
```


Inheritance

```
class DerivedClass: BaseClass
{
}

class DerivedSubClass: DerivedClass
{
}
```



Composition

- A kind of relationship that allows one class to contain another.
- Referred to as Has-A relationship: A Car has an Engine
- Benefits: Code re-use, flexibility and a means to loose-coupling

Favour Composition over Inheritance

- Problems with inheritance:
 - Easily abused by amateur designers / developers
 - Leads to large complex hierarchies
 - Such hierarchies are very fragile, and a change may affect many classes
 - Results in tight coupling
- Benefits of composition:
 - Flexible
 - Leads to loose coupling


Interface

- An interface is a language construct that is similar to a class (in terms of syntax) but is fundamentally different.
- An interface is simply a declaration of the capabilities (or services) that a class should provide.
- An interface is purely a declaration.
- An interface can only declare methods and properties, but not fields
- Members of an interface do not have access modifiers
- Interfaces help building loosely coupled applications

Abstract Classes

```
public abstract class AbstractExampleClass
{
    public void PrintToConsole(string text)
    {
        Console.WriteLine(text);
    }
}

class Program
{
    static void Main(string[] args)
    {
        AbstractExampleClass example = new AbstractExampleClass();
    }
}
```


 class AbstractionExamples.AbstractExampleClass

Cannot create an instance of the abstract class or interface 'AbstractExampleClass'

Sealed Classes

```
public sealed class SealedClass  
{  
    ...  
}
```

```
public class DerivedClass : SealedClass  
{  
}
```

 class ConsoleApp1.DerivedClass

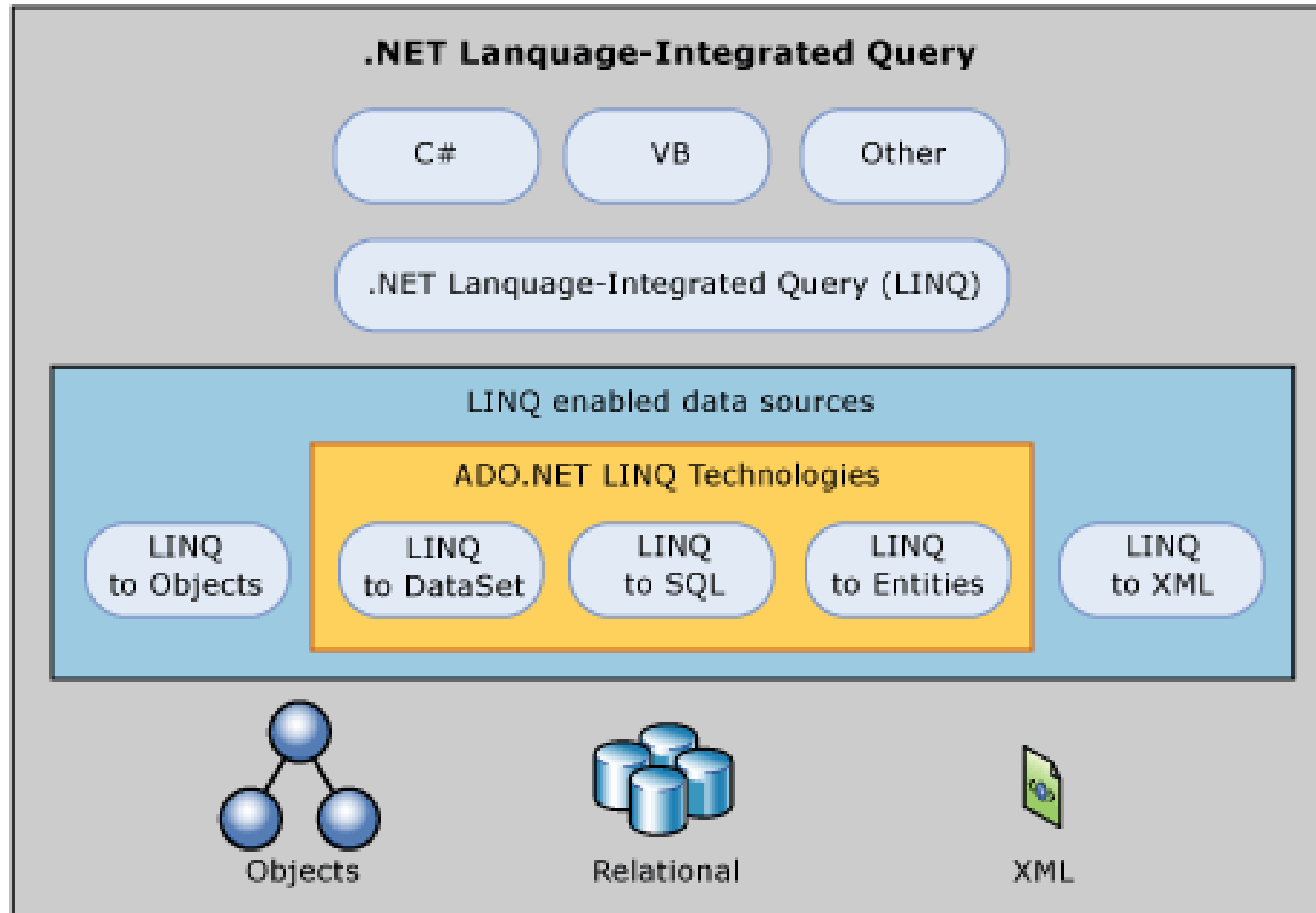
'DerivedClass': cannot derive from sealed type 'SealedClass'

References

- What do you think will be printed by this code?

```
Dissertation diss1 = new Dissertation();  
Dissertation diss2 = diss1;  
diss1.CurrentPage = 0;  
diss2.CurrentPage = 16;  
Console.WriteLine(diss1.CurrentPage);  
Console.WriteLine(diss2.CurrentPage);
```

LinQ Architecture



LinQ Query Syntax

`from <range variable> in <IEnumerable<T> or
IQueryable<T> Collection>`

`<Standard Query Operators> <Lambda expression>`

`<select or groupBy operator> <result formation>`

```
// string collection
IList<string> courses = new List<string>() {
    "C# Tutorials",
    "VB.NET Tutorials",
    "Learn C++",
    "MVC Tutorials" ,
    "Java"
};

// LINQ Query Syntax
var result = from s in courses
              where s.Contains("Tutorials")
              select s;
```

Delegate

- Delegate Syntax
 - `delegate Result_Type identifier([parameters]);`
- There are three steps in defining and using delegates:
 - Declaration of our delegate
 - Instantiation, creating the delegate's object
 - Invocation, where we call a referenced method

Delegate

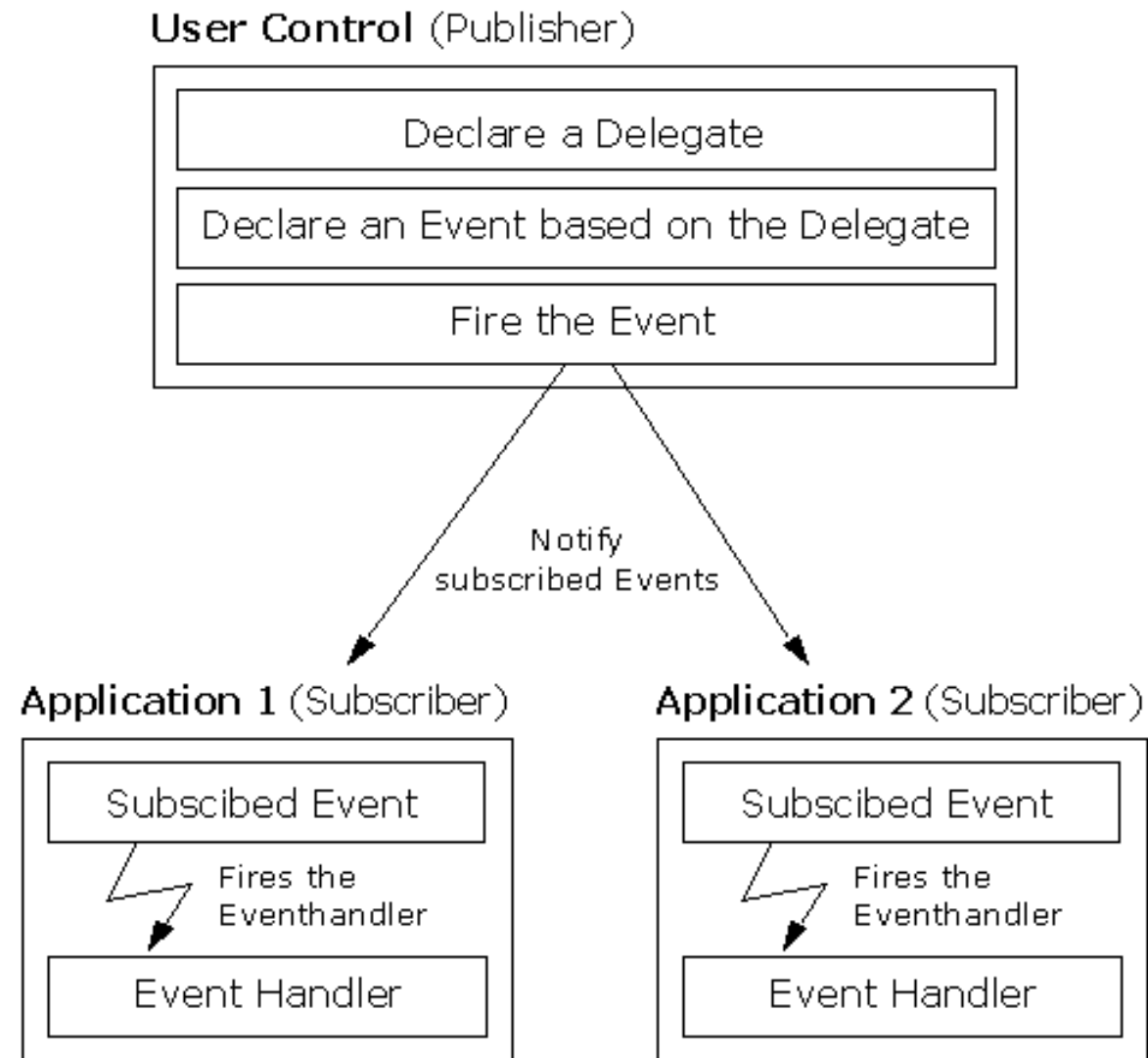
```
public delegate void SimpleDelegate ()
```

This declaration defines a delegate named `SimpleDelegate`, which will encapsulate any method that takes no parameters and returns no value.

```
public delegate int ButtonClickHandler (object obj1, object obj2)
```

This declaration defines a delegate named `ButtonClickHandler`, which will encapsulate any method that takes two objects as parameters and returns an int.

Event



Asynchronous C# programming

