

## Classes

A building block of an application.

The class was two parts.

- Data represented by fields.
- Behaviors represented by functions.

Object is an instance of an class that resides on the memory.

Why use static members?

To represent concepts that are singleton. That means that we have only one instance of that concept into memory.

## Constructor

Is a method that is called when an instance of a class is created. The constructor serve to put an object on the earlier state, to initialize the fields in the class. No return type.

## Constructor Overloading

Method with the same name but with different signature (Return type; types and names of parameters).

### Constructor Overloading

```
public class Customer
{
    public Customer() { ... }

    public Customer(string name) { ... }

    public Customer(int id, string name) { ... }
}
```

```

12 2 references
13  public Customer()
14  {
15      orders = new List<order>();
16  }
17 // public Customer(int id) : this() -> ": this()" uses the first constructor to
18 0 references
19  public Customer(int id) : this()
20  {
21      this.Id = id;

```

But we can choose what constructor is call in other constructors.

```

22 1 reference
23  public Customer(int id, string name) : this(id)
24  {
25      this.Id = id;
26      this.Name = name;

```

```

//OBJECT initializer, initialize object without constructors
Customer c = new Customer { Id = 1 , Name = "Maria"};
Console.WriteLine(c.Name);

```

## Methods

Signature methods – Consist in the name and the number of the type parameters.

```

public class Point
{
    public void Move(int x, int y) {}
}

```

→ Signature

Overload is method with the same name but different signatures.

```
public class Point
{
    public void Move(int x, int y) {}

    public void Move(Point newLocation) {}

    public void Move(Point newLocation, int speed) {}
}
```

Params modifier - no need to initialize the array when the function is called.

## The Params Modifier

```
public class Calculator
{
    public int Add(params int[] numbers){}
}

var result = calculator.Add(new int[]{ 1, 2, 3, 4 });
var result = calculator.Add(1, 2, 3, 4);
```

Ref modifier – change var a.

## The Ref Modifier

```
public class Weirdo
{
    public void DoAWeirdThing(ref int a)
    {
        a += 2;
    }
}

var a = 1;
weirdo.DoAWeirdThing(ref a);
```

Out modifier - returns value to the caller

## The Out Modifier

```
public class MyClass
{
    public void MyMethod(out int result)
    {
        result = 1;
    }
}

int a;
myClass.MyMethod(out a);
```

In C#, both `ref` and `out` are used as parameter modifiers, but they have different purposes:

### 1. `ref` Modifier:

- The `ref` keyword is used to pass arguments by reference.
- It allows a method to modify the parameter value passed to it.
- The parameter must be initialized before being passed to the method.
- It's useful when you need to pass a value to a method and have that method modify the original value.
- Example:

```
void ModifyValue(ref int x)
{
    x = x * 2;
}

int value = 5;
ModifyValue(ref value);
// value is now 10
```

### 2. `out` Modifier:

- The `out` keyword is similar to `ref`, but it doesn't require the parameter to be initialized before being passed to the method.
- It's typically used when the method needs to return multiple values.
- The method is required to assign a value to the `out` parameter before it returns.
- Example:

```
void GetValues(out int x, out int y)
{
    x = 10;
    y = 20;
}

int a, b;
GetValues(out a, out b);
// a is 10, b is 20
```

In summary:

- `ref` is used when you want to pass a value by reference and modify it within the method.

- `out` is used when you want the method to return multiple values or modify the parameter without requiring it to be initialized before the method call.

Field – variable declared at the class level.

## Initialization

```
public class Customer
{
    List<Order> Orders = new List<Order>();
}
```

Read-only fields – only initialized once.

## Read-only Fields

```
public class Customer
{
    readonly List<Order> Orders = new List<Order>();
}
```

Access Modifier – a way to control access to a class or its members.  
This control stops the or try the creation of bugs.

## Access Modifiers

- Public
- Private
- Protected
- Internal
- Protected Internal

## Object-oriented programming

- Encapsulation / Information Hiding
- Inheritance
- Polymorphism

## Encapsulation

### Encapsulation (in practice)

- Define fields as private
- Provide getter/setter methods as public

Encapsulation is one of the fundamental principles of object-oriented programming (OOP). It refers to the bundling of data (attributes or fields) and methods (functions or procedures) that operate on the data into a single unit called a class. This unit controls access to the data, preventing unauthorized access and modification.

Encapsulation helps in achieving several important objectives:

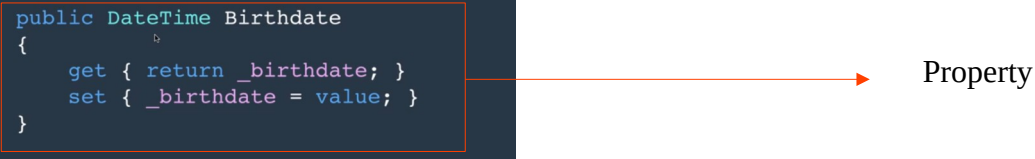
1. **Data Hiding**: By encapsulating data within a class, you can hide the implementation details of how the data is stored and manipulated. This allows you to protect the integrity of the data and prevent unintended modifications.
2. **Abstraction**: Encapsulation allows you to represent complex real-world entities as objects with well-defined interfaces. Users of the class only need to know how to interact with the object through its public methods, without needing to understand the internal workings of those methods.
3. **Modularity**: Encapsulated classes can be easily reused in different parts of a program or in different programs altogether. The encapsulated data and methods form a self-contained unit that can be easily understood and maintained.

In many object-oriented programming languages like C#, Java, and Python, encapsulation is achieved by declaring the data members of a class as private and providing public methods (getters and setters) to access and modify those data members. This allows the class to enforce constraints on how the data is accessed and manipulated, ensuring data integrity and promoting a clean and modular design.

A class member – encapsulates a getter/setter for accessing a field.  
Property- is used to creating a setetr with less code.

```
public class Person
{
    private DateTime _birthdate;

    public DateTime Birthdate
    {
        get { return _birthdate; }
        set { _birthdate = value; }
    }
}
```



Property

## Auto-implemented Properties

```
public class Person
{
    public DateTime Birthdate { get; set; }
}
```

Create indexer

## How?

```
public class HttpCookie
{
    public string this[string key]
    {
        get { ... }
        set { ... }
    }
}
```

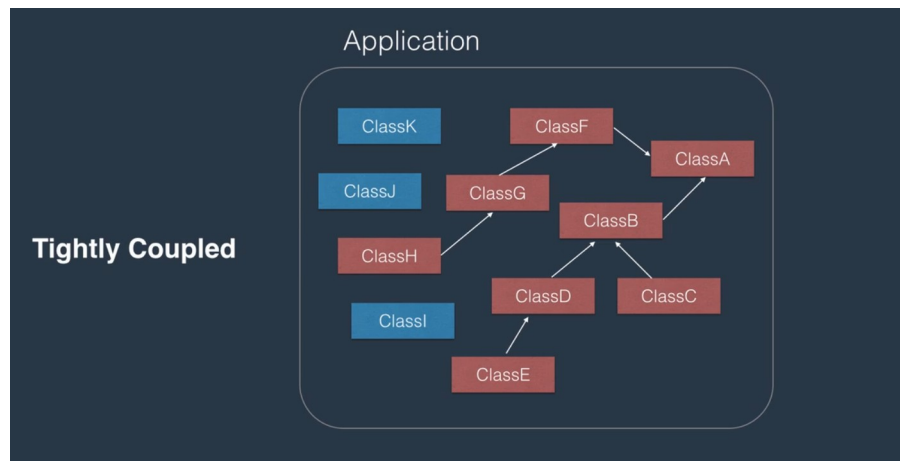


Association between classes

## Class Coupling

Coupling – is how interconnected classes and subsystems are.

Example of a coupled application where classes are dependent of each other.



So to if we want to change one class, and that change don't affect other we need to:

You need to understand

- Encapsulation
- The relationships between classes
- Interfaces

Relationships between classes:

## Types of Relationships

- Inheritance
- Composition

Inheritance – Relationship between two classes that allows one to inherit code from the other.

Composition

Composition – Relationship between two classes that allow one to contain the other. Has-a relationship example: (car has an engine).

## Benefits

- Code re-use
- Flexibility
- A means to loose-coupling

## Example

### Syntax

```
public class Installer
{
    private Logger _logger;

    public Installer(Logger logger)
    {
        _logger = logger;
    }
}
```

Inheritance- less flexibility more tightly coupled.

Composition – more flexibility loose coupling.

## Inheritance – Access Modifiers

### Access Modifiers

- public
- private
- protected
- internal
- protected internal

## Public

Accessible from everywhere.

```
public class Customer
{
    public void Promote()
    {
    }
}

...

var customer = new Customer();
customer.Promote();
```

## Private

Accessible only from the class.

```
public class Customer
{
    private int CalculateRating()
    {
    }
}

...

var customer = new Customer();
customer.calculateRating();
```

## Protected

Accessible only from the class  
and its derived classes.

↓  
Inherited class

```
public class Customer
{
    protected int CalculateRating()
    {
    }
}

...

var customer = new Customer();
customer.calculateRating();
```

## Internal

Accessible only from the same  
assembly.

↓  
Same Library

```
internal class RateCalculator
{
}

...

// In the same assembly
var calc = new RateCalculator();

// In another assembly
var calc = new RateCalculator();
```

## Protected Internal

Accessible only from the same  
assembly or any derived classes.

```
public class Customer
{
    protected internal void Weirdo()
    {
    }
}
```

