TOC

What is the difference between `HAVING` and `WHERE`?

`WHERE` works with columns, `HAVING` works with aggregations and requires `GROUP BY`.

What is OOP?

Object-oriented programming (OOP) is a computer programming model that organizes software design around data, or objects, rather than functions and logic. An object can be defined as a data field that has unique attributes and behavior.

OOP has 4 concepts.

Why are we using `OOP`?

For reusability of the code.

What is the garbage collector?

Garbage collection (GC) is a form of automatic memory management. The garbage collector attempts to reclaim memory which was allocated by the program, but is no longer referenced—also called garbage.

What is a `namespace`?

A namespace is designed for providing a way to keep one set of names separate from another. The class names declared in one namespace does not conflict with the same class names declared in another.

Access modifiers

- `public`

The type or member can be accessed by any other code in the same assembly or another assembly that references it. The accessibility level of public members of a type is controlled by the accessibility level of the type itself.

على مستوى الsolution

- `private`

The type or member can be accessed only by code in the same class or struct.

على مستوى الclass فقط

- `protected`

The type or member can be accessed only by code in the same class, or in a class that is derived from that class.

على مستوى الclass والclasses اللي وارثة منه

- `internal`

The type or member can be accessed by any code in the same assembly, but not from another assembly. In other words, internal types or members can be accessed from code that is part of the same compilation.

على مستوى الproject

- `protected internal`

The type or member can be accessed by any code in the assembly in which it's declared, or from within a derived class in another assembly.

على مستوى الproject والclasses اللي وارثة من الclass الأب

- `private protected`

The type or member can be accessed by types derived from the class that are declared within its containing assembly. A private protected member is accessible by types derived from the containing class, but only within its containing assembly.

متاح للclasses الأبناء لكن في نفس الproject

`static`

Use the static modifier to declare a static member, which belongs to the type itself rather than to a specific object.

Use cases

- shared value
- a member that is used once without modeling

Notes on `static class`

- Cannot be instantiated
- All its members must be `static`
- Cannot be used as the base class for inheritance

`params`

By using the params keyword, you can specify a method parameter that takes a variable number of arguments. The parameter type must be a single-dimensional array.

```
public class MyClass
{
    public static void UseParams(params int[] list)
    {
        for (int i = 0; i < list.Length; i++)
        {
            Console.Write(list[i] + " ");
        }
        Console.WriteLine();
    }

    public static void UseParams2(params object[] list)
    {
        for (int i = 0; i < list.Length; i++)
        {
            Console.Write(list[i] + " ");
        }
        Console.WriteLine();
    }

    static void Main()
    {
        // You can send a comma-separated list of arguments of the
        // specified type.
        UseParams(1, 2, 3, 4);
        UseParams2(1, 'a', "test");

        // A params parameter accepts zero or more arguments.
        // The following calling statement displays only a blank line.
        UseParams2();

        // An array argument can be passed, as long as the array
        // type matches the parameter type of the method being called.
        int[] myIntArray = { 5, 6, 7, 8, 9 };
        UseParams(myIntArray);
```

```
        object[] myObjArray = { 2, 'b', "test", "again" };
        UseParams2(myObjArray);

        // The following call causes a compiler error because the object
        // array cannot be converted into an integer array.
        //UseParams(myObjArray);

        // The following call does not cause an error, but the entire
        // integer array becomes the first element of the params array.
        UseParams2(myIntArray);
    }
}
/*
Output:
    1 2 3 4
    1 a test

    5 6 7 8 9
    2 b test again
    System.Int32[]
*/
```

What is the difference between `this` and `base`?

`this` represents the current class instance while `base` the parent.

Example of usage:

```
public class Parent
{
    public virtual void Foo()
    {
    }
}

public class Child : Parent
{
    // call constructor in the current type
    public Child() : this("abc")
    {
    }

    public Child(string id)
    {
    }

    public override void Foo()
    {
        // call parent method
        base.Foo();
```

```
    }
}
```

```
1 reference
class Parent
{
    private string _ParentFieldTest1;
    protected string _ParentFieldTest2;
    public string _ParentFieldTest3;

    0 references
    void Test()
    {
        this.
    }
}

0 references
class Child :
{
    private s          ieldTest1;
    protected          dFieldTest2;
    public string _ChildFieldTest3;

}
```

Equals
GetHashCode
GetType
MemberwiseClone
Test
ToString
_ParentFieldTest1
_ParentFieldTest2
_ParentFieldTest3

bool object.Equals(object obj)
Determines whether the specified object is equal to the current object.
Note: Tab twice to insert the 'Equals' snippet.

```
1 reference
class Parent
{
    private string _ParentFieldTest1;
    protected string _ParentFieldTest2;
    public string _ParentFieldTest3;

    0 references
    void Test()
    {
        //this.
    }
}

0 references
class Child : Parent
{
    private s
    protected          ieldTest2;
    public st          dTest3;

    0 references
    void Test
    {
        this.
    }
```

GetType
MemberwiseClone
TestChild
ToString
_ChildFieldTest1
_ChildFieldTest2
_ChildFieldTest3
_ParentFieldTest2
_ParentFieldTest3

bool object.Equals(object obj)
Determines whether the specified object is equal to the current object.
Note: Tab twice to insert the 'Equals' snippet.

```
1 reference
class Parent
{
    private string _ParentFieldTest1;
    protected string _ParentFieldTest2;
    public string _ParentFieldTest3;

    0 references
    void Test()
    {
        //this.
    }
}

0 references
class Child : Parent
{
    private string _ChildFieldTest1;
    protected
    public st
                Equals
                GetHashCode
                GetType
                MemberwiseClone
                ToString
                _ParentFieldTest2
                _ParentFieldTest3
    0 references
    void Test
    {
        base.
    }
```

bool object.Equals(object obj)
Determines whether the specified object is equal to the current object.
Note: Tab twice to insert the 'Equals' snippet.

What is the difference between `override` and `overload`?

`override`	`overload`
Overriding in C# is to provide a specific implementation in a derived class method for a method already existing in the base class	Overloading in C# is to create multiple methods with the same name with different implementations
Parameters	
In C# *overriding*, the methods have the same name, same parameter types and the same number of parameters	In C# *overloading*, the methods have the same name but a different number of parameters or a different type of parameters
Occurrence	
In C#, *overriding* occurs within the base class and the derived class	In C#, *overloading* occurs within the same class
Binding Time	
The binding of the *overriden* method call to its definition happens at runtime	The binding of the *overloaded* method call to its definition happens at compile time

`override`	`overload`
Synonyms	
Overriding is called as: - **runtime polymorphism** - **dynamic polymorphism** - **late binding**	*Overloading* is called as: - **compile time polymorphism** - **static polymorphism** - **early binding**

What is the difference between `abstract class` and `interface`?

`abstract class`	`interface`
Can contain both declaration, and definition part	It contains only declaration part
Multiple inheritance is **not** achieved by `abstract class`	Multiple inheritance is achieved by `interface`
It contains `constructor`	It does **not** contain `constructor`
It can contain different types of access modifiers like: - `public` - `private` - `protected` - etc.	It only contains `public` access modifier because everything in the interface is `public`
The performance of the abstract class is fast	The performance of the interface is slow because it requires time to search actual method in the corresponding class
It is used to implement the core identity of class	It is used to implement peripheral abilities of class
A class can only use (*inherit from*) **one** `abstract class`	A class can use (*implement*) multiple `interface`s
If many implementations are of the same kind and use common behavior, then it is superior (*preferred*) to use `abstract class`	If many implementations only share methods, then it is superior (*preferred*) to use `interface`
`abstract class` can contain: - **methods** - **fields** - **constants** - **properties** - etc.	`interface` can only contain: - **methods** - **properties** - **indexers** - **events**
It can be *fully*, *partially*, or *not implemented*	It has to be **fully** implemented

What is the difference between `struct` and `class`?

`class`

Class and Object are the basic concepts of Object-Oriented Programming which revolve around the real-life entities. A class is a user-defined blueprint or prototype from which objects are created. Basically, a class combines the fields and methods(member function which defines actions) into a single unit. In C#, classes support polymorphism, inheritance and also provide the concept of derived classes and base classes.

`struct`

Structs are light versions of classes. Structs are value types and can be used to create objects that behave like built-in types.

Structs share many features with classes but with the following limitations as compared to classes.

- Struct cannot have a default constructor (a constructor without parameters, the constructor must fully implement all the fields) or a destructor.
- Structs are value types and are copied on assignment.
- Structs are value types while classes are reference types.
- Structs can be instantiated without using a new operator.
- A struct cannot inherit from another struct or class, and it cannot be the base of a class. All structs inherit directly from `System.ValueType`, which inherits from `System.Object`.
- Struct cannot be a base class. So, Struct types cannot abstract and are always implicitly sealed.
- `abstract` and `sealed` modifiers are not allowed and `struct` member cannot be `protected` or `protected internal`s.
- Function members in a struct cannot be `abstract` or `virtual`, and the `override` modifier is allowed only to the `override` methods inherited from `System.ValueType`.
- Struct does not allow the instance field declarations to include variable initializers. But, static fields of a struct are allowed to include variable initializers.
- A struct can implement interfaces.
- Default value is not `null`
- A struct can be used as a nullable type and can be assigned a null value. *Must add `?` at the end of the type.*

  ```
  int? x;
  ```

- Faster than classes.

When to use `struct` or `class`?

To answer this question, we should have a good understanding of the differences.

`struct`	`class`
Structs are value types, allocated either on the stack or inline in containing types	Classes are reference types, allocated on the heap and garbage-collected
Allocations and de-allocations of value types are in general cheaper than allocations and de-allocations of reference types	Assignments of large reference types are cheaper than assignments of large value types
In structs, each variable contains its own copy of the data (except in the case of the `ref` and `out` parameter variables), and an operation on one variable does not affect another variable.	In classes, two variables can contain the reference of the same object and any operation on one variable can affect another variable.

In this way, struct should be used only when you are sure that,

- It logically represents a single value, like primitive types (`int`, `double`, etc.).
- It is immutable.
- It should not be boxed and un-boxed frequently.

What is the difference between `value types` and `reference types`?

The general difference is that a reference type lives on the heap, and a value type lives inline, that is, wherever it is your variable or field is defined.

A variable containing a value type contains the entire value type value. For a `struct`, that means that the variable contains the entire `struct`, with all its fields.

A variable containing a reference type contains a pointer, or a reference to somewhere else in memory where the actual value resides.

This has one benefit, to begin with:

- value types always contain a value
- reference types can contain a null-reference, meaning that they don't refer to anything at all at the moment

Internally, reference types are implemented as pointers, and knowing that, and knowing how variable assignment works, there are other behavioral patterns:

- copying the contents of a value type variable into another variable, copies the entire contents into the new variable, making the two distinct. In other words, after the copy, changes to one won't affect the other
- copying the contents of a reference type variable into another variable, copies the reference, which means you now have two references to the same somewhere else storage of the actual data. In other

words, after the copy, changing the data in one reference will appear to affect the other as well, but only because you're really just looking at the same data both places

RAM Diagram

OS	Stack	Heap
Stores stuff related to the OS	Stores values, and pointers	Stores referenced values

OOP Concepts

1. Abstraction

(تجرد على مستوى الاستخدام)

Used upon instantiating or inheriting from the class.

> لما تكون سواق، مش لازم تبقى ميكانيكي

Abstraction is the concept of taking some object from the real world, and converting it to programming terms. Such as creating a `Human` class and giving it:

- properties like: `int health`, `int age`, `string name`, etc.
- and methods that execute an action or operation on these properties, like: `Eat()`, `Walk()`, etc.

In software engineering and computer science, abstraction is:

- the process of removing physical, spatial, or temporal details or attributes in the study of objects or systems to focus attention on details of greater importance; it is similar in nature to the process of generalization;
- the creation of abstract concept-objects by mirroring common features or attributes of various non-abstract objects or systems of study – the result of the process of abstraction.

2. Encapsulation

> تغليف

In object-oriented computer programming (OOP) languages, the notion of encapsulation (or OOP Encapsulation) refers to the bundling of data, along with the methods that operate on that data, into a single unit. Many programming languages use encapsulation frequently in the form of classes.

Encapsulation may also refer to a mechanism of restricting the direct access to some components of an object, such that users cannot access state values for all of the variables of a particular object (access modifiers: `public`, `private`, etc.). Encapsulation can be used to hide both data members and data functions or methods associated with an instantiated class or object.

3. Inheritance

In object-oriented programming, inheritance is the mechanism of basing an object or class upon another object (prototype-based inheritance) or class (class-based inheritance), retaining similar implementation. Also

defined as deriving new classes (sub classes) from existing ones such as super class or base class and then forming them into a hierarchy of classes.

4. Polymorphism

> تعدد الأوجه

In programming language theory and type theory, polymorphism is the provision of a single `interface` to entities of different types or the use of a single symbol to represent multiple different types.

The word polymorphism means having many forms. In object-oriented programming paradigm, polymorphism is often expressed as *one interface, multiple functions*.

Polymorphism can be static or dynamic. In static polymorphism, the response to a function is determined at the compile time. In dynamic polymorphism, it is decided at run-time.

1. static or compile-time

   Method overloading represents static polymorphism.

2. dynamic

   Method overriding, and interfaces represent static polymorphism.

Notes

Position of the increment operator changes its order of execution

```
int x = 1;
int y = x++;
```

> x = 2 y = 1

Assignment occurs first, then the increment.

```
int x = 1;
int y = ++x;
```

> x = 2 y = 2

The increment happens first, then the assignment.

```
6++;
```

> Error CS1059: The operand of an increment or decrement operator must be a variable, property or indexer

Interruption tools

- `continue`

  The continue statement: starts a new iteration of the closest enclosing iteration statement.

- `break`

  The break statement: terminates the closest enclosing iteration statement or switch statement.

- `return`

  The return statement: terminates execution of the function in which it appears and returns control to the caller.

- `goto`

  The goto statement: transfers control to a statement that is marked by a label.

  > `finally` gets performed before `goto`

`abstarct`, `sealed`, `static class`

	`abstract`	`sealed`	`static`
Allows inheritance?	✅	❌	❌
Allows instantiation?	❌	✅	❌