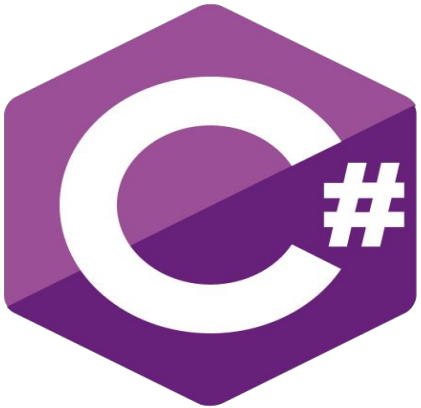
epsi
l'école d'ingénierie informatique

Pro Alterna®



Fondamentaux

Programme de formation Socle Numérique 2ème année

Les Tableaux

Les tableaux permettent de regrouper plusieurs valeurs dans une seule variable, évitant ainsi de déclarer des variables distinctes pour chaque valeur.

Pour déclarer un tableau, définissez le type de variable entre crochets :

```
string[] Color
```

Les Tableaux

Les tableaux permettent de regrouper plusieurs valeurs dans une seule variable, évitant ainsi de déclarer des variables distinctes pour chaque valeur.

Pour déclarer un tableau, définissez le type de variable entre crochets :

```
string[] Color
```

Maintenant, nous avons créé une variable qui stocke un tableau de chaînes. Pour y ajouter des valeurs, nous pouvons utiliser un tableau littéral en plaçant les valeurs dans une liste séparée par des virgules et entourées d'accolades.

```
string[] Color = {"Red", "Bleu", "Black", "Green"};
```

Pour créer un tableau d'entiers, vous pouvez écrire :

```
int[] Num = {15, 25, 35, 45};
```

Accéder aux éléments d'un tableau

Pour accéder à un élément spécifique du tableau, vous utilisez son index. Cette instruction récupère la valeur du premier élément du tableau “Color”".

```
string[] Color = {"Red", "Bleu", "Black", "Green"};
Console.WriteLine(color[0]);

// Outputs Red
```

Changer un élément de tableau

Pour modifier la valeur d'un élément spécifique, faut juste indiquer le numéro d'index :

```
string[] Color = {"Red", "Bleu", "Black", "Green"};
Color[0] = "White";
Console.WriteLine(Color[0]);

// Now outputs White
```

Changer un élément de tableau

Pour connaître le nombre d'éléments d'un tableau, utilisez Length propriété :

```
string[] Color = {"Red", "Bleu", "Black", "Green"};
Console.WriteLine(Color.Length);

// Outputs 4
```

Parcourir les tableaux en utilisant les boucles en C#

Vous pouvez itérer à travers les éléments d'un tableau en utilisant une boucle et en utilisant la propriété Length pour déterminer le nombre d'itérations.

L'exemple suivant illustre comment afficher tous les éléments du tableau Color.

```
string[] Color = {"Red", "Bleu", "Black", "Green"};
for (int i = 0; i < Color.Length; i++)
{
  Console.WriteLine(Color[i]);

}
```

Parcourir les tableaux en utilisant les boucles en C#

Il existe également une foreach boucle, qui sert exclusivement à parcourir les éléments d'un tableau :

```
string[] colors = { "Red", "Bleu", "Black", "Green" };

foreach (string i in color)
{
   Console.WriteLine(i);
}
```

Trier un tableau

Il existe de nombreuses méthodes de tableau disponibles, par exemple Sort(), qui trie un tableau par ordre alphabétique ou croissant :

Exemple:

```
// Sort a string

string[] Color = {"Red", "Blue", "Black", "Green"};
Array.Sort(Color);
foreach (string i in Color)
{
  Console.WriteLine(i);

}

// output  Black,Blue,Green,Red

// pour inverser le tableau

Array.Reverse(Color);
foreach (string i in Color)
{
  Console.WriteLine(i);

}

// output Green,Black,Blue,Red
```

Trier un tableau

```
// Sort un int
int[] Numbers = {9, 15, 3, 5};
Array.Sort(Numbers);
foreach (int i in Numbers)
{
  Console.WriteLine(i);
}

//output 3,5,9,15,

//Tri en ordre décroissant

        Array.Sort(Numbers, (x, y) => y.CompareTo(x));
```

Trier un tableau

D'autres méthodes pratiques pour les tableaux, comme Min, Max et Sum, sont disponibles dans l'espace de noms System.Linq.

```
using System;
using System.Linq;
namespace MyApplication
{
  class Program
  {
    static void Main(string[] args)
    {
      int[] myNumbers = {7, 2, 16, 9};
      Console.WriteLine(myNumbers.Max());  // returns the largest value
      Console.WriteLine(myNumbers.Min());  // returns the smallest value
      Console.WriteLine(myNumbers.Sum());  // returns the sum of elements
    }
  }

}

// Output Min=2, Max=16, Sum=34
```

Tableaux multidimensionnels

- Dans le chapitre précédent, vous avez appris à utiliser les tableaux, également connus sous le nom de tableaux à une dimension. Ils sont très utiles et seront fréquemment utilisés dans vos programmes en C#. Cependant, si vous avez besoin de stocker des données sous forme de tableau, par exemple un tableau avec des lignes et des colonnes, vous devrez vous familiariser avec les tableaux multidimensionnels.
- Un tableau multidimensionnel est essentiellement un tableau de tableaux. Ils peuvent avoir n'importe quel nombre de dimensions, mais les plus courants sont les tableaux bidimensionnels (2D).

Tableaux bidimensionnels

Pour créer un tableau 2D en C#, vous pouvez ajouter chaque tableau individuel à l'intérieur de son propre ensemble d'accolades, en insérant une virgule (,) entre les crochets. Cela crée une structure de tableau avec des lignes et des colonnes.

Exemple:

```
int[,] numbers = { {2, 5, 3}, {5, 4, 3} };
```

Accès aux Éléments d'un Tableau à Deux Dimensions

Pour accéder à un élément d'un tableau à deux dimensions en C#, vous devez spécifier deux index : un pour la rangée du tableau et un pour la colonne de l'élément à l'intérieur de cette rangée.

Exemple:

```
int[,] numbers = { {2, 5, 3}, {5, 4, 3} };

Console.WriteLine(numbers[0, 2]);  // Outputs 3

Console.WriteLine(numbers[1, 1]);  // Outputs 4
```

Accès aux Éléments d'un Tableau à Deux Dimensions

Des modifications peuvent également être apportées au tableau.

Exemple:

```
int[,] numbers = { {2, 5, 3}, {5, 4, 3} };
numbers[1, 0] = 7;  // Change value to 7

Console.WriteLine(numbers[1, 0]); // Outputs 7 instead of 5
```

Parcourir les tableaux 2D en utilisant les boucles en C#

Foreach boucle :

```
int[,] numbers = { {2, 5, 3}, {5, 4, 3} };

foreach (int i in numbers)
{
  Console.WriteLine(i);

}
```

Parcourir les tableaux 2D en utilisant les boucles en C#

Boucle For:Il est également important de noter que nous devons utiliser GetLength() au lieu de Length pour déterminer le nombre d'itérations de la boucle.

```
int[,] numbers = { {2, 5, 3}, {5, 4, 3} };

for (int i = 0; i < numbers.GetLength(0); i++)
{
  for (int j = 0; j < numbers.GetLength(1); j++)
  {
    Console.WriteLine(numbers[i, j]);
  }

}
```

Saisie des éléments dans un tableau

```
Console.WriteLine("Entrez la taille du tableau : ");
int size = Convert.ToInt32(Console.ReadLine());
int[] numbers = new int[size];

for (int i = 0; i < size; i++)
{
    Console.WriteLine($"Entrez l'élément {i + 1} : ");
    numbers[i] = Convert.ToInt32(Console.ReadLine());
}
```

Les Listes

En C#, les listes sont une structure de données dynamique très utile qui permet de stocker et de manipuler des collections d'objets. Contrairement aux tableaux, les listes sont redimensionnables, ce qui signifie que leur taille peut changer dynamiquement pendant l'exécution du programme.

Déclaration et Initialisation

Pour utiliser des listes en C#, vous devez d'abord importer l'espace de noms System.Collections.Generic. Ensuite, vous pouvez déclarer et initialiser une liste comme suit :

```
using System.Collections.Generic;

List<int> numbers = new List<int>(); // Création d'une liste vide d'entiers
List<string> names = new List<string>() { "Alice", "Bob", "Charlie" }; // Création d'une liste avec des éléments
```

Ajout et Suppression d'Éléments

Les listes offrent des méthodes pratiques pour ajouter et supprimer des éléments :

```
// Ajout d'éléments à la fin de la liste
numbers.Add(10);
numbers.Add(20);

// Suppression d'un élément par valeur
numbers.Remove(10);

// Suppression d'un élément à une position spécifique
numbers.RemoveAt(0); // Supprime le premier élément

// Suppression de tous les éléments de la liste
numbers.Clear();
```

Propriétés et Méthodes Utiles

Les listes offrent plusieurs propriétés et méthodes utiles pour travailler avec les données :

- **Count** : Renvoie le nombre d'éléments dans la liste.

  nomList.Count

- **Capacity** : Renvoie le nombre total d'éléments que la liste peut contenir sans avoir besoin de redimensionner sa capacité interne.

  nomList.Capacity

- **Contains()** Vérifie si un élément spécifique est présent dans la liste.

  nomList.Contains(element) :

- **IndexOf()** : Renvoie l'index de la première occurrence d'un élément spécifique dans la liste.

  nomList.IndexOf(element);

POO

- La POO est l'acronyme de Programmation Orientée Objet.

- La programmation procédurale implique la création de procédures ou de méthodes qui effectuent des opérations sur les données, tandis que la programmation orientée objet implique la création d'objets qui contiennent à la fois des données et des méthodes.

Les classes et les objets

Les classes et les objets sont les deux principaux aspects de la programmation orientée objet.

Exemple :

Class	Objet
Couleur	Rouge Noir Bleu

- Une classe définit un modèle pour les objets, tandis qu'un objet représente une instance concrète de cette classe.
- Lorsqu'un objet est instancié à partir d'une classe, il hérite de toutes les propriétés et méthodes définies dans cette classe.
- Les membres ou champs d'une classe peuvent être des données (attributs), des méthodes (fonctions), des propriétés.

Modificateur

- **Privé** (private) n'est accessible que par les seules méthodes internes de la classe
- **Public** (public) est accessible par toute fonction définie ou non au sein de la classe
- **Protégé** (protected) n'est accessible que par les seules méthodes internes de la classe ou d'un objet dérivé
- **Final** (Final) ne peut pas etre modifiée
- **Abstrait** (abstract) contient une ou des méthodes abstraits, qui n’ont pas de définition explicite

Classes et objets

une classe appelée CompteBancaire qui représente un compte bancaire. Chaque compte bancaire serait un objet de cette classe. Les comptes bancaires auraient des attributs tels que le numéro de compte, le solde et le titulaire du compte, ainsi que des méthodes telles que Déposer(), Retirer() et VérifierSolde(). Cela nous permettrait de gérer et de manipuler facilement les données des comptes bancaires dans notre programme.

Créer une classe

Créez une classe nommée " CarteBancaire" avec une variable Titulaire

```
class CarteBancaire
{
  string Titulaire = "Alex";
}
```

Un objet est instancié à partir d'une classe. Ayant déjà défini la classe nommée "CarteBancaire", nous sommes maintenant en mesure de l'utiliser pour créer des instances.

Créer un objet

Créez un objet appelé " maCarte" et utilisez-le pour imprimer la valeur de Titulaire :

Céer un objet de CarteBancaire, précisez le nom de la classe, suivi du nom de l'objet, et utilisez le mot-clé new:

```
class CarteBancaire
{
   string titulaire = "Alex";

   static void Main(string[] args)
   {
      CarteBancaire maCarte = new CarteBancaire();
      Console.WriteLine(maCarte.titulaire);
   }
}
// Output: Alex
```

Plusieurs classes et objets

Créez deux objets de CarteBancaire

```
class CarteBancaire
{
    string titulaire = "Alex";

    static void Main(string[] args)
    {
        CarteBancaire maCarte1 = new CarteBancaire();
        CarteBancaire maCarte2 = new CarteBancaire();
        Console.WriteLine(maCarte1.titulaire);
        Console.WriteLine(maCarte2.titulaire);
    }
}
```

Plusieurs classes

Vous pouvez créer un objet d'une classe et y accéder dans une autre classe, ce qui est couramment utilisé pour organiser les classes de manière plus structurée. Une classe peut contenir tous les champs et méthodes, tandis qu'une autre classe peut contenir la méthode Main() on a /

PROG1

```
class CarteBancaire
{
  public string Titulaire = "Alex";

}
```

PROG2

```
class Program
{
  static void Main(string[] args)
  {
    CarteBancaire maCarte = new
CarteBancaire();
    Console.WriteLine(maCarte.titulaire);
  }

}
```

Constructeur d’une classe

- Un constructeur est une méthode spéciale qui sert à initialiser un objet lors de sa création.
- Il porte toujours le nom de la classe pour laquelle il est définie.
- Il est public et n’as pas de type de retour
- Une classe peut avoir un ou plusieurs constructeurs.
- **Sans constructeur** : On doit affecter les valeurs après avoir créé l’objet.
- **Avec un constructeur** : Les valeurs sont déjà attribuées dès la création de l’objet.

Types de constructeurs

- **Par défaut** : pour la création puis l'initialisation d'un objet dans le cas ou le programmeur omet de donner des valeurs.
- **Paramétré**: pour la création puis l'initialisation d'un objet avec des valeurs données par le programmeur .
- **Par recopie**: pour la création puis l'initialisation d'un objet en copiant les valeurs d'un autre objet.

Types de constructeurs

```
13      // Constructeur par default
          0 references
14      public Person(){
15        nom="";
16        prenom="";
17        age=0;
18      }
19      // Constructeur Paramétré
          0 references
20      public Person(string nom1,string prenom1, int age1){
21       this.nom=nom1;
22       this.prenom=prenom1;
23       this.age=age1;
24      }
25
26      // Constructeur par ecopie
          0 references
27      public Person(Person p){
28       this.nom=p.nom;
29       this.prenom=p.prenom;
30       this.age=p.age;
31      }
32
```

Types de constructeurs

```
//par default
Person p1=new Person();
p1.nom="alex";
p1.prenom="alex1";
p1.age=25;
Console.Write($"nom :{p1.nom},  prenom : {p1.prenom}, age : {p1.age} \n");
//Paramétré

 Person p2 = new Person("alex2","alex3",20 );

 Console.Write($"nom :{p2.nom},  prenom : {p2.prenom}, age : {p2.age} \n");
 //par ecopie

 Person p3 = new Person(p2);
  Console.Write($"nom :{p3.nom},  prenom : {p3.prenom}, age : {p3.age} ");
```

Avec constructeur

```
70
    3 références
71  class Voiture
72      {
73          public string marque;
74          public int annee;
75
76          // Constructeur qui initialise les valeurs
            1 référence
77          public Voiture(string marque, int annee)
78          {
79              this.marque = marque;
80              this.annee = annee;
81          }
82      }
83
        0 références
84      class Program
85      {
            0 références
86          static void Main()
87          {
88              // Création d'un objet avec le constructeur
89              Voiture v = new Voiture("Toyota", 2022);
90
91              Console.WriteLine($"Marque : {v.marque}, Année : {v.annee}");
92          }
```

Constructeur par défaut

```
using System;

0 références
class Voiture
{
    public string marque;
    public int annee;
}

1 référence
class Program
{
    0 références
    static void Main()
    {
        // Création d'un objet Voiture avec le constructeur par défaut implicite
        Voiture v = new Voiture();

        // Affectation manuelle des valeurs
        v.marque = "Toyota";
        v.annee = 2022;

        Console.WriteLine($"Marque : {v.marque}, Année : {v.annee}");
    }
}
```

Getters

```
public string GetNom()
  {
     return nom;
  }

  // Méthode de getter pour obtenir l'âge
  public int GetAge()
  {
     return age;
  }
```

Setters

```
public void SetNom(string nouveauNom)
   {
      nom = nouveauNom;
   }

// Méthode de setter pour définir l'âge
public void SetAge(int nouvelAge)
   { age=nouvelAge;
   }
```

L’ENCAPSULATION

- C’est le regroupement dans une même entité appelée objet des données.
- On ne peut atteindre les attributs d’un objet que par l’intermédiaire de ses méthodes ou services
- Les accesseurs et mutateurs : Un accesseur est une méthode qui va nous permettre d'accéder aux variables des objets en lecture et un mutateur, en écriture .
- On parle de Getters et de Setters.
- Les Getters sont du même type que la variable qu'ils doivent retourner
- Les Setters sont, par contre, de type void. ces méthodes ne retournent aucune valeur, elles se contentent de les mettre à jour

Les propriétés

- L’expérience de la programmation nous as enseigné que pour chaque attribut privé d’une classe il faut prévoir deux méthodes pour la lecture et pour la modification de cet attribut.
- On appelle ces méthodes des getter/setter.
- Ainsi pour l’attribut name de la classe person par exemple il faut prévoir les méthodes

Les propriétés

- Les propriétés permettent d'éviter la lourdeur de ces méthodes accesseurs/modifieurs.
- Elles permettent de manipuler des attributs privés comme s'ils étaient publics.
  - Leurs forme est comme suit:

public type propriété
{
get {...}
set {...}
}
	public String Nom
	{
	get {return nom;}
	set {nom=value;}
	}

Association de classes

Les associations entre les classes en programmation orientée objet se produisent lorsqu'une classe utilise une autre classe ou à une référence à une autre classe.

Il existe différents types d'associations, tels que l'agrégation, la composition, l'héritage.

Agrégation

L'agrégation est une relation "a un" faible. Cela signifie qu'un objet de classe peut exister indépendamment de l'autre, et la relation n'est pas exclusive. Par exemple, une voiture a un moteur. Si la voiture est détruite, le moteur peut toujours exister

Agrégation

```
public class Moteur
{
    // Propriétés et méthodes du moteur
}

public class Voiture
{
    public Moteur M_oteurType ;
    public Moteur M_oteurNom ;
    // Autres propriétés et méthodes de la
voiture
}
```

```
Moteur moteur = new Moteur();

Voiture voiture = new Voiture();

voiture.M_oteurNom = moteur; //

Association entre la voiture et le moteur
```

Ce que fait cette ligne :
Elle **associe** l'objet `moteur` à l'attribut `M_oteurNom` de `voiture`.
Cela signifie que la voiture a maintenant une **référence** vers l'objet moteur.

Composition

La composition est une relation "a un" forte. Cela signifie qu'un objet de classe est composé d'un autre objet de classe, et la relation est exclusive. Si l'objet parent est détruit, l'objet enfant est également détruit. Par exemple, une maison est composée de pièces. Si la maison est détruite, les pièces sont également détruites.

Composition

```
public class Piece
{
    // Propriétés et méthodes de la pièce
}
public class Maison
{
    public Piece Cuisine ;
    public Piece Salon ;
    // Autres propriétés et méthodes de la maison
}
```

```
Piece cuisine = new Piece();
Maison maison = new Maison();
maison.Cuisine = cuisine; //
Association entre la maison et la
cuisine
```

L’héritage

- Le but de l'héritage est de "personnaliser" une classe existante pour qu'elle satisfasse à nos besoins.
- Supposons qu'on veuille créer une classe enseignant : un enseignant est une personne particulière.
- Il a des attributs qu'une autre personne n'aura pas : la matière qu'il enseigne par exemple. Mais il a aussi les attributs de toute personne :prénom, nom et âge.
- Un enseignant fait donc pleinement partie de la classe personne mais a des attributs supplémentaires.
- Plutôt que d'écrire une classe enseignant à partir de rien, on reprend l'acquis de la classe personne qu'on adaptait au caractère particulier des enseignants.

L’héritage

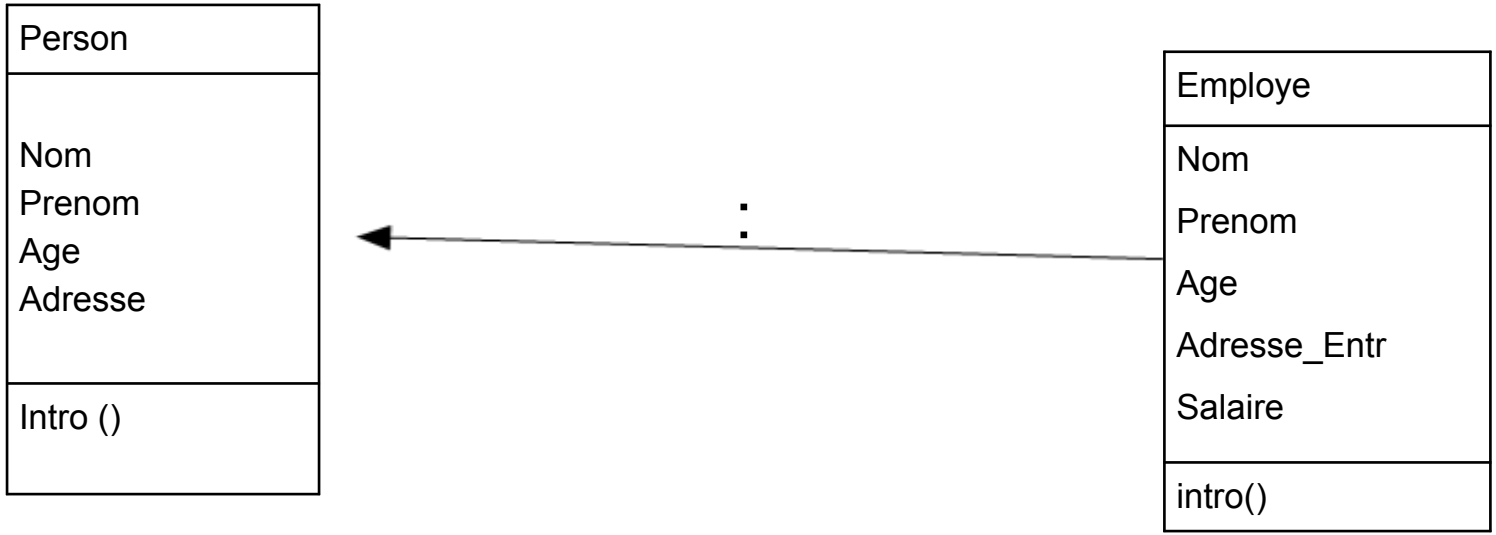
Exemple:

- **Classe Person**
  - Une personne contient un nom , prénom, âge et adresse
- **Classe Employe**
  - Un employé est une personne qui travaille dans une Entreprise.
  - Un employé a une entreprise et un salaire.

L’héritage

Classe de base (parent)

Classe dérivée (enfant)



L’héritage

- La notion d'héritage est l'un des fondements de la programmation orientée objet.Grâce à elle, nous pourrons créer des classes héritées (appelées aussi classes dérivées ou classes filles) de nos classes mères (appelées aussi classes de base).
- (:) permet de spécifier le nom de la classe mère.
- La classe fille (Employé) hérite des propriétés et méthodes de la classe Personne(Person)

L’héritage

```
class Person
{
    2 references
    protected string Nom;

    2 references
    protected string Prenom;
    2 references
    protected int Age;
    2 references
    protected string Adresse;


    0 references
    public Person()
    {
        Console.WriteLine("constructeur par default de la calss Person");
    }
    1 reference
    public Person(string nom,string prenom, int age,string adresse){
        this.Nom=nom;
        this.Prenom=prenom;
        this.Age=age;
        this.Adresse=adresse;
    }

    1 reference
    public void intro(){
        Console.WriteLine($"je m'appelle {Nom},{Prenom}, j'ai {Age}, et j'habite à {Adresse}");
    }


}
```

L’héritage

```
4 references
class Employe : Person
{
    1 reference
    private string AdreEntr;
    1 reference
    private double Salaire;

    0 references
    public Employe(){

    }
    1 reference
    public Employe(string nom1, string prenom1, int age1,string adresse1,string adreEntr,double salaire ): base(nom1,prenom1,age1,adresse1)
                {
                    this.AdreEntr=adreEntr;
                    this.Salaire=salaire;
                }
}
0 references
class program
{

    0 references
    static void Main (string[] args)
    {
        Employe e=new Employe("aa","bb",25,"cc","dd",2000);
        e.intro();

    }
}
```

L’héritage

- Appel implicite au constructeur d'Individu

```
Employe e=new Employe();
e.intro();
```

- L'objet se présente comme s'il était une personne (person) car il hérite de la méthode intro(). On obtient l’affichage :

```
//constructeur par défaut de la class Person
je m’appelle   j’ai 0 et j’habite à
```

- pour construire l’instance de Employé, le compilateur appelle d’abord le constructeur par défaut de la classe Person :
- Comment initialiser le nom, le prénom, l'âge et la ville d’un employé ?

L'héritage

- on va utiliser le constructeur suivant :

```
public Employe(string nom1, string prenom1, int age1,string adresse1,string adreEntr,double salaire ):
base(nom1,prenom1,age1,adresse1)
        {
            this.AdreEntr=adreEntr;
            this.Salaire=salaire;
        }
```

Rq: base est le mot-clé pour désigner l'appel au constructeur de la classe de base (ou classe mère).

```
public Person(string nom,string prenom, int age,string adresse){
        this.Nom=nom;
        this.Prenom=prenom;
        this.Age=age;
        this.Adresse=adresse;
    }
```

L’héritage

- la class Main

```
Employe e=new Employe("a1","b1",25,"c1","d1",2000);
        e.intro();
```

affichage :

```
je m'appelle a1,b1, j'ai 25, et j'habite à c1
```

Polymorphisme

Le polymorphisme, littéralement "plusieurs formes", se produit lorsque plusieurs classes sont interconnectées par le biais de l'héritage.

Comme mentionné précédemment, l'héritage nous permet d'hériter des attributs et des méthodes d'une classe parent. Le polymorphisme utilise ces méthodes pour exécuter différentes actions, offrant ainsi la possibilité d'effectuer une même action de différentes manières.

Par exemple, considérons une classe de base appelée Véhicule avec une méthode nommée SonMoteur(). Les classes dérivées de Véhicule pourraient être des Voitures, des Camions, des Motocyclettes, etc. Chacune de ces classes dérivées pourrait avoir sa propre implémentation spécifique du son du moteur (une voiture peut vrombir, un camion peut rugir, etc.).

Polymorphisme

```
class Vehicule  // Classe de base (parent)
{
    public void SonMoteur()  // Méthode pour le son du moteur
    {
        Console.WriteLine("Le véhicule fait un bruit de moteur");
    }
}

class Voiture : Vehicule  // Classe dérivée (enfant) pour les voitures
{
    public void SonMoteur()  // Méthode pour le son du moteur des voitures
    {
        Console.WriteLine("La voiture fait un bruit de vroum vroum");
    }
}

class Camion : Vehicule  // Classe dérivée (enfant) pour les camions
{
    public void SonMoteur()  // Méthode pour le son du moteur des camions
    {
        Console.WriteLine("Le camion fait un bruit de vrombissement");
    }
}
```

Polymorphisme

Affichage:
Dans ce cas, l'appel de la méthode SonMoteur() sur des objets de types Voiture ou Camion exécutera toujours la version de la méthode dans la classe de base Véhicule. Donc, l'affichage pour les objets Voiture et Camion sera identique et correspond au message de la classe de base :

```
Le véhicule fait un bruit de moteur
```

Polymorphisme

```
class Vehicule  // Classe de base (parent)
{
    public virtual void SonMoteur()  // Méthode virtuelle pour le son du moteur
    {
        Console.WriteLine("Le véhicule fait un bruit de moteur");
    }
}
class Voiture : Vehicule  // Classe dérivée (enfant) pour les voitures
{
    public override void SonMoteur()  // Méthode override pour le son du moteur des voitures
    {
        Console.WriteLine("La voiture fait un bruit de vroum vroum");
    }
}
class Camion : Vehicule  // Classe dérivée (enfant) pour les camions
{
    public override void SonMoteur()  // Méthode override pour le son du moteur des camions
    {
        Console.WriteLine("Le camion fait un bruit de vrombissement");
    }
}
```

Polymorphisme

Affichage:

Dans ce cas, les méthodes SonMoteur() dans les classes dérivées Voiture et Camion remplacent la version de la méthode héritée de la classe de base Vehicule. Donc, l'affichage pour les objets Voiture et Camion sera spécifique à chaque classe dérivée :

Pour la classe Voiture, l'affichage sera :

- La voiture fait un bruit de vroum vroum

Pour la classe Camion, l'affichage sera:

- Le camion fait un bruit de vrombissement

Polymorphisme

Dans le premier code, la méthode SonMoteur() dans la classe de base Véhicule n'est pas marquée comme virtual, et les méthodes correspondantes dans les classes dérivées Voiture et Camion ne sont pas marquées comme override. Par conséquent, lors de l'appel de la méthode SonMoteur() sur des objets de types Voiture ou Camion, la version de la méthode dans la classe de base sera toujours exécutée.

Dans le deuxième code, la méthode SonMoteur() dans la classe de base Véhicule est marquée comme virtual, ce qui permet aux classes dérivées Voiture et Camion de la redéfinir. Les méthodes correspondantes dans les classes dérivées sont marquées comme override, ce qui indique qu'elles remplacent la version de la méthode héritée de la classe de base. Ainsi, lorsque vous appelez la méthode SonMoteur() sur des objets de types Voiture ou Camion, la version spécifique de la méthode dans la classe dérivée sera exécutée.

Mots clé virtual, new et override

Le mot clé virtual: • Est utilisé par les méthodes de la classe mère pour autoriser les classe filles à modifier la méthode

exemple:

```
public virtual void informations()
{ Console.WriteLine("méthode virtuelle");
}
```

Redéfinition des méthodes virtuelles

La redéfinition des méthodes virtuelles dans la classe fille on utilise 2 mots clés :

- **new** : indique au compilateur que vous ajoutez une méthode à une classe dérivée avec le même nom que la méthode dans la classe de base, mais sans aucune relation entre elles:

```
public new void informations()
{ Console.WriteLine("méthode redéfinie par new");
}
```

Redéfinition des méthodes virtuelles

override: indique au compilateur que les deux méthodes sont liés

exemple:

```
public override void informations()
{ Console.WriteLine("méthode redéfinie par override");
}
```

Récapitulative Pour l'héritage

- Une classe ne peut hériter que d'une seule et unique classe !
- Si aucun constructeur n'est défini dans une classe fille, le compilateur en créera un et appellera automatiquement le constructeur de la classe mère.
- La classe fille hérite de toutes les propriétés et méthodes public et protected de la classe mère.
- Les méthodes et propriétés privées d'une classe mère ne sont pas accessibles dans la classe fille.
- On peut redéfinir (changer tout le code) d'une méthode héritée.
- On peut utiliser le polymorphisme sur une méthode par le biais du mot clé base.Correspond à la p
- Le polymorphisme possibilité pour un opérateur ou une fonction d'être utilisable dans des contextes différents (différenciables par le nombre et le types des paramètres) et d'avoir un comportement adapté à ces paramètres.
- Si une méthode d'une classe mère n'est pas redéfinie ou polymorphe, à l'appel de cette méthode par le biais d'un objet enfant, c'est la méthode de la classe mère qui sera appelée !
- Vous ne pouvez pas hériter d'une classe déclarée const. Une méthode déclarée const est non redéfinissable.

Les classes abstraites

- L'abstraction des données est le processus consistant à masquer certains détails et à afficher uniquement les informations essentielles à l'utilisateur.
- L'abstraction peut être réalisée soit avec des classes abstraites , soit avec des interfaces (chapitre suivant)
- Le **abstract** mot-clé est utilisé pour les classes et les méthodes :
  - **Classe abstraite :** est une classe restreinte qui ne peut pas être utilisée pour créer des objets (pour y accéder, elle doit être héritée d'une autre classe).
  - **Méthode abstraite :** ne peut être utilisée que dans une classe abstraite et n’a pas de corps. Le corps est fourni par la classe dérivée (hérite de).

- Une classe **abstraite** est comme une classe normale. Ceci dit, elle a tout de même une particularité

Les classes abstraites

- **vous ne pouvez pas l'instancier !**
- **c.a.d :**

```
class program
{ static void Main(string[] args)
  { A obj1=new A();// Erreur de compilation

  }
}
```

Les classes abstraites

- Une classe considérée comme abstraite. Elle doit être déclarée avec le mot clé abstract.
- Une telle classe peut avoir le même contenu qu'une classe normale (attributs et méthodes).
- Cependant, ce type de classe permet de définir des méthodes abstraites.
- Ces méthodes ont une particularité ; elle n'ont pas de corps !
- Une méthode abstraite ne peut exister que dans une classe abstraite

Les classes abstraites

Exemple :

```
public abstract class Forme
{
    // Méthode abstraite pour calculer l'aire de la forme
    public abstract double CalculerAire();
}
```

Les classes abstraites

```
public class Cercle : Forme
{
    private double Rayon;
// Constructeur de la class Cercle
    public Cercle(double rayon)
    {
        Rayon = rayon;
    }

    public override double CalculerAire()
    {
        return Math.PI * Rayon * Rayon;
    }
}
```

Les classes abstraites

```
public class Rectangle : Forme
{
    private double Largeur;
    private double Hauteur;
// Constructeur de la class Rectangle
    public Rectangle(double largeur, double hauteur)
    {
        Largeur = largeur;
        Hauteur = hauteur;
    }

    public override double CalculerAire()
    {
        return Largeur * Hauteur;
    }
}
```

Les classes abstraites

```
class Program
{
    static void Main(string[] args)
    {
        Forme cercle = new Cercle(5);
        Forme rectangle = new Rectangle(4, 6);

        Console.WriteLine("Aire du cercle : " + cercle.CalculerAire());
        Console.WriteLine("Aire du rectangle : " + rectangle.CalculerAire());
    }
}
```

Aire du cercle : 78,53981633974483
Aire du rectangle : 24

Les interfaces

- Avec l'héritage multiple, une classe peut hériter en même temps de plusieurs super classes. Ce mécanisme n'existe pas en c#.
- Les interfaces permettent de mettre en oeuvre un mécanisme de remplacement.
- Une interface est un ensemble de déclarations de méthodes abstraites.
- Tous les objets qui implémentent cette interface possèdent les méthodes déclarées dans celle-ci.
- Plusieurs interfaces peuvent être implémentées dans une même classe.
- Les classes peuvent implémenter une ou plusieurs interfaces
- Elles en dérivent comme une classe
- Les implémentations des méthodes doivent être publiques
- Les interfaces se définissent avec le mot-clef « interface »

Déclaration d'une interface

Déclaration d'une interface:

Modificateurs class nomClasse : I1 **{** **//insérer ici des méthodes et des champs .** **}**	**Modificateurs class nomClasse :I1,I2,I3….** **{** **//insérer ici des méthodes et des champs .** **}**

```
Public interface nomInterface
{
// insérer ici des méthodes abstraites et public.
}
```

Déclaration d'une interface

```
using System;
// Définition de l'interface
public interface IVehicule
{
    void Demarrer();
    void Arreter();
}
// Implémentation de l'interface dans une classe
public class Voiture : IVehicule
{
    public void Demarrer()
    {
        Console.WriteLine("La voiture démarre.");
    }
    public void Arreter()
    {
        Console.WriteLine("La voiture s'arrête.");
    }
}
```

Déclaration d'une interface

```
public class Program
{
    public static void Main(string[] args)
    {
        // Création d'un objet de type Voiture
        Voiture maVoiture = new Voiture();

        // Appel des méthodes de l'interface
        maVoiture.Demarrer();
        maVoiture.Arreter();
    }
}
```

Déclaration d'une interface

```
using System;

// Première interface
public interface ICalcul
{
    int Additionner(int a, int b);
    int Soustraire(int a, int b);
}
// Deuxième interface
public interface IAffichage
{
    void AfficherMessage(string message);
}
// Classe qui implémente les deux interfaces
```

```
public class CalculEtAffichage : ICalcul, IAffichage
{
    public int Additionner(int a, int b)
    {
        return a + b;
    }

    public int Soustraire(int a, int b)
    {
        return a - b;
    }

    public void AfficherMessage(string message)
    {
        Console.WriteLine(message);
    }
}
```

Déclaration d'une interface

```
class Program
{
    static void Main(string[] args)
    {
 // Création d'une instance de la classe qui implémente les deux interfaces

        CalculEtAffichage ca = new CalculEtAffichage();

// Utilisation des méthodes des interfaces

        int somme = ca.Additionner(5, 3);
        int difference = ca.Soustraire(10, 4);
        ca.AfficherMessage($"La somme est {somme} et la différence est {difference}");
    }
}
```