

# **Oriented object programming in C#**

---

# The classes

---

- ❑ A class is a combination of code (methods) and data (attributes)
- ❑ We can create the different objects(instances) by the class.
- ❑ Instantiating a class consists of creating an object on its model.

# The classes:Syntax

---

- ❑ The syntax for declaring a class is as follows:

```
modifier class name_of_classe {  
//...  
}
```

# The classes : example

```
9      class Person
10     {
11         private string name;
12         private string family;
13         private int age;
14
15         //default constructor
16         0 references
17         public Person()
18         {
19             this.name = "";
20             this.family = "";
21             this.age = 0;
22         }
23         //Method
24         2 references
25         public void Identify()
26         {
27             Console.WriteLine(name + " " + family + " " + age + " ");
28         }
29     }
```

# The classes

---

- ❑ The members or fields of a class can be data (attributes), methods (functions) and properties.
- ❑ These fields can be accompanied by one of the following different keywords:
  - **private** is accessible only by the internal methods of the public class
  - **public** is accessible by any function defined
  - **protected** is accessible only by the internal methods of the class or a derivative object
  - **Final** cannot be modified
  - **Abstract** contains one or more abstract methods, that do not have an explicit definition

# The classes

---

- ❑ A class is instantiated through the new operator.

- **Example:**

```
Person p = new Person("John","Smith",30);
```

- ❑ The keyword **this** refers to the current object.

- **Example :**

```
this.age=age;
```

# Constructors

---

- ❑ A constructor is a special method used to initialize an object when it is created.
- ❑ It always has the name of the class for which it is defined.
- ❑ It is public and has no return type
- ❑ A class can have one or more constructors.

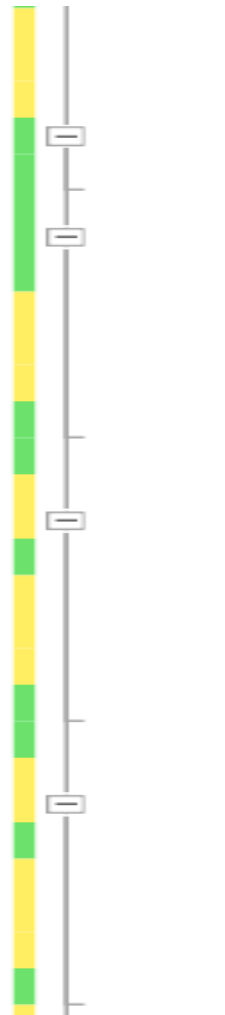
# Constructors

---

- ❑ There are three types of constructors:
- ❑ **By default:** for the creation and then the initialization of an object in the case where the programmer do not give values.
- ❑ **Parameterized:** for the creation and initialization of an object with values given by the programmer.
- ❑ **By copying:** for the creation and then initialization of an object by copying the values of another object.



# Example



```
11 private string name;
12 private string family;
13 private int age;
14 //constructors
15 //default constructor
16 0 references
17 public Person()
18 {
19     this.name = "";
20     this.family = "";
21     this.age = 0;
22 }
23 //Parameterized constructor
24 1 reference
25 public Person(string n,string f, int a)
26 {
27     this.name = n;
28     this.family = f;
29     this.age = a;
30 }
31 //copy constructor
32 1 reference
33 public Person(Person p)
34 {
35     name = p.name;
36     family = p.family;
37     age = p.age;
38 }
```

# Example

❑ Here is a short test program with the result obtained:

```
9      class Program
10     {
11         0 references
12         static void Main(string[] args)
13         {
14             Person p1 = new Person("John", "Smith", 30);
15             Console.Out.Write("p1=");
16             p1.Identify();
17             Person p2 = new Person(p1);
18             Console.Out.Write("p2=");
19             p2.Identify();
20             Console.ReadKey();
21         }
22     }
23 }
```

```
p1=John Smith 30
p2=John Smith 30
```

# ENCAPSULATION

---

- ❑ It is the grouping in the same entity called the object of the data. We can only reach the attributes of an object through its methods or services
- ❑ The accessors and mutators: An accessor is a method that will allow us to access the variables of the objects in reading and a mutator, in writing.
- ❑ We are talking about Getters and Setters. The Getters are of the same type as the variable they must return
- ❑ The Setters are, on the other hand, of the void type. these methods do not return any value, they just update them.

# Getters and Setters

```
42 public string getName()  
43 {  
44     return name;  
45 }  
46 0 references  
47 public void setName(string Name)  
48 {  
49     this.name = Name;  
50 }  
51 0 references  
52 public string getFamily()  
53 {  
54     return family;  
55 }  
56 0 references  
57 public void setFamily(string Family)  
58 {  
59     this.family = Family;  
60 }  
61 0 references  
62 public int getAge()  
63 {  
64     return age;  
65 }  
66 0 references  
67 public void setAge(int Age)  
68 {  
69     this.age = Age;  
70 }
```

# Properties

---

- ❑ Our experience in programming has taught us that for each private attribute of a class there must be two methods for reading and for modifying that attribute.
- ❑ These methods are called getter/setter.
- ❑ Thus for the name attribute of the person class for example it is necessary to write the set and get methods:

```
42  
43  
44  
45  
46  
47  
48  
49  
50  
public string getName()  
{  
    return name;  
}  
0 references  
public void setName(string Name)  
{  
    this.name = Name;  
}
```

# Properties

---

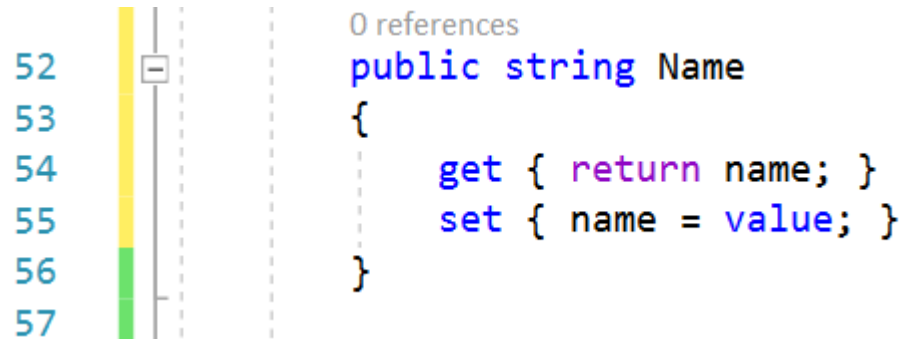
- ❑ The properties help avoid the heaviness of these accessor/modifier methods.
- ❑ They allow private attributes to be manipulated as if they were public. Their syntax is as follows:

```
public type property{  
  get {...}  
  set {...}  
}
```

# Properties

---

❑ Thus for the "name" attribute of the example they are declared as follows:

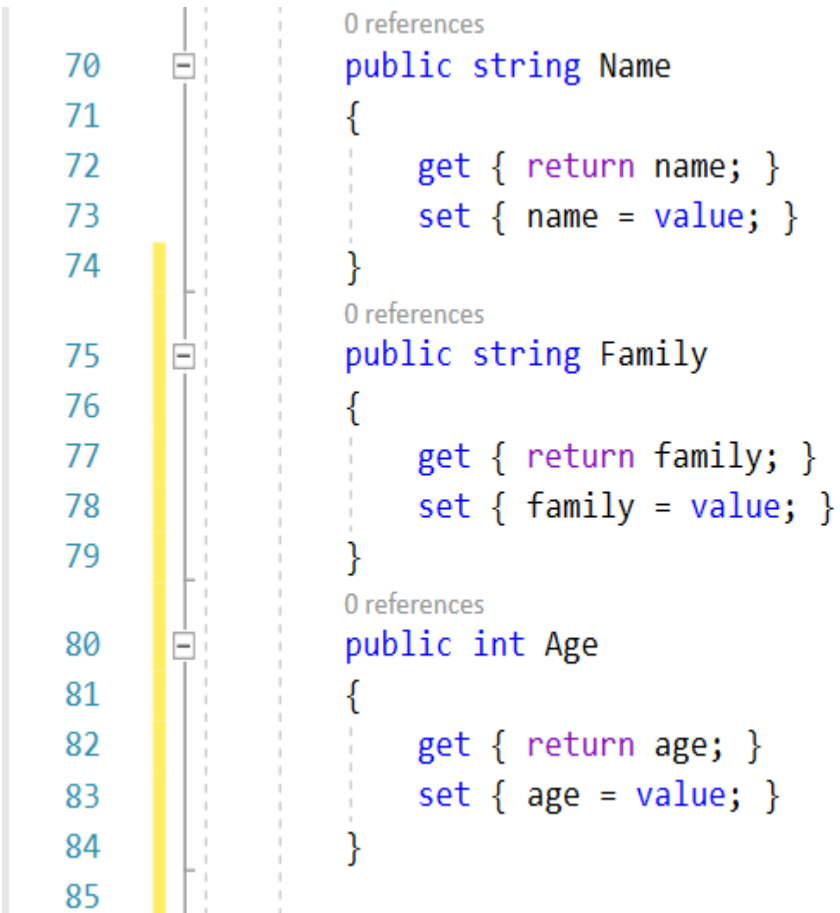


```
52  
53  
54  
55  
56  
57  
0 references  
public string Name  
{  
    get { return name; }  
    set { name = value; }  
}
```

❑ Thus Name="John" is equivalent to calling the setName("John") method for example.

# Properties

---



```
70 0 references  
71 public string Name  
72 {  
73     get { return name; }  
74     set { name = value; }  
75 }  
76 0 references  
77 public string Family  
78 {  
79     get { return family; }  
80     set { family = value; }  
81 }  
82 0 references  
83 public int Age  
84 {  
85     get { return age; }  
86     set { age = value; }  
87 }
```



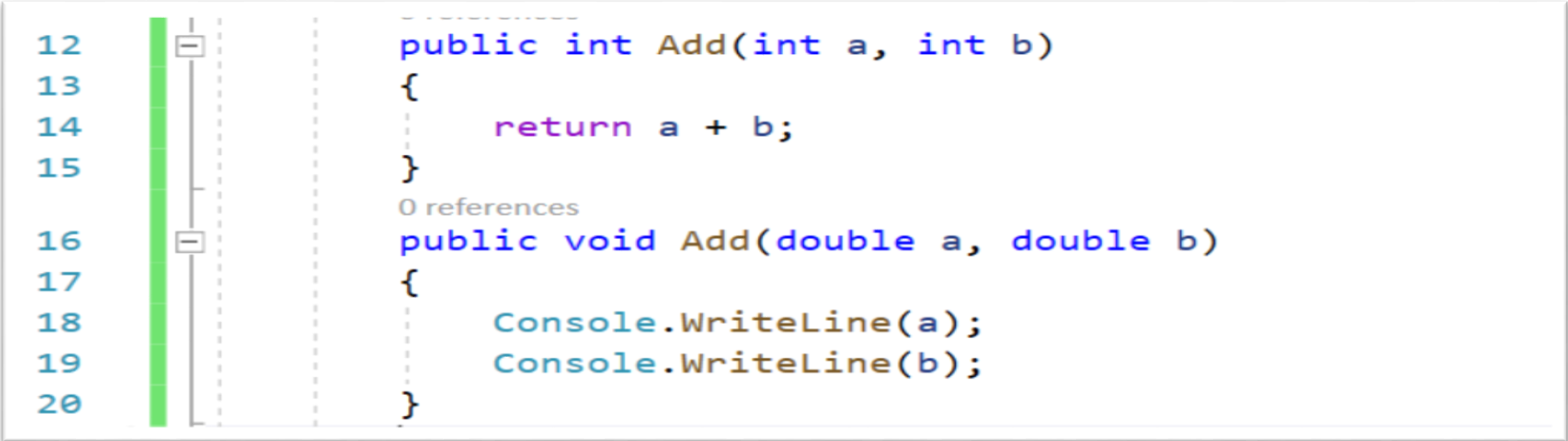
# The method

---

- ❑ Methods are functions that allow access to objects to simulate them, perform actions or verify behavior;

**modifier return-type method name ( arg1, ... ) { ... }**

- The returned type can correspond to an object
- If the method does not return anything, then void is used.



```
12 public int Add(int a, int b)
13 {
14     return a + b;
15 }
16 0 references
17 public void Add(double a, double b)
18 {
19     Console.WriteLine(a);
20     Console.WriteLine(b);
21 }
```

# The method

---

Modificateur	Rôle
public	la méthode est accessible aux méthodes des autres classes
private	l'usage de la méthode est réservé aux autres méthodes de la même classe
protected	la méthode ne peut être invoquée que par des méthodes de la classe ou de ses sous classes
const	la méthode ne peut être modifiée (redéfinition lors de l'héritage interdite)
abstract	Toutes les méthodes de cette classe abstract ne sont pas implémentées et devront être redéfinies par des méthodes complètes dans ses sous classes
static	la méthode appartient simultanément à tous les objets de la classe

# The method and attribute

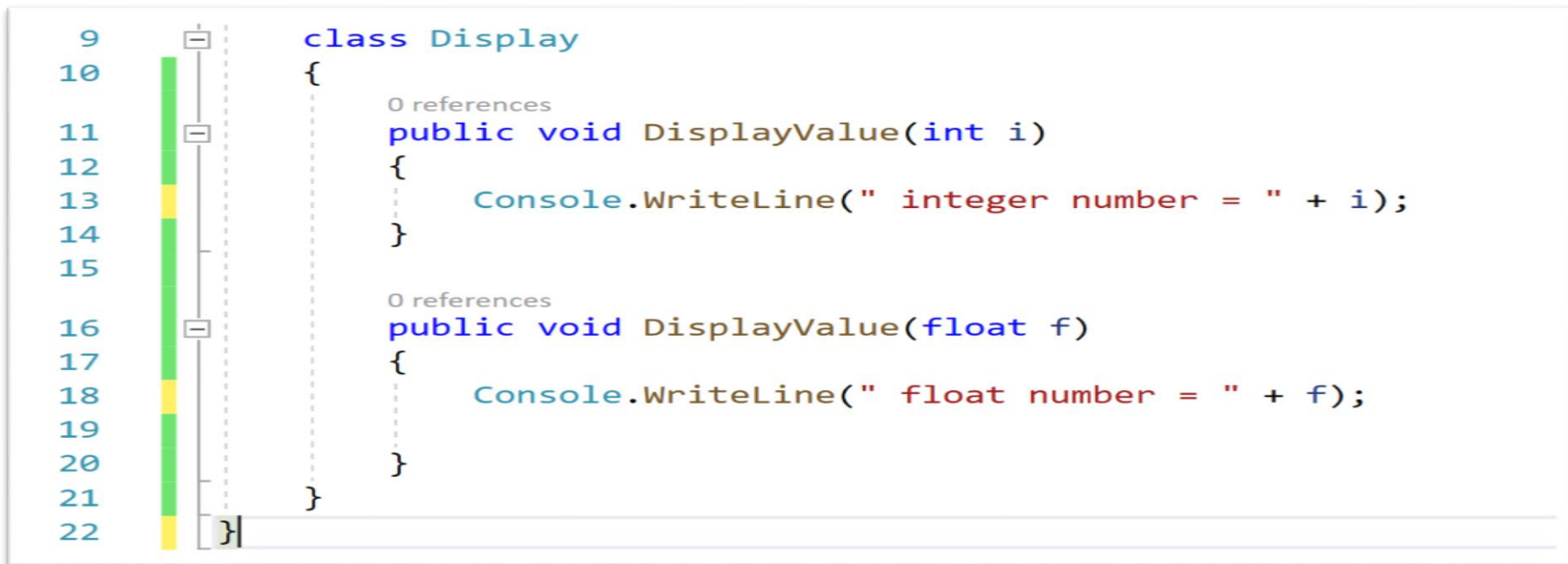
---

- ❑ There are Methods or attributes that does not depend on a particular object.
- ❑ They are called static methods or attributes: i.e. their declaration is preceded with the static keyword.
- **Example :**

```
58 1 reference  
59 public static void DisplayValue(float f)  
60 {  
61     Console.WriteLine("float number = " + f);  
62 }
```

# Methods overloading

- ❑ Overloading a method allows you to define the same method several times with different arguments.
- ❑ A method is overloaded when it performs different actions depending on the type and number of parameters passed.



```
9      class Display
10     {
11         0 references
12         public void DisplayValue(int i)
13         {
14             Console.WriteLine(" integer number = " + i);
15         }
16
17         0 references
18         public void DisplayValue(float f)
19         {
20             Console.WriteLine(" float number = " + f);
21         }
22     }
```

# The enumerations

---

- ❑ An enumeration is a data type whose value domain is a set of integer constants.
- ❑ Consider a program that manages the months of the year. There would be 7:  
January, February, ....
- ❑ We could then define an enumeration for these seven constants:

7  
8  
9  
10  
11  
12  
13  
14  
15  
16  
17

```
enum Months
{
    January,    // 0
    February,   // 1
    March,      // 2
    April,      // 3
    May,        // 4
    June,       // 5
    July        // 6
}
```

# The enumerations

---

- ❑ Internally, these seven constants are encoded by consecutive integers starting with 0 for the first constant, 1 for the next, etc.
- ❑ A variable can be declared as taking these values in the enumeration:
- **Examples :**

```
20  int myNum = (int)Months.April;  
21  Console.WriteLine(myNum);
```

# Passing a parameter to a function

---

- ❑ Parameters declared in a function's signature are called formal parameters.
- ❑ When we call a function, we pass it variables that will be copied into the formal parameters to perform the calculation. These are called effective(argument) parameters.

# Example

```
9  class Program
10 {
11     0 references
12     static void Main(string[] args)
13     {
14         int age = 20;
15         ChangInt(age);
16         Console.WriteLine("Argument parameter age =" + age);
17         Console.ReadKey();
18     }
19     1 reference
20     private static void ChangInt(int a)
21     {
22         a = 30;
23         Console.WriteLine("Formal parameter a= " + a);
24     }
25 }
26
```

Here **a** is a formal parameter

Here **age** is an effective parameter



# Passing by the values

---

- ❑ The value of the effective parameter is copied back to the corresponding formal parameter.
- ❑ We have two separate entities.
- ❑ If the function changes the formal parameter, the actual parameter is not modified in any way.

# Example

```
9  class Program
10 {
11     0 references
12     static void Main(string[] args)
13     {
14         int age = 20;
15         ChangInt(age);
16         Console.WriteLine("Argument parameter age =" + age);
17         Console.ReadKey();
18     }
19     1 reference
20     private static void ChangInt(int a)
21     {
22         a = 30;
23         Console.WriteLine("Formal parameter a= " + a);
24     }
25 }
26
```

```
Formal parameter a= 30
Argument parameter age= 20
```

# Passing by the references

---

- ❑ In a reference pass, the effective parameter and the formal parameter are one and the same entity.
- ❑ If the function changes the formal parameter, the actual parameter is also modified.
- ❑ In C#, they must both be preceded by the keyword **ref**.

# Example

```
9  - 0 references
10  class Program
11  {
12      0 references
13      static void Main(string[] args)
14      {
15          int age = 20;
16          ChangInt(ref age);
17          Console.Out.WriteLine("Argument parameter age= " + age);
18          Console.ReadKey();
19      }
20
21      1 reference
22      private static void ChangInt(ref int a)
23      {
24          a = 30;
25          Console.Out.WriteLine("Formal parameter a= " + a);
26      }
27  }
```

# Result

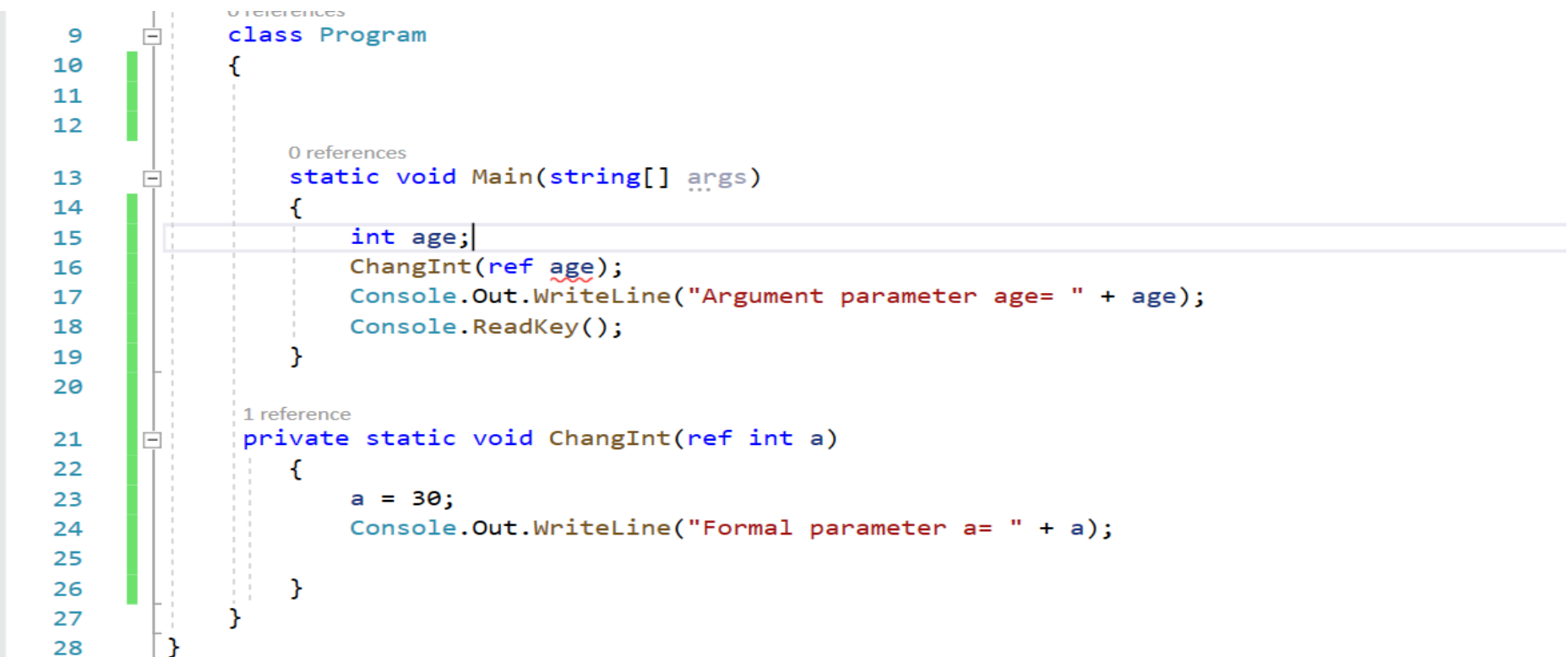
---

- ❑ The result of running the previous example is as follows:
- ❑ The effective parameter followed the change in the formal parameter.
- ❑ This passing mode is suitable for the output parameters of a function.

```
Formal parameter a= 30  
Argument parameter age= 30
```

# Passing by references with Out

- ❑ Consider the previous example in which the age variable would not be initialized before the function is called **changeInt** :



```
9  class Program
10 {
11
12
13     0 references
14     static void Main(string[] args)
15     {
16         int age;
17         ChangInt(ref age);
18         Console.Out.WriteLine("Argument parameter age= " + age);
19         Console.ReadKey();
20     }
21
22     1 reference
23     private static void ChangInt(ref int a)
24     {
25         a = 30;
26         Console.Out.WriteLine("Formal parameter a= " + a);
27     }
28 }
```

# Passing by references with Out

---

- ❑ When we compile this program, we have an error: Use of unassigned local variable 'age'
- ❑ We can solve the problem by assigning an initial value to age.
- ❑ You can also replace the **ref** keyword with the **out** keyword. We then express that the parameter is only an output parameter and therefore we do not need an initial value.

# Example

```
9  0 references
   class Program
10  {
11
12
13  0 references
   static void Main(string[] args)
14  {
15      int age;
16      ChangInt(out age);
17      Console.Out.WriteLine("Argument parameter age= " + age);
18      Console.ReadKey();
19  }
20
21  1 reference
   private static void ChangInt(out int a)
22  {
23      a = 30;
24      Console.Out.WriteLine("Formal parameter a= " + a);
25  }
26  }
27  }
28  }
```

```
Formal parameter a= 30
Argument parameter age= 30
```



# Remark

---

- Passing from an object to a function:
  - When an object is passed as a parameter to a function it is passed by reference and not by value.
  - This is because the objects are of the reference type. Therefore, there is no copy of the effective object parameter in the object formal parameter.

# Inheritance

- ❑ The purpose of inheritance is to "customize" an existing class to meet our needs.
- ❑ Suppose we want to create a **teacher** class: a teacher is a particular person.
- ❑ He has attributes that another person will not have: the subject he teaches for example.
- ❑ But it also has the attributes of any person: first name, last name and age.
- ❑ A teacher is therefore fully part of the person class but has additional attributes.
- ❑ Rather than writing a teaching class from base, we would prefer to take up the achievements of the person class that we would adapt to the particular character of the teachers. It is the concept of inheritance that allows us to do this.

# Problem

---

## ❑ Class Person

A person contains a name, first name, age and address

## ❑ Class Employee

An employee is a person who works in a company. Its specifications therefore take up those of the Person class. An employee has a company and a salary.

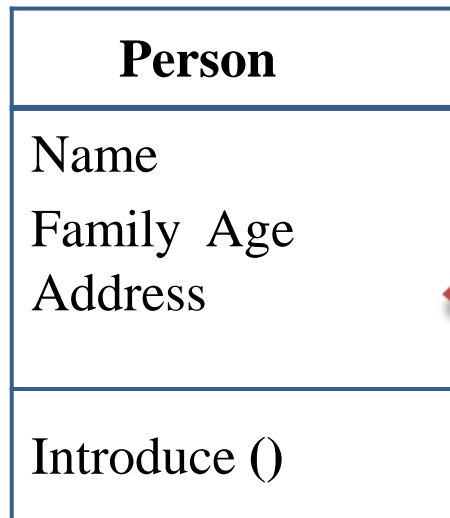
Person
Name
Family
Age
Address
Introduce ()

Employee
Name
Family
Age
Address
Society
salary
Introduce()

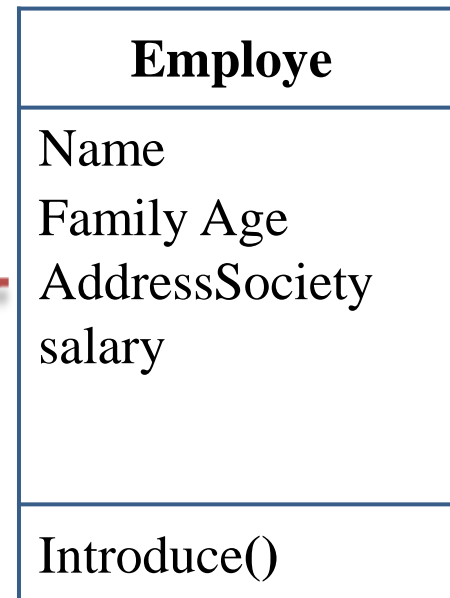
# Solution

---

**Base class**



**Derived class**

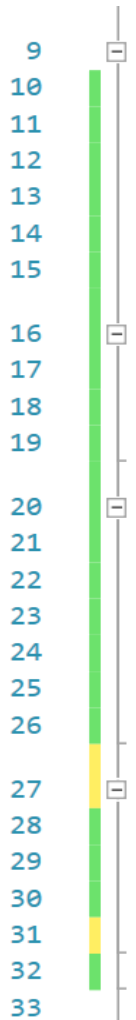


# Solution

---

- ❑ The notion of inheritance is one of the foundations of object-oriented programming. Thanks to it, we will be able to create inherited classes (also called derived classes) from our parent classes (also called base classes).
- ❑ `(:)` allows you to specify the name of the parent class. The `Employe` class inherits the properties and methods of the `Person` class.

# Example



```

9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33

4 references
class Person
{
    protected String name;
    protected String family;
    protected int age;
    protected String city;

    0 references
    public Person()
    {
        Console.WriteLine("From default constructor in person");
    }

    1 reference
    public Person(String name, String family, int age, String city)
    {
        this.name = name;
        this.family = family;
        this.age = age;
        this.city = city;
    }

    0 references
    public void Introduce()
    {
        Console.WriteLine("My name is " + name + " " + family +
            " I am " + age + " years old " + " I live in " + city);
    }
}

```

# Example

```
9  4 references
10  class Employee : Person
11  {
12      private string society;
13      private double salary;
14
15      1 reference
16      public Employee()
17      {
18      }
19
20      0 references
21      public Employee(String n, String f, int a, String c, String s, double sa) : base(n, f, a, c)
22      {
23          this.society = s;
24          this.salary = sa;
25      }
26  }
```

```
9  0 references
10  class Program
11  {
12      0 references
13      static void Main(string[] args)
14      {
15          Employee E = new Employee();
16          E.Introduce();
17          Console.ReadKey();
18      }
19  }
20 }
```

# Inheritance and constructors

---

- ❑ **Implicit call to the Constructor of Individual Employee**

**E= new Employee(); E.Introduce();**

- ❑ Object E appears as if it were a person because it inherits the introduce() method.

```
From default constructor in person  
My name is   I am 0 years old I live in
```

- ❑ Without having been specified in the constructor of Employee, the instance variable name has taken the value "null". Indeed, to build the instance of Employee, the compiler first calls the default constructor of the Person class:
- ❑ How to initialize the family, the first name , the age and the city of an employee?

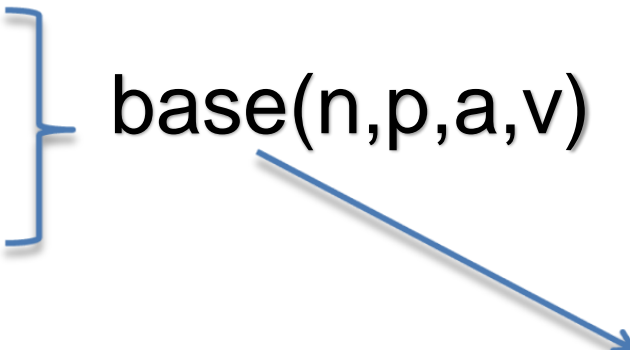


# Inheritance and constructors

---

- We can use the following constructor:

```
public Employee(String n, String f, int a, String c, String s, double sa){  
    this.name=n,  
    this.family=f;  
    this.age=a;  
    this.city=c;  
    this.society=s;  
    this.salary=sa;  
}
```



```
public Person(String name, String  
family, int age, String city){  
    this.name= name;  
    this.family= family;  
    this.age= age;  
    this.city= city;  
}
```

**Note: Base is the keyword for calling the base class (or parent class) constructor.**

# Inheritance and constructors

---

## ❑ The solution is:

```
18 0 references  
19 public Employee(String n, String f, int a, String c, String s, double sa) : base(n, f, a, c)  
20 {  
21     this.society = s;  
22     this.salary = sa;  
    }
```

## ❑ The main class:

```
16 Employee p = new Employee("name1", "family1", 32, "city1", "ECE", 200);  
17 p.Introduce();  
18
```

## ❑ The result is:

```
My name is  name1 family1 I am 32 years old  I live in city1
```

## Redefine the method: The polymorphisme

---

```
24 2 references
25 public void Introduce()
26 {
27     base.Introduce();
28     Console.WriteLine("I work in " + society + " My salary is " + salary);
29 }
```

❑ The result:

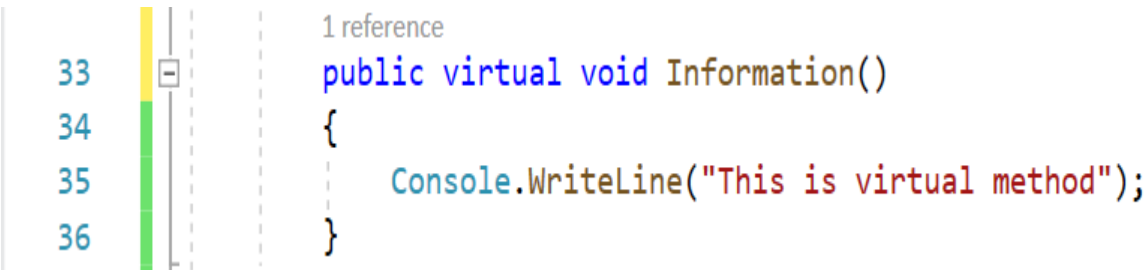
```
My name is name1 family1 I am 32 years old I live in city1
I work in ECE My salary is 200
```

# virtual, new and override

---

## ❑ Virtual as a keyword:

- Is used by parent class methods to allow derived classes to modify the method
- **Example :**



The screenshot shows a code editor with a line number column on the left (33, 34, 35, 36) and a vertical scrollbar. The code is a C# method definition. Line 33: `public virtual void Information()`. Line 34: `{`. Line 35:  `Console.WriteLine("This is virtual method");`. Line 36: `}`. A tooltip "1 reference" is visible above the `virtual` keyword on line 33.

```
33 public virtual void Information()  
34 {  
35     Console.WriteLine("This is virtual method");  
36 }
```

# virtual, new and override

---

- ❑ The redefinition of virtual methods in the derived class we use 2 keywords:
- ❑ **new**: tells the compiler that you are adding a method to a derived class with the same name as the method in the base class, but without any relationship between them
- **Example :**

```
34 0 references  
35 public new void Information()  
36 {  
37     Console.WriteLine("Redefined method by new");  
}
```

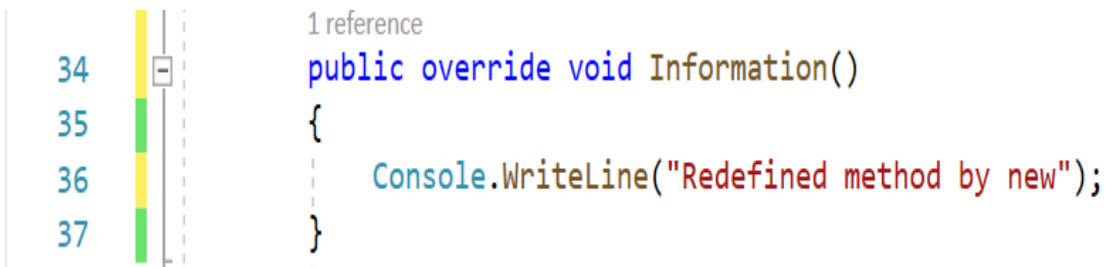
# virtual, new and override

---

- ❑ The redefinition of virtual methods in the derived class we use 2 keywords:

**override:** tells the compiler that the two methods are related

- **Example :**



```
1 reference
34 public override void Information()
35 {
36     Console.WriteLine("Redefined method by new");
37 }
```

# Inheritance summary

---

- ❑ A class can only inherit from one class!
- ❑ If no constructor is defined in a derived class, the compiler will create one and automatically call the constructor of the parent class.
- ❑ The derived class inherits all public and protected properties and methods from the parent class.
- ❑ The methods and private properties of a parent class are not accessible in the derived class.

# Inheritance summary

---

- ❑ Polymorphism corresponds to the possibility for an operator or a function to be usable in different contexts (distinguishable by the number and types of parameters) and to have a behavior adapted to these parameters.



# Inheritance summary

---

- ❑ If a method of a parent class is not redefined or polymorphed, when that method is called through a child object, the method of the parent class will be called!
- ❑ You cannot inherit from a class declared const. A method declared const is not redefinable.

# The abstract class

---

❑ an abstract class is like a normal class. That said, it still has a particularity:

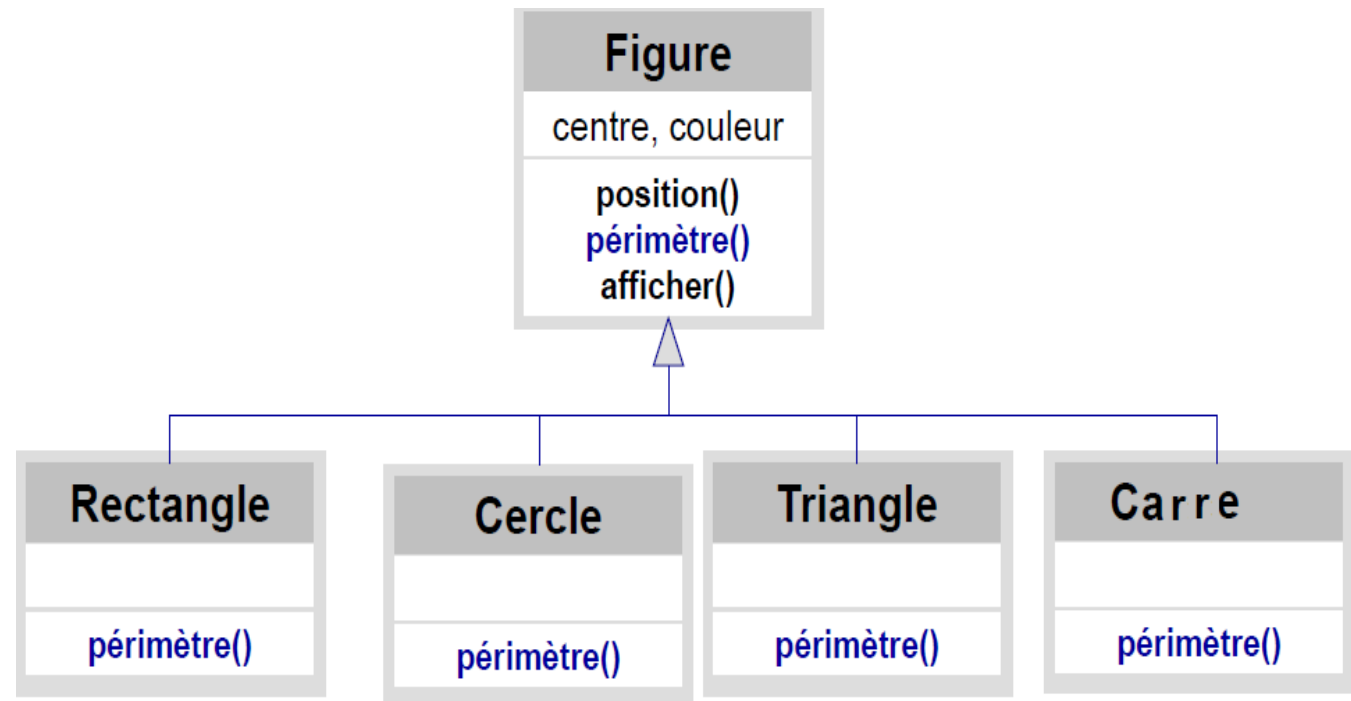
you can not instantiate it!

```
public class Test{  
    public static void main(String[] args){  
        A obj = new A();    //Erreur de compilation !!  
    }  
}
```

# The abstract class

- ❑ Imagine that you are making a program that manages different types of figures:
- ❑ In this program, you have:

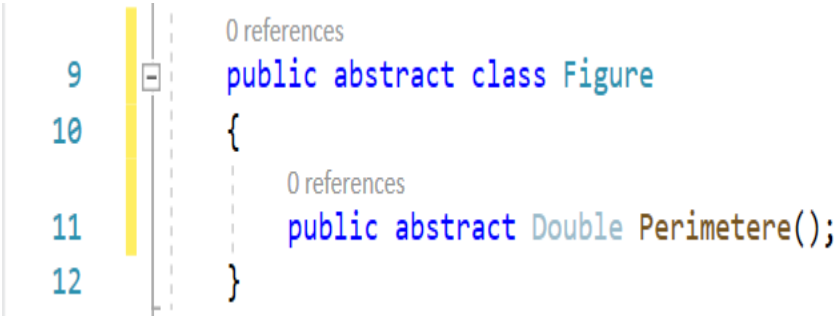
Circles;  
Rectangles  
Triangles  
squares



# Class abstract

---

- ❑ A class considered abstract. It must be declared with the keyword `abstract`.
- ❑ Such a class can have the same content as a normal class (attributes and methods). However, this type of class is used to define abstract methods. These methods have a particularity; they have no body!
- ❑ An abstract method can only exist in an abstract class



```
9  public abstract class Figure
10 {
11     public abstract Double Perimetre();
12 }
```

The screenshot shows a code editor with line numbers 9 to 12 on the left. The code defines an abstract class `Figure` with an abstract method `Perimetre()` that returns a `Double`. There are two "0 references" indicators above the class and method declarations. A yellow vertical bar is positioned between lines 9 and 10, and a dashed vertical line is between lines 10 and 11.

# Class abstract

```
9      4 references
10     abstract public class Figure
11     {
12         protected string center;
13         protected String color;
14
15         //constructor
16         0 references
17         public Figure()
18         {
19             }
20         1 reference
21         public Figure(string Cen, string Co)
22         {
23             center = Cen;
24             color = Co;
25         }
26         // abstract method
27         2 references
28         public abstract double Perimetere();
29
30         //normal methods
31         1 reference
32         public string Display() { return " The center is: " + center + " the color is: " + color; }
33         1 reference
34         public string Position() { return " The center is " + center; }
35     }
```

# Class cercle

```
9  4 references
10  class Cercle : Figure
11  {
12      private double r;
13      private const double p = 3.14;
14      //the constructor
15      0 references
16      public Cercle()
17      {
18      }
19      1 reference
20      public Cercle(string c, string co, double ra) : base(c, co)
21      {
22          this.r = ra;
23      }
24      //implementing abstract method
25      2 references
26      public override double Perimetre()
27      {
28          Console.WriteLine(" The perimetre is equal to : ");
29          return 2*p*r;
30      }
31      //the polymorphisme of method Display
32      1 reference
33      public string Display()
34      {
35          return base.Display() + " radius is :" + r;
36      }
37  }
```

# Class test

---

```
9      0 references  
      class Program  
10     {  
11         0 references  
        static void Main(string[] args)  
12     {  
13         Cercle c = new Cercle("C", "red", 12);  
14  
15         Console.WriteLine(c.Display());  
16         Console.WriteLine(c.Perimetre());  
17         Console.WriteLine(c.Position());  
18         Console.ReadKey();  
19     }  
20 }  
21
```

# The interfaces and multiple inheritance

---

- ❑ With multiple inheritance, a class can inherit multiple super classes at the same time.
- ❑ This mechanism does not exist in C#.
- ❑ Interfaces make it possible to implement an alternative mechanism.
- ❑ An interface is a set of abstract method declarations. All objects that implement this interface have the methods declared in it.
- ❑ Multiple interfaces can be implemented in the same class.
- ❑ Interfaces are declared with the interface keyword.



# The interfaces

---

- Classes can implement one or more interfaces
- They derive from it as a class
- Implementations of methods must be public
- Interfaces are defined with the keyword "interface"
- Interfaces can derive from one or more other interfaces

# The interfaces and multiple inheritance

---

❑ Declaring the interface:

```
Public interface InterfaceName {  
    // Insert here the abstract methods.
```

❑ Implementing the interface :

```
Modifier class ClassName : Interface1, Interface 2, ...{  
    //Insert here the methods and attributes  
}
```

# The delegates

---

- ❑ .NET encapsulates method pointers in objects called "delegates"
- ❑ Delegates allow a method to be invoked in a polymorphic way:
  - no need to know the class of the method;
  - Only the signature is important for initialization.
  - the invocation is done internally by Invoke.

# The delegates

---

- ❑ Once initialized, they can no longer be modified
- ❑ Need to rebuild a new delegate to change target methods and objects  
Performed transparently by the operators "=", "+=", "-="»

# The delegates

---

- ❑ A delegate is the .Net equivalent of a function pointer.
- ❑ Its role is to call one or more functions that may vary depending on the context.
- ❑ A delegate's declaration defines the signature of a method, the return type of which is preceded by the keyword `delegate`.
- ❑ **Example:** This declaration actually produces a `CompareDelegate` class derived from the `System` class. `Delegate`. It is therefore possible to place it outside of any class or namespace, as for a normal class.

```
7  public delegate void DisplayDelegate(string message);
```

# The delegates

---

## ❑ Instantiation

A variable of the delegate type can be declared, as with a normal class: Initially, this variable is null because it does not reference any method.

## ❑ Add and remove function

A delegate is associated with one or more methods that all have the same signature as the delegate. The addition or removal of function is done by the operators `=`, `+=` and `-=`

# Example

```
5 namespace DelegateEX
6 {
7     public delegate void DisplayDelegate(string message);
8     3 references
9     class Display
10    {
11        string name;
12        1 reference
13        public Display(string n)
14        {
15            this.name = n;
16        }
17        2 references
18        public void DisplayConsole1(string msg)
19        {
20            Console.WriteLine("Name:" + name + "-----Message:" + msg + "-----Methode:DisplayConnsole1");
21        }
22        1 reference
23        public void DisplayConsole2(string msg)
24        {
25            Console.WriteLine("Name:" + name + "-----Message:" + msg + "-----Methode:DisplayConnsole2");
26        }
27    }
28 }
```

# Example

```
5  0 references
   class Program
6  {
   0 references
7  static void Main(string[] args)
8  {
9      Display d = new Display("Name1");
10
11     //two type of using delegates
12     //DisplayDelegate displayDelegate = d.DisplayConsole1;
13     DisplayDelegate displayDelegate = new DisplayDelegate(d.DisplayConsole1);
14     displayDelegate += d.DisplayConsole2;
15     displayDelegate -= d.DisplayConsole1;
16     displayDelegate("message from delegate");
17     Console.ReadLine();
18 }
19 }
20 }
```



# Exceptions

---

- The handling of an exception is done according to the following :

```
try{  
  call of the function likely to generate the  
  exception  
}  
catch (Exception e){handle the exception e  
}  
Next statements
```

# Exceptions

---

- ❑ If the function does not generate an exception, then we move to the next statement, otherwise we move into the body of the catch clause and then to the next statement.
- ❑ Note the following: `e` is an object derived from the `Exception` type.
- ❑ We can be more precise by using types such as `IOException`, `SystemException`, etc. There are several types of exceptions
- ❑ By writing `catch (Exception e)`, we indicate that we want to manage all types of exceptions.

# Exceptions

---

- ❑ If the code of the try clause is likely to generate several types of exceptions, we may want to be more precise by managing the exception with several catch clauses:

```
try{  
    call of the function likely to generate the exception  
}  
  
catch (IOException e){  
    //handle the exception e  
}  
  
catch (SystemException e){  
    //handle the exception e  
}
```

# Exceptions

---

- We can add to the try/catch clauses, a finally clause: Whether there is an exception or not, the code of the finally clause will always be executed.

```
try{  
    //call of the function likely to generate the exception  
}  
  
catch (Exception e){handle the exception e  
}  
  
finally{  
    //code execute after try/catch  
}
```

# Exceptions

---

- ❑ The Exception class has a **Message** property that is a message detailing the error that occurred. So if we want to display this one, we will write:

```
catch (Exception ex){  
  
    Console.Error.WriteLine("The error : "+ex.Message);  
    ...  
} //catch
```

# Exceptions

---

- ❑ The Exception class has a **ToString** method that renders a character string indicating the type of the exception as well as the value of the Message property. We will be able to write:

```
catch (Exception ex){  
    Console.Error.WriteLine("The error : "+ex.ToString());  
    ...  
} //catch
```

# Collections

---

## **System.Collections Classes**

1. class ArrayList
2. class Hashtable <TKey,TValue>
3. class Queue
4. class Stack

# Collections

---

## **System.Collections.Generic Classes**

1. `class List<T>`
2. `class HashSet<T>`
3. `class LinkedList<T>`
4. `class Dictionary<TKey,TValue>`
5. `class Queue<TKey,TValue>`
6. `class Stack<TKey,TValue>`
7. `class SortedList<TKey,TValue>`



# Class ArrayList

---

- ❑ The ArrayList class implements a "dynamic array of objects."
- ❑ The size of such a list is automatically adjusted, if necessary, when elements are added to it, which is not the case for traditional array.
- ❑ A dynamic array, sometimes called a vector, can contain any object because it contains objects of a class derived from the Object class.
- ❑ In practice, a dynamic array typically contains objects of the same type
- ❑ The ArrayList class implements the IList interface, which itself implements the ICollection and IEnumerable interfaces. This means that the ArrayList class implements the properties and methods mentioned in the IList interface: Add, Clear, and so on..

# Class ArrayList

---

Méthode	Rôle
public void <b>add</b> ( <b>Object obj</b> )	allows you to add an object to the end of the list
<i>public int Count</i>	Count the number of elements
public void <b>Clear</b> ()	Delete all the elements in the list
<b>Void removeAt</b> ( <b>int index</b> )	delete the index position element. It returns the deleted item.
<b>Void remove</b> (object o)	Delete object o

# Class ArrayList

---

Méthode	Rôle
public bool <b>Contains</b> (object obj)	returns True if obj is in the list, False otherwise
public void <b>Insert</b> (object obj, int index)	inserts obj to the index position of the list

# Example

```
9      13 references
10     class Person
11     {
12         protected String name;
13         protected String family;
14         protected int age;
15         protected String city;
16
17         0 references
18         public Person()
19         {
20             Console.WriteLine("From default constructor in person");
21         }
22
23         5 references
24         public Person(String name, String family, int age, String city)
25         {
26             this.name = name;
27             this.family = family;
28             this.age = age;
29             this.city = city;
30         }
31
32         2 references
33         public void Introduce()
34         {
35             Console.WriteLine("My name is " + name + " " + family +
36                               " I am " + age + " years old " + " I live in " + city);
37         }
38     }
```

# Example

```
12 0 references
13 static void Main(string[] args)
14 {
15     // create 4 instances
16     Person p1 = new Person("name1", "family1", 11, "city1");
17     Person p2 = new Person("name2", "family2", 22, "city2");
18     Person p3 = new Person("name3", "family3", 33, "city3");
19     Person p4 = new Person("name4", "family4", 44, "city4");
20     // creating the list
21     ArrayList list = new ArrayList();
22
23     // fill the list
24     list.Add(p1);
25     list.Add(p2);
26     list.Add(p3);
27     list.Add(p4);
28
29     foreach (Person p in list)
30     {
31         p.Introduce();
32     }
33
34     Console.ReadKey();
```

# Class generic List<T>

---

- ❑ The System.Collections.Generic.List class<T> is used to implement collections of T-type objects that vary in size during program execution.
- ❑ A List object<T> is manipulated almost like an array.
- ❑ Thus element i of a list l is denoted l[i].
- ❑ For a List object<T> or T is a class, the list again stores the references of objects

# Example

```
12 0 references
13 static void Main(string[] args)
14 {
15     // create 4 instances
16     Person p1 = new Person("name1", "family1", 11, "city1");
17     Person p2 = new Person("name2", "family2", 22, "city2");
18     Person p3 = new Person("name3", "family3", 33, "city3");
19     Person p4 = new Person("name4", "family4", 44, "city4");
20     // creating the list
21     List<Person> list = new List<Person>();
22
23     // fill the list
24     list.Add(p1);
25     list.Add(p2);
26     list.Add(p3);
27     list.Add(p4);
28
29     foreach (Person p in list)
30     {
31         p.Introduce();
32     }
33
34     Console.ReadKey();
35 }
```

# Classe Dictionary<TKey,TValue>

---

- ❑ The `System.Collections.Generic.Dictionary class<TKey,TValue>` is used to implement a dictionary. A dictionary can be seen as a two-column table<TKey,TValue>:

Key	Value
key 1	value1

- ❑ In the Dictionary class<TKey,TValue> the keys are of type TKey, the values of type TValue.
- ❑ Keys are unique, i.e. there cannot be two identical keys.<TKey,TValue>



# La classe Dictionary<TKey,TValue>

Méthode	Rôle
public void <b>Add</b> (TKey key, TValue value)	add(key, value) to the dictionary
<i>public int Count</i>	Number of elements
public void <b>Clear</b> ()	Delete the elements
public bool <b>ContainsKey</b> (TKey key)	returns True if key is a dictionary key, False otherwise
public bool <b>ContainsValue</b> (TValue value)	returns True if value is a dictionary value, False otherwise
public bool <b>Remove</b> (TKey key)	removes the key pair from the dictionary. Makes True the operation succeeds, False otherwise.

# Class Dictionary<TKey,TValue>

```
12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
// references
static void Main(string[] args)
{
    // create 4 instances
    Person p1 = new Person("name1", "family1", 11, "city1");
    Person p2 = new Person("name2", "family2", 22, "city2");
    Person p3 = new Person("name3", "family3", 33, "city3");
    Person p4 = new Person("name4", "family4", 44, "city4");

    // create the dictionary
    Dictionary<int, Person> dic = new Dictionary<int, Person>();

    // fill the dictionary
    dic.Add(1, p1);
    dic.Add(2, p2);
    dic.Add(3, p3);
    dic.Add(4, p4);

    //show the values
    foreach (Person p in dic.Values)
    {
        p.Introduce();
    }

    //show the keys
    foreach (int k in dic.Keys)
    {
        Console.WriteLine("the key is : " + k + " the value is : " );
        dic[k].Introduce();
    }
}
```

# The serialization

---

- ❑ Serialization is a process of saving the state of an object on disk or network rather than keeping it in memory.
- ❑ We can say that the object is "flattened" to be able to convert it into a stream of data, which can be transmitted to another object via deserialization
- ❑ Serialization and deserialization is done via files (binary, XML,...).

# Serialize an object (XML)

---

- ❑ We create the class Person, to be able to store the object in a file it must be serializable.

```
7  using System.Xml.Serialization;
8
9  namespace XmlSerialization
10 {
11     [Serializable]
12     class Person
13     {
14         string name;
15         public string Name
16         {
17             get { return name; }
18             set { name = value; }
19         }
20         string family;
21         public string Family
22         {
23             get { return family; }
24             set { family = value; }
25         }
26         string tel;
27         public string Tel
28         {
29             get { return tel; }
30             set { tel = value; }
31         }
32     }
33 }
```

# Serialize an object (XML)

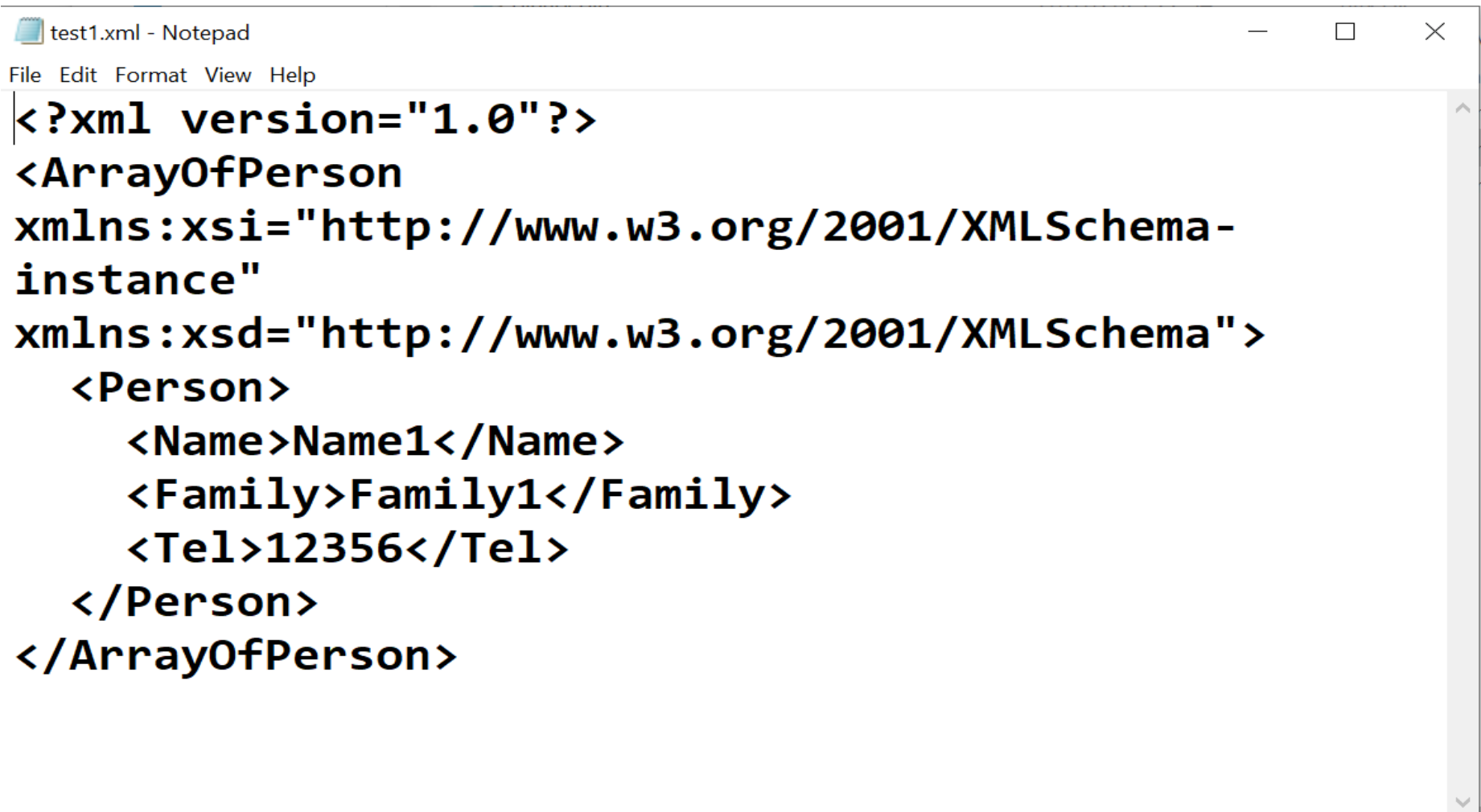
---

```
32 // The list
33 public static List<Person> phonebook = new List<Person>();
34 // serialize the object :
35 1 reference
36 public static void Save(string path)
37 {
38     FileStream f = File.Open(path, FileMode.Open);
39     XmlSerializer s = new XmlSerializer(typeof(List<Person>));
40     s.Serialize(f, phonebook);
41     f.Close();
}
```

# Serialize an object (XML)

```
9      0 references
10     class Program
11     {
12         0 references
13         static void Main(string[] args)
14         {
15             Person p = new Person();
16             p.Name = "Name1";
17             p.Family = "Family1";
18             p.Tel = "12356";
19
20             // Add the object to the list
21             Person.phonebook.Add(p);
22             string path = @"C:\test1.xml";
23             // save in test file
24             Person.Save(path);
25             Console.ReadKey();
26         }
27     }
```

# Serialize an object (XML)

A screenshot of a Notepad window titled "test1.xml - Notepad". The window has a menu bar with "File", "Edit", "Format", "View", and "Help". The text area contains XML code for serializing an array of person objects. The code is as follows:

```
<?xml version="1.0"?>
<ArrayOfPerson
xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Person>
    <Name>Name1</Name>
    <Family>Family1</Family>
    <Tel>12356</Tel>
  </Person>
</ArrayOfPerson>
```

# Deserialize an object

---

```
1 reference
43 public static List<Person> Charge(string path)
44 {
45     FileStream f = File.Open(path, FileMode.Open);
46     XmlSerializer s = new XmlSerializer(typeof(List<Person>));
47     List<Person> lis = (List<Person>)
48     s.Deserialize(f);
49     f.Close();
50     return lis;
51 }
52
```



# Deserialize an object

---

```
26 List<Person> k = Person.Charge(path);  
27  
28 foreach (Person s in k)  
29 {  
30     System.Console.WriteLine(s.Name + " " +  
31     s.Family + " " + s.Tel);  
32 }
```

```
Name1 Family1 12356
```

# Serialize an object (BIN)

- ❑ We create the class Person, to be able to store the object in a file it must be serializable.

```
7  using System.Xml.Serialization;
8
9  namespace XmlSerialization
10 {
11     [Serializable]
12     class Person
13     {
14         string name;
15         public string Name
16         {
17             get { return name; }
18             set { name = value; }
19         }
20         string family;
21         public string Family
22         {
23             get { return family; }
24             set { family = value; }
25         }
26         string tel;
27         public string Tel
28         {
29             get { return tel; }
30             set { tel = value; }
31         }
32     }
33 }
```

# Serialize and deserialize an object (BIN)

```
33
34 // serialize object in binary :
35 public static List<Person> phonebook = new List<Person>();
36 1 reference public static void SaveBin(string path)
37 {
38
39     FileStream f = File.Open(path, FileMode.OpenOrCreate);
40     IFormatter s = new BinaryFormatter();
41     s.Serialize(f, phonebook);
42     f.Close();
43 }
44 // deserialize object in binary :
45 1 reference public static List<Person> Charge(string path)
46 {
47     FileStream f = File.Open(path, FileMode.Open, FileAccess.ReadWrite);
48     IFormatter s = new BinaryFormatter();
49     List<Person> lo = (List<Person>)
50     s.Deserialize(f);
51     f.Close();
52     return lo;
53 }
54
55 }
```

# Serialize and deserialize an object (BIN)

```
9  0 references
10  class Program
11  {
12      0 references
13      static void Main(string[] args)
14      {
15          string path = @"C:\binary.bin";
16          Person p = new Person();
17          p.Name = "Name1";
18          p.Family = "Family1";
19          p.Tel = "123456";
20
21          // add a person to the list
22          Person.phonebook.Add(p);
23          // save in the file
24          Person.SaveBin(path);
25
26          // create list and fill it by the file
27          List<Person> k = Person.Charge(path);
28
29          foreach (Person s in k)
30          {
31              System.Console.WriteLine(s.Name + " " + s.Family + " " + s.Tel);
32          }
33      }
```

# Lambda expressions

---

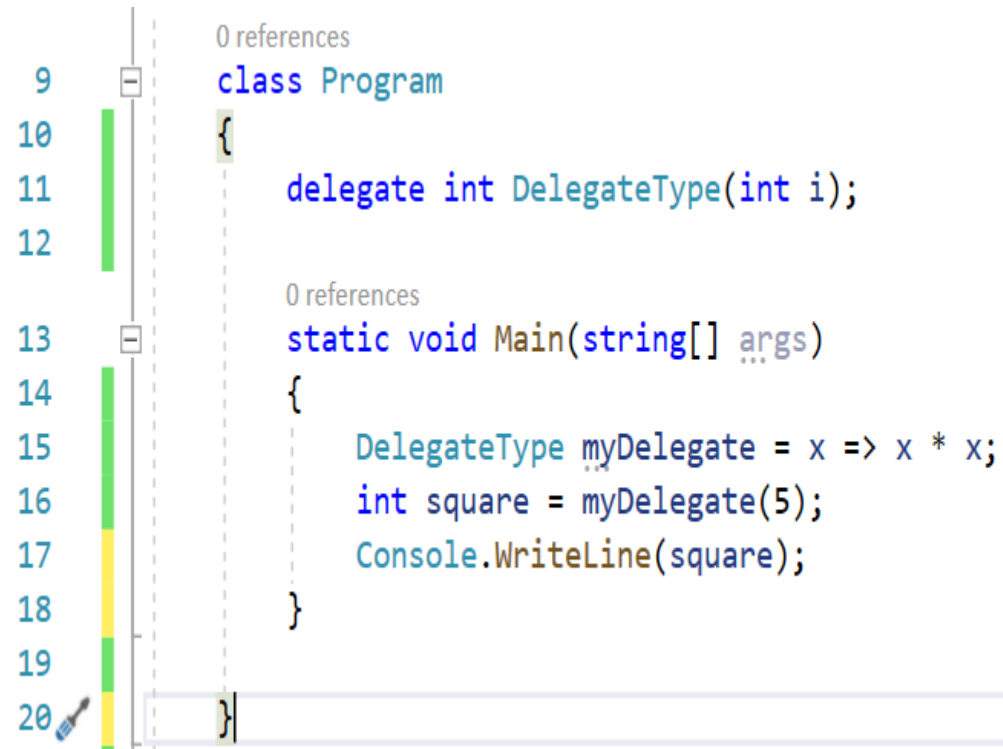
- ❑ Lambda expressions were introduced in C#3 (.NET 3.5).
- ❑ These expressions are now used throughout the code, to make LINQ queries, to filter lists, for delegates and events.
- ❑ It is therefore essential to understand how this type of expression works.

# Lambda expressions

---

- ❑ A lambda expression is an anonymous function that can contain expressions and statements.
- ❑ All expressions use the lambda => operator (reads "leads to").
- ❑ An expression is always made up of two parts:
  - the left side gives the input parameters (if any),
  - the right side gives the instructions for the anonymous method.

# Example



```
9 0 references  
10 class Program  
11 {  
12     delegate int DelegateType(int i);  
13  
14     0 references  
15     static void Main(string[] args)  
16     {  
17         DelegateType myDelegate = x => x * x;  
18         int square = myDelegate(5);  
19         Console.WriteLine(square);  
20     }
```

# Lambda expressions

---

- ❑ The framework offers a lot of extension methods since the appearance of Linq.
- ❑ The most used is probably **Count** which allows you to return the number of items in a collection.
- ❑ This method supports several overloads, including one that provides a parameter of type Func, which allows you to specify a filter expression.
- ❑ Thanks to this mechanism, it is possible to count certain elements in particular.



# Example

---

```
9  0 references
10  class Program
11  {
12      0 references
13      static void Main(string[] args)
14      {
15          int[] nombres = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
16
17          // count the number of pair elements
18          int pairs = nombres.Count(n => n % 2 == 0);
19          Console.WriteLine(pairs);
20
21          // count the average
22          int avg = (int)nombres.Average(n =>n);
23          Console.WriteLine(avg);
24
25          Console.ReadKey();
26      }
27  }
```