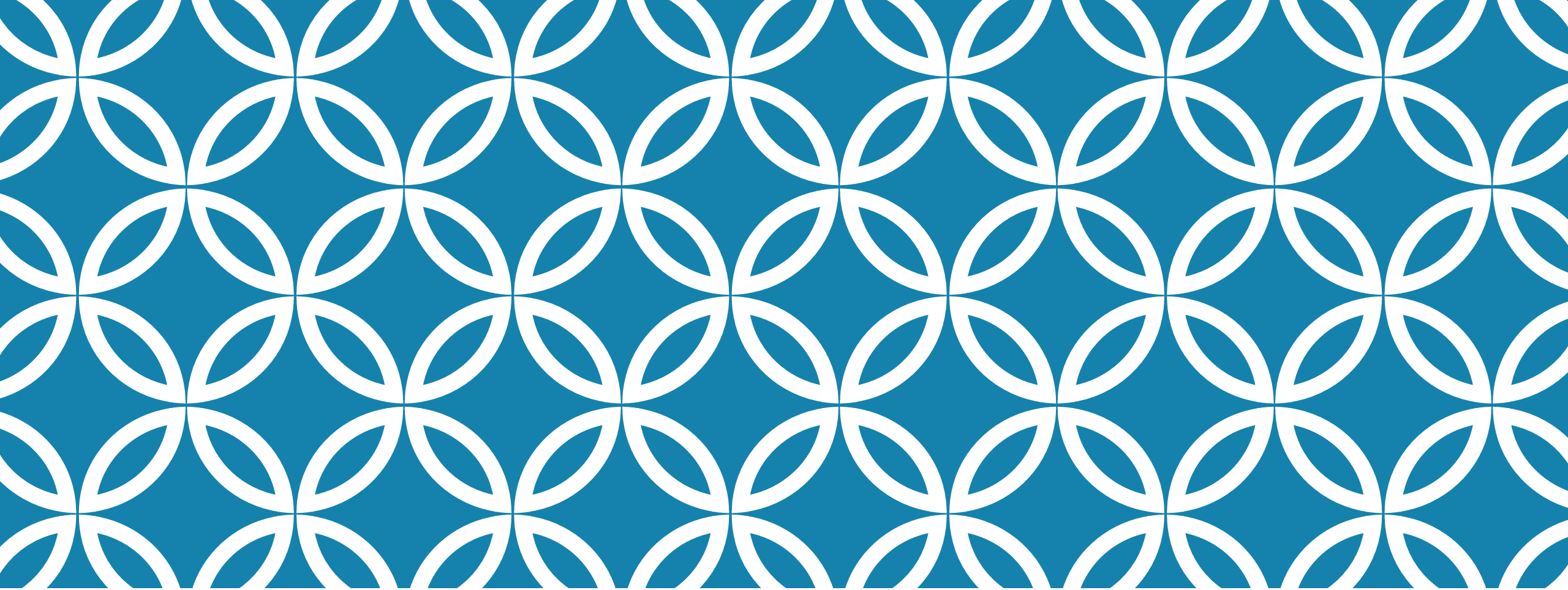


IMPLEMENTATION OF OOP CONCEPTS

http://www.Madadyar.ir

INTRODUCTION

- Designing an application using Object Oriented Programming is inevitable due to the features it provides, like inheritance, polymorphism, encapsulation and so on.
- Apart from these concepts, it is very important to understand some logical concepts along with technical concepts so that writing the code is easy, clean and re-usable.
- This Lecture will explain some of the concepts we use in most of the classes that we write.

CLASS

- A class is basically a combination of a set of rules on which we will work in a specific program. It contains definitions of new data types like fields or variables, operations that we will perform on data (methods) and access rules for that data.

```
Class Example
{
    /* fields,
    Variables,
    Methods,
    Properties,
    */
}
```

OBJECT

- Objects are defined as an instance of a class. Objects are built on the model defined by the class.
- An object can be created in memory using the "new" keyword.
- In C# objects are reference types and other data type variables are value types.
- In C# objects are stored in the heap while other value types are stored in the stack.

Example

```
Example exmplObject = new Example();
```

NESTED TYPES

- You can declare a class within another class.

- The inner class in called the nested class and the outer class is called the nesting class. This is done when the nested class is meaningless out of the scope of the nesting class.

NESTED TYPES

```
public class Human
{
    int age;
    public int Age
    {
        get { return age; }
        set { age = value; }
    }
    public class Heart
    {
        Human owner;

        public Human Owner
        {
            get { return owner; }
            set { owner = value; }
        }
        public Heart(Human ownerVal)
        {
            Owner = ownerVal;
        }
        bool isHealthy;

        public bool IsHealthy
        {
            get { return isHealthy; }
            set { isHealthy = value; }
        }
    }
}
```

INHERITANCE (GENERALIZATION)

- Inheritance is relevant due to the concept of “Code Reusability”. Inheritance is a feature by which a class acquires attributes of another class. The class that provides its attributes is known as the base class and the class that accepts those attributes is known as a derived class. It allows programmers to enhance their class without reconstructing it.

Example

```
class BaseClass
 { }
 class DerivedClass : BaseClass
 { }
```

INHERITANCE

```
public class Vehicle   // Base Class
{
    public String Color { get; set; }           // Accessible to all derived classes
    public Int32 Price { get; set; }            // Accessible to all derived classes
    private String VehicleType { get; set; }    // Accessible to only to base class
}

public class Car : Vehicle // Derived Class
{
    public Int32 Discount { get; set; }         // Derived class specific property

    public Car()
    {
        this.Discount = 9000;
    }
}

public static class ClientCode
{
    public void GetVehicleProperties()
    {
        Car _car = new Car();

        // All base class properties accessible except the 'VehicleType' as it is private
        _car.Color = "Red";
        _car.Price = 50000;
    }
}
```

INTERFACE

- An interface is a description of the actions that an object can do. for example when you flip a light switch, the light goes on, you don't care how, just that it does.

- In Object Oriented Programming, an Interface is a description of all functions that an object **must** have in order to be an "X".

- C# does not support multiple inheritances. An interface on the other hand resolves the multiple inheritances problem. One class can implement **many** interfaces.

INTERFACE

For instance, you might have an interface called IWriter. IWriter has one method:

```
public interface IWriter
{
    void Write(string s);
}
```

Notice how generic that is. It doesn't say WHAT it is writing, or how it's writing... Only that it writes.

You can then have concrete methods that implement IWriter, such as MemoryWriter, ConsoleWriter, PrinterWriter, HTMLWriter, etc... each writes in a different way, but they all implement the same simple interface with only one method.

```
public class ConsoleWriter : IWriter
{
    public void Write(string s) {
         Console.WriteLine(s);
    }
}

public class MemoryWriter : IWriter
{
    public void Write(string s) {
        // code to create a memory object and write to it
    }
}
```

POLYMORPHISM

- Ploy means more than one form. In the beginning we have already seen an example of Polymorphism. Same method name with different parameter is an example for the polymorphism.

- **Method Overloading** and **Method Overriding** will be used in polymorphism.

METHOD OVERLOADING

```
public class MyLogger
{
    public void LogError(Exception e)
    {
        // Implementation goes here
    }

    public bool LogError(Exception e, string message)
    {
        // Implementation goes here
    }
}
```

METHOD OVERRIDING

```
using System;
public class Complex
{
    private int real;
    public int Real
    { get { return real; } }

    private int imaginary;
    public int Imaginary
    { get { return imaginary; } }

    public Complex(int real, int imaginary)
    {
        this.real = real;
        this.imaginary = imaginary;
    }

    public static Complex operator +(Complex c1, Complex c2)
    {
        return new Complex(c1.Real + c2.Real, c1.Imaginary + c2.Imaginary);
    }

    public override string ToString()
    {
        return (String.Format("{0} + {1}i", real, imaginary));
    }
}
```

OPERATOR OVERLOADING (NOT POLYMORPHISM)

```
public class Complex
{
    private int real;
    public int Real
    { get { return real; } }

    private int imaginary;
    public int Imaginary
    { get { return imaginary; } }

    public Complex(int real, int imaginary)
    {
        this.real = real;
        this.imaginary = imaginary;
    }

    public static Complex operator +(Complex c1, Complex c2)
    {
        return new Complex(c1.Real + c2.Real, c1.Imaginary + c2.Imaginary);
    }
}
```

ABSTRACT CLASS

- The Abstract class will be as a super class for all our class. Abstract class cannot be accessed by an object, which means we cannot create an object for an abstract class.

```
    static void Main(string[] args)
    {
        GuestVist objnew = new GuestVist();
    }
}
```

class ShanuHouseClassConsole.GuestVist
C#: An instance of an abstract class can not be created

```
abstract class GuestVist
{
    public int noofSecurity;
    public String SecurityName = String.Empty;
}
```

ABSTRACT CLASS

```
abstract class GuestVist
    {
         public void Guestwelcomemessage()
         {
             Console.WriteLine("Welcome to our AbstractHome");
         }
         public void GuestName()
         {
             Console.WriteLine("Guest name is: Abstract");
         }
         public abstract void purposeofVisit();

    }
// derived class to inherit the abstract class
public class Houseclass : GuestVist
    {
         static void Main(string[] args)
         {
             Houseclass objHouse = new Houseclass();
             objHouse.Guestwelcomemessage();
         }

         public override void purposeofVisit()
         {
             Console.WriteLine("Abstract just came for a Meetup and spend some time ");
         }
    }
```

DIFFERENCE BETWEEN ABSTRACT METHOD AND VIRTUAL METHOD

- Both similarities use the override keyword.
- Abstract Method can only be declared in Abstract Class, which means no body part for abstract method in Abstract class.
- However, for virtual it can have body part.

DIFFERENCE BETWEEN ABSTRACT METHOD AND VIRTUAL METHOD

```
abstract class GuestVist
    {
public abstract void purposeofVisit();  // Abstract Method

        public virtual void NoofGuestwillvisit()  // Virtual Method
        {
            Console.WriteLine("Total 5 Guest will Visit your Home");
        }
    }
    class AbstractHouseClass : GuestVist
    {
public override void purposeofVisit()  // Abstract method Override
        {
            Console.WriteLine("Abstract just for a Meetup and spend some time ");
        }

public override void NoofGuestwillvisit() // Virtual method override
        {
            Console.WriteLine("Total 20 Guest Visited our Home");
        }

        static void Main(string[] args)
        {
            AbstractHouseClass objHouse = new AbstractHouseClass();

objHouse.purposeofVisit();
            objHouse.NoofGuestwillvisit();
            Console.ReadLine();
        }

    }
```

SEALED CLASS

- As name says this class cannot be inherited by other classes.

```
public sealed class OwnerofficialRoom
{
    public void AllMyPersonalItems()
    {
        Console.WriteLine("All Items in this rooms are personal to me no one else can access or inherit me");
    }
}
class HouseSealedClass : OwnerofficialRoom
{
    static void Main(stri
    {
    }
}
```

class SealedClassConsole.OwnerofficialRoom

C#: The sealed class 'SealedClassConsole.OwnerofficialRoom' can not be subclassed

DIFFERENCE BETWEEN STATIC CLASS AND SEALED CLASS

- We can create an Object (instance) for the Sealed Class, we can have in my sealed class section I created a sample Sealed class and in Main Method I created an object to access the sealed Class. And in a Sealed Class both Static and non-Static methods can be written.
- But for a Static Class it's not possible to create an Object. In Static Class only Static members are allowed which means in a static Class it's not possible to write non-static method.
- Another interesting one in Static class is that memory will be allocated for all static variable and methods during execution but for the non static variable and methods memory will be allocated only when the object for the class are created.

What will happen when we inherit Static class in our derived class?

Let's see an example when I try to inherit my static class from my derived class. It shows me the below warning message.

```
public static class OwnerofficialRoom
{
    public static void AllMyPersonalItems()
    {
        Console.WriteLine("All Items in this rooms are personal to me no one else can access or inherit me");
    }
}

class HouseStaticClass : OwnerofficialRoom
{
    static void Main(stri
    {
    }
}
```

class StaticClass.OwnerofficialRoom
C#: 'StaticClass.OwnerofficialRoom': cannot derive from static class 'StaticClass.HouseStaticClass'

What will happen when we declare non-Static method in a Static class?

Let's see an example when I try to create a non-Static method at my Static Class.

```
public static class OwnerofficialRoom
{
    public static void AllMyPersonalItems()
    {
        Console.WriteLine("All Items in this rooms are personal to me no one else can access or inherit me");
    }
    public void Iam_non_Static_Method()
    {
    }
}
```

C#: Cannot declare instance members in a static class
C#: Name does not match the naming convention. Suggested name 'IamNonStaticMethod'

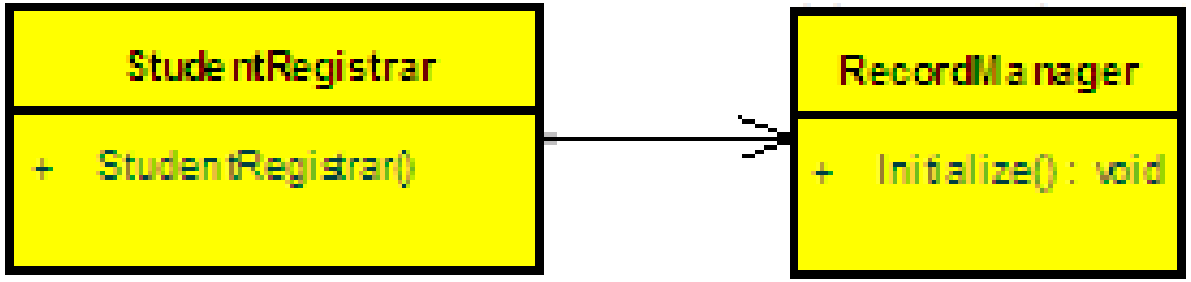
ASSOCIATION

- Association is defined as a structural relationship, that conceptually means that the two components are linked to each other.

- This kind of relation is also referred to as a using relationship, where one class instance uses the other class instance or vice-versa, or both may be using each other.

- But the main point is, the **lifetime** of the instances of the two classes are independent of each other and there is no ownership between two classes.

ASSOCIATION




```
public class StudentRegistrar
{
    public StudentRegistrar ();
    {
        new RecordManager().Initialize();
    }
}
```

DEPENDENCY

- also called a using relationship, which means, one class is dependent on another class.

- Dependency is defined as a relation between two classes, where one class depends on another class but another class may or not may depend on the first class. So any change in one of the classes may affect the functionality of the other class, that depends on the first one.

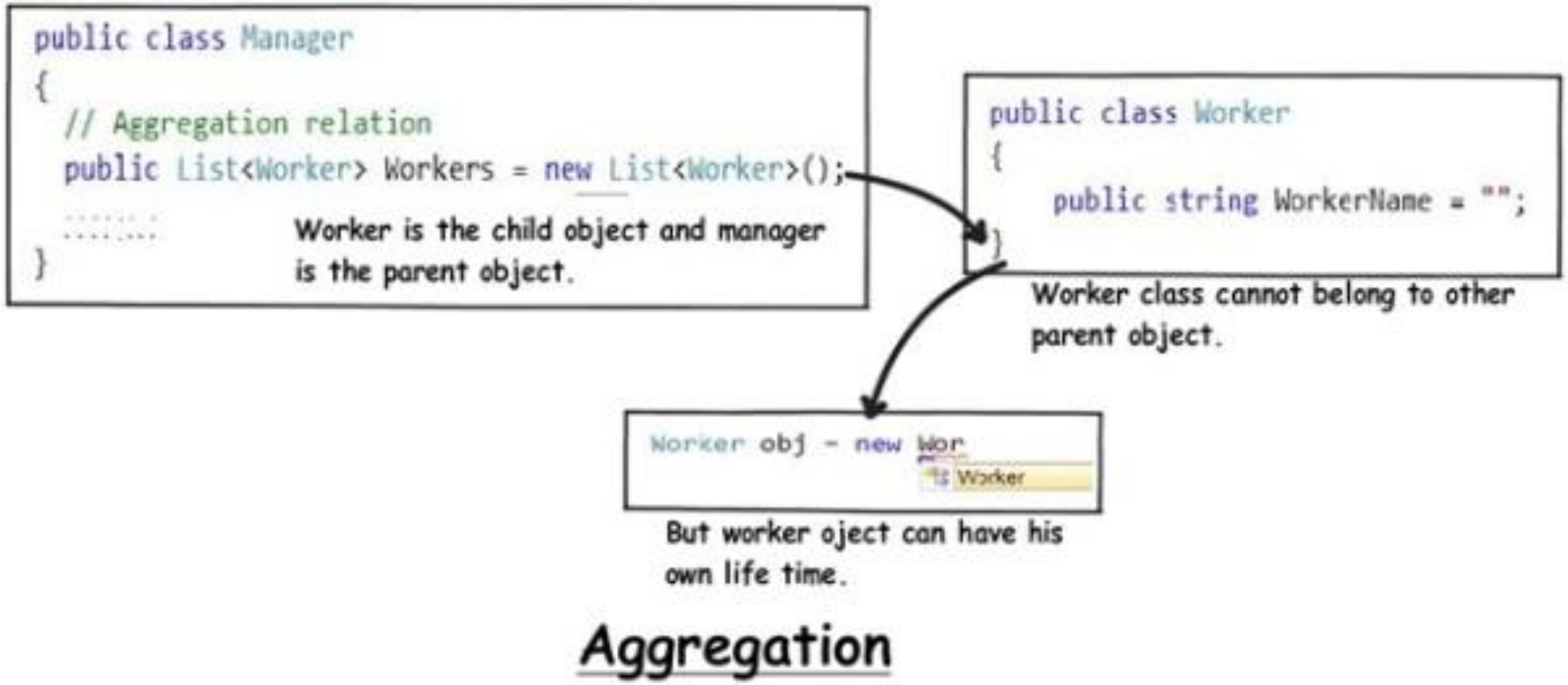
DEPENDENCY

```
public class Customer
{
    public Guid CustomerId { get; set; }
    public String CustomerName { get; set; }

    // Other Customer related functions & properties
}

public class Order
{
    public Int32 OrderId { get; set; }
    public Guid OrderCustomerId { get; set; }
    public DateTime OrderDateTime { get; set; }

    // Other Order functions & properties

    public Order(Customer customer)
    {
        // Save order with CustomerId
        this.OrderCustomerId = customer.CustomerId;
    }
}
```

AGGREGATION

- Aggregation is the same as association but with an additional point that there is an **ownership** of the instances, unlike association where there was no ownership of the instances. To understand it better, let's add another class named Department to our example explained above.

AGGREGATION

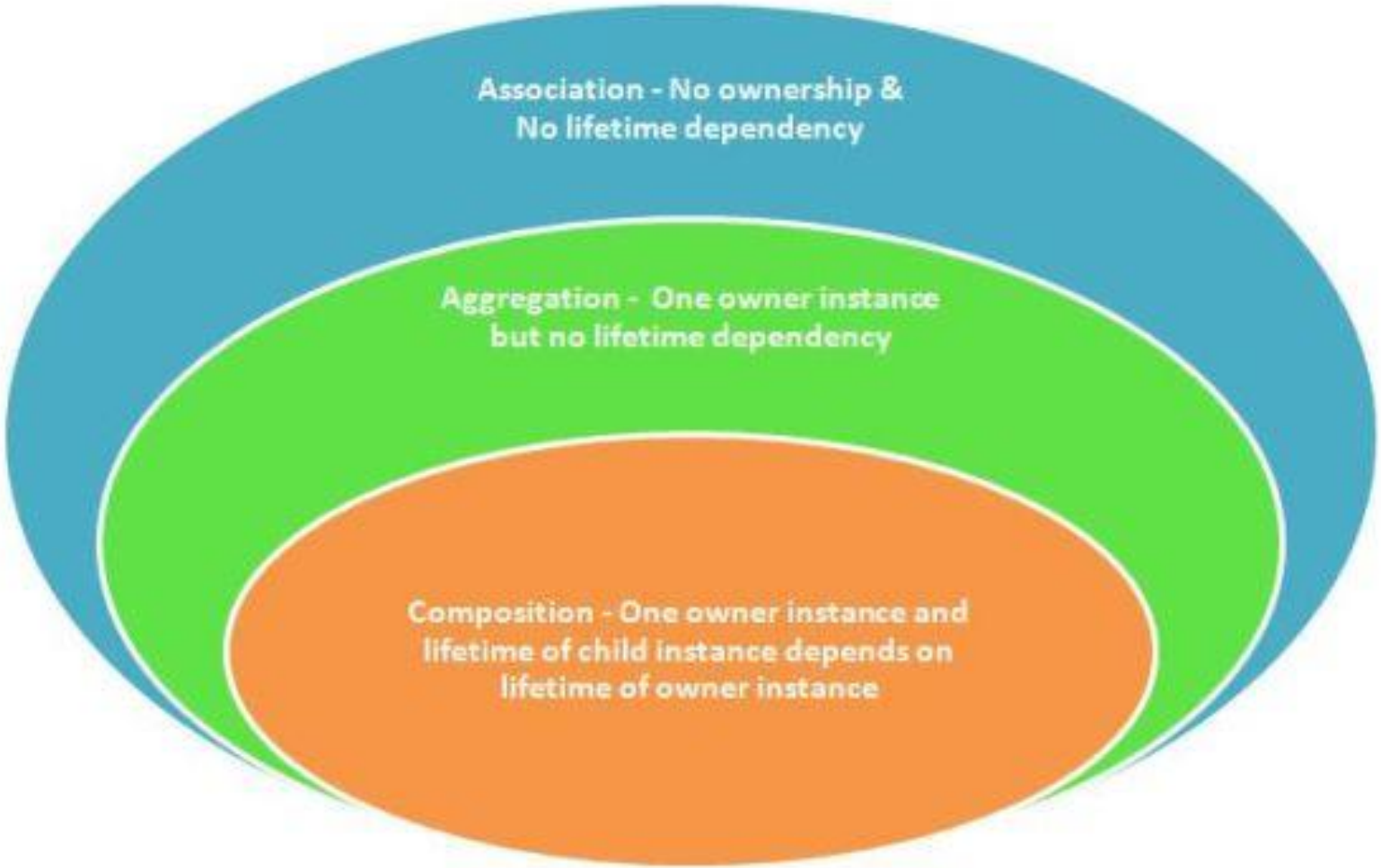


COMPOSITION

- This is the same as that of aggregation, but with the additional point that the **lifetime** of the child instance is dependent on the owner or the parent class instance. To the same code above, let's add another class named University.

COMPOSITION

```
public class University
{
    public String Name { get; set; }
    List<Department> _lstDepartments;

    public void GetDepartments()
    {
        // _university1 here is the owner of '_department1' & '_department2'
        // Lifetime of the instances will end, if the '_university1' is disposed
        // So lifetime is dependent on the owner instance i.e. '_university1'
        Department _department1 = new Department { Name = "Department1" };
        Department _department2 = new Department { Name = "Department2" };

        _lstDepartments.Add(_department1);
        _lstDepartments.Add(_department1);
    }
}

public class ClientCode
{
    public void CallingCode()
    {
        University _university1 = new University { Name = "University1" };
    }
}
```


Association - No ownership &
No lifetime dependency
Aggregation - One owner instance
but no lifetime dependency
Composition - One owner instance and
lifetime of child instance depends on
lifetime of owner instance

AND ………………


computer
language
bits
numbers
programming
number
binary
instructions
like
something
decimal
program
groups
write
just
see
need
Java
Language
ASCII
Figure
really
languages
objects
get
make
series
Logo
color
bit
values
octal
programs
group
output
symbols
English
computers
Table
value
memory
assembler
represent
steps
lesson
writing
line
shows
know
understand
computer's
pixels
used
want
first
one
going
microprocessor
use
two
give
things
software
high-level
lots
looks
read
kinds
three
form
etc