

PlateForme C#.Net

Ing. Meryem OUARRACHI

Plan

- ☐ **L'environnement .Net**
- ☐ **Initiation à la programmation C#**
- ☐ **Programmation Orienté Objet C#**
- ☐ **Programmation avancée en C#**

CHAPITRE 3:

Programmation O.O en C#

Héritage

- En C# ,l'héritage est spécifié au niveau de la déclaration de classe

```
class X :Y { // la classe X dérive de la classe Y  
}
```

- L'héritage multiple est interdit en c#

```
class X :Y,Z { // Erreur  
}
```

Le mot clé « base »

- Le mot-clé « base » permet d'accéder aux méthodes et aux attributs de la super-classe
- « base » est utilisé dans le constructeur de la classe dérivée pour appeler celui de la super-classe.
 - « base(...) » doit être la première instruction du constructeur.
- L'absence d'appel à un constructeur de la super-classe dans le constructeur d'une classe entraîne implicitement un appel au constructeur par défaut de cette super-classe (sans paramètres).

Le mot clé « base »

```
class ObjectGraphique {  
    public int x, y ;  
    public ObjectGraphique(int x, int y) {  
        this.x = x ;    this.y = y ;  
    }  
    public String toString( ) {  
        return x + " : " + y ;  
    } }  
}
```

```
class Cercle: ObjectGraphique {  
    public double rayon;  
    // première instruction dans le constructeur  
    public Cercle(int x, int y, double ray):base(x, y) {  
        rayon = ray;  
    }  
    public String Affiche( ) {  
        return base. toString( ) + " : " + rayon ;    } }  
}
```

Le mot clé « sealed »

- Le mot clé sealed permet d'empêcher l'héritage d'une classe

```
sealed class Cercle { }  
  
//Erreur puisque Cercle est déclaré comme sealed  
class SousCercle : Cercle { }
```

Le polymorphisme

- La classe dérivée peut changer l'implémentation d'une ou plusieurs méthodes héritées : redéfinition
- Une méthode polymorphe est une méthode déclarée dans une super-classe et redéfinie dans une sous-classe

Exemple:

-On a la méthode toString défini dans les deux classe A et Cercle

```
ObjectGraphique A= new Cercle(2,3,6);
```

```
Console.WriteLine(A. toString( )) ;
```

→ C'est la méthode de classe cercle qui va être exécutée

Le polymorphisme

- Le polymorphisme est obtenu grâce au liaison dynamique: la méthode qui sera exécutée est déterminée:
 - seulement à l'exécution, et pas à la compilation
 - par le type réel de l'objet qui reçoit le message (et pas par son type déclaré)
- Bien utilisé le polymorphisme évite les codes qui comportent de nombreux embranchements et tests

Surdéfinition des méthodes

- Pour surdéfinir une méthode d'une classe mère il faut utiliser le mot clé **override**:

```
class Cercle
{
    public override void toString() { }
}
```

- Afin de pouvoir surdéfinir une méthode de la classe mère, elle doit être soit :
 - Une méthode virtuelle.
 - Une méthode abstraite.
 - Une surdéfinition d'une autre méthode.

Méthode virtuelle

- Les méthodes virtuelles sont des méthodes qui ont un corps (un comportement) et qu'on peut les surdéfinir dans les classes filles

→ surdéfinition optionnelle.

```
class ObjectGraphique
{
    public virtual void Affiche() { }
}
```

Méthode et classe abstraite

- Les méthodes abstraites est une méthode dont on donne la signature sans décrire l'implémentation et on *doit* surdéfinir dans les classes filles → surdéfinition obligatoire.

public abstract nom_methode(params); //pas de code

- Une classe abstraite est une classe qui contient au moins une méthode abstraite.
 - Une classe abstraite est une classe qui n'est pas faite pour être instanciée, elle est destinée à être dérivée

public abstract class nom_classe {...}

Méthode et classe abstraite

```
abstract class Class1
{ public abstract void affiche();
}
```

```
class Class2 : Class1
{
    public override void affiche()
    {
        Console.WriteLine("Bonjour");
    }
}
```

Surdéfinition de méthode

- Si une surdéfinition est marquée par le mot clé sealed aucune surdéfinition supplémentaire n'est autorisée.

```
class ClasseMere
{
    public virtual void Methode1() { }
    public virtual void Methode2() { }
}
class ClasseFille : ClasseMere
{
    public override void Methode1() { } //OK
    public sealed override void Methode2() { }
    //sealed, surdéfinition supplémenataire interdite
}
class ClasseSousille : ClasseFille
{
    public override void Methode1() { } //OK
    public override void Methode2() { } //Erreur
}
```

Masquage des Membres

```
class Forme
{
    public void affiche()
    {Console.WriteLine("forme");    }
}
```

```
class Cercle : Forme
{
    public void affiche()
    {Console.WriteLine("cercle");    }
}
```

```
class Program
{
    static void Main(string[] args) {
        Forme f = new Cercle();
        f.affiche();
        Console.ReadLine();
    }
}
```


Quel est le résultat d'exécution de ce programme?

Masquage des Membres

-Le programme va afficher « Forme »

Pourquoi? Si on déclare un membre « X » dans la classe fille « B » qui est identique à un autre membre « Y » de la classe mère « A » sans utiliser le mot clé `override`, on dit que *le membre « Y » est masqué dans le membre « X »*;

- Dans ce cas, ce membre n'est plus polymorphique entre les deux classes « A », et « B » donc:

 Les règles du polymorphisme qu'on venait de voir ne s'appliquent pas.

Masquage des Membres

- Dans ce cas aussi,le compilateur affiche un avertissement.
- Pour affirmer que le masquage est fait exprès et ne pas avoir l'avertissement du compilateur,on utilise le mot clé new

```
class Forme
{
    public void affiche()
    {Console.WriteLine("forme");    }
}
class Cercle : Forme
{
    public new void affiche()
    {Console.WriteLine("cercle");    }
}
```

Les interfaces

- C'est une classe composée d'un ensemble de méthodes abstraites

```
interface uneInterface {  
    nomMeth1 (paramètres) ;  
    nomMeth2 (paramètres) ;  
    ...  
}
```

- Les classes qui implémentent une interface ,doivent obligatoirement redéfinir toutes les méthodes de cet interface

```
public class UneClasse : uneInterface
```

- Une classe peut **simultanément** dériver et implémenter (la dérivation est indiquée avant l'implémentation).
- Une classe peut hériter de plusieurs interfaces.

```
public class UneClasse : uneInterface1 , uneInterface2
```

Les interfaces

```
public interface UneInterface {  
    void m1();  
    void m2();  
}
```

```
public class ClasseInstanciable:UneInterface {  
    public void m1() {...}  
    public void m2() {...}  
}
```

Les interfaces

- Une interface est un type qui:
 - Ne peut pas avoir des constructeurs et des champs.
 - Ne spécifie aucun modificateur d'accès pour ses membres, ils sont tous implicitement public et abstract.
 - Ne peut pas avoir des membres statiques.
 - Ne peut jamais hériter d'une classe même si celle-ci est abstraite.
 - Peut hériter d'autres interfaces (une ou plusieurs).

Les interfaces

- Une interface est un type qui:
 - Ne peut pas avoir des constructeurs et des champs.
 - Ne spécifie aucun modificateur d'accès pour ses membres, ils sont tous implicitement public et abstract.
 - Ne peut pas avoir des membres statiques.
 - Ne peut jamais hériter d'une classe même si celle-ci est abstraite.
 - Peut hériter d'autres interfaces (une ou plusieurs).

Les interfaces: Implémentation explicite

- Problématique

```
//2 interfaces avec 2 méthodes identiques en terme de signature  
interface Iinterface1{ void methode1(); }  
  
interface Iinterface2 { void methode1(); }
```

```
class class1: Iinterface1, Iinterface2  
{  
    public void methode1()  
    {  
        // s'agit-il de la méthode de l'interface1 ou bien de Iinterface2?  
        }  
    }  
}
```

Les interfaces: Implémentation explicite

- Problématique

```
Iinterface1 c1 = new class1();  
Iinterface2 c2 = new class1();  
class1 c3 = new class1();  
c1.methode1();//On veut l'exécution de methode1 de interface1 pas ce qu'on aura  
c2.methode1();// On veut l'exécution de methode1 de interface1 pas ce qu'on aura  
c3.methode1();//Exécution de la méthode implémentée
```

Les interfaces: Implémentation explicite

- Solution

```
//2 interfaces avec 2 méthodes identiques en terme de signature  
interface Iinterface1{    void methode1(); }
```

```
interface Iinterface2 {    void methode1(); }
```

```
class class1: Iinterface1, Iinterface2
```

```
{
```

```
    void Iinterface1.methode1()
```

```
    { //Le mot clé public n'est pas autorisé sur  
      //une déclaration d'interface explicite
```

```
    }
```

```
    void Iinterface2.methode1()
```

```
    { //Le mot clé public n'est pas autorisé sur  
      //une déclaration d'interface explicite
```

```
    }
```

```
    public void methode1()
```

```
    { //Le mot clé public n'est pas autorisé sur  
      //une déclaration d'interface explicite
```

```
    }
```

```
}
```


Les interfaces: Implémentation explicite

- Solution

```
Iinterface1 c1 = new class1();
```

```
Iinterface2 c2 = new class1();
```

```
class1 c3 = new class1();
```

```
c1.methode1();//On va avoir l'exécution de methode1 de interface1
```

```
c2.methode1();//On va avoir l'exécution de methode1 de interface2
```

```
c3.methode1();//Exécution de la méthode implémentée
```

Le mot clé « is »

-Ce mot clé « is » permet de vérifier le type d'une variable:

```
if (obj1 is class1)
{
}
```

Les Conversions

```
int x = 7;  
long y = x; // Cast implicite  
  
double z = (double)x; //Cast explicite  
  
object o="salut" ;  
string s = o as string; // cast via le mot clé as
```

Remarque:

- as ne prend pas en considération les types values.
- Contrairement au cast explicite, le mot clé **as** ne déclenche pas une exception si le type n'est pas correct mais retourne un **null**.

Les collections

Les collections

- Une collection est un tableau dynamique d'objets.
- En C#, on a plusieurs types de collections:
 - List
 - ArrayList
 - Vector
 - Dictionary
- Une collection fournit un ensemble de méthodes qui permettent:
 - D'ajouter ou supprimer un objet
 - Rechercher un élément
 - Trier les éléments
 - etc

List

- récupère un élément de la liste:

```
List<int> p =new List<int> ()
```

```
p[int index]
```

- Parmi les méthodes proposées par cette classe on a :
 - void Add (Object elt) : insertion de l'élément elt à un index de la liste
 - void RemoveAt(int index) : supprime un élément à un index de la liste
 - int Count() : retourne le nombre d'éléments de la liste
 - Object Max(): récupère le maximum d'une liste
 - Object[]:Sort():pour trier la liste

List

```
List<int> f = new List<int>();  
f.Add(100);  
f.Add(200);  
f.Add(90);
```

```
//parcourir la liste  
for (int i = 0; i < f.Count; i++)  
{Console.WriteLine("value N[{0}]= {1}" ,i, f[i]); }
```

```
//recupérer le maximum  
Console.WriteLine("le maximum dans la liste est"+f.Max());
```

```
//Trier la liste  
f.Sort();  
for (int i = 0; i < f.Count; i++)  
{Console.WriteLine("value N[{0}]= {1}", i, f[i]); }
```

Dictionary

- Un dictionnaire est une collection d'objet de type clé/ Valeur. Chaque clé référence l'objet dans le dictionnaire.

```
Dictionary<int, string> dc = new Dictionary<int, string>();
```

dc[int index]: récupère un élément de dictionnaire de la clé index

- **Les méthodes:**

- **Object Add(Object key, Object value)** : ajoute un élément dans le dictionnaire
- **void Clear()** : supprime tous les éléments de dictionnaire,
- **Values** : retourne tous les éléments de la table,
- **keys** : retourne cette fois ci toutes les clefs,
- **int Count()** : nombre d 'éléments dans la table,
- **Object Remove(Object key)** : supprime un élément.

Dictionary

```
Dictionary<int, string> dc = new Dictionary<int, string>();
```

```
dc.Add(1, "printemps");  
dc.Add(10, "été");  
dc.Add(12, "automne");  
dc.Add(45, "hiver");
```

```
//récupérer les valeurs  
foreach (string v in dc.Values)  
{ Console.WriteLine("les valeurs = {0}", v); }
```

```
//récupérer les clés  
foreach (int c in dc.Keys)  
{ Console.WriteLine("les clés = {0}", c); }
```

Les interfaces Comparable / Comparer

- Le framework définit aussi des interfaces pour faciliter le parcours des collections et leur tri :
 - **Comparable** : interface pour définir un ordre de tri naturel pour un objet
 - **Comparer** : interface pour définir un ordre de tri quelconque

Interface Comparable

- Elle propose une méthode :

int CompareTo(Object o)

Elle doit renvoyer :

- <0 si this est **inférieur** à o
- $=0$ si this est **égal** à o
- >0 si this est **supérieur** à o

Interface IComparable

```
public class Personne : IComparable
```

```
{ public String nom;  
  public Personne(String nom)  
  { this.nom = nom; }  
}
```

```
public int CompareTo(object o)  
{ Personne p = o as Personne;  
  return nom.CompareTo(p.nom);  
} }
```

```
class test  
{ static void Main(string[] args)  
  { List<Personne> p = new List<Personne>();  
    p.Add(new Personne("ouarrachi"));  
    p.Add(new Personne("nouri"));  
  
    p.Sort();  
    foreach (var v in p)  
    { Console.WriteLine(v.nom); }  
  }  
}
```

Interface IComparable

```
public class Personne : IComparable<Personne>
{
    public String nom;
    public Personne(String nom)
    { this.nom = nom; }

    public int CompareTo(Personne p)
    {return nom.CompareTo(p.nom); }
}

class test
{
    static void Main(string[] args)
    {
        List<Personne> p = new List<Personne>();
        p.Add(new Personne("ouarrachi"));
        p.Add(new Personne("nouri"));

        p.Sort();
        foreach (var v in p)
        { Console.WriteLine(v.nom); }
    }
}
```

Interface IComparer

- Elle propose une méthode :

int Compare(Object o1, Object o2)

Elle doit renvoyer :

- Un entier positif si o1 est "plus grand" que o2.
- 0 si la valeur de o1 est identique à celle de o2.
- Un entier négatif si o1 est "plus petit" que o2.

Interface IComparer

```
public class ComparatorPersonne : IComparer<Personne>
{
    public int Compare(Personne p1, Personne p2)
    {
        String nom1 = p1.Nom;
        String nom2 = p2.Nom;
        return nom1.CompareTo(nom2);
    }
}
```

Interface IComparer

```
static void Main(string[] args)
{
    List<Personne> p = new List<Personne>();

    p.Add(new Personne("ouarrachi"));
    p.Add(new Personne("aaaa"));
    p.Add(new Personne("nouri"));

    p.Sort(new ComparatorPersonne());

    Console.WriteLine(p.BinarySearch(new Personne("nouri"), new ComparatorPersonne()));
}
```


LINQ : Language INtegrated Query

-Linq est une technologie permettant de faciliter la gestion de divers sources de donnée

-Base de donnée relationnelles SQL,

-Fichier XML

-Collections

-Tableau

via l'utilisation d'un seul modèle



Les développeurs ne sont pas obligés d'apprendre le langage de requête pour chaque type de donnée

LINQ : Language INtegrated Query

- Linq propose des requêtes qui s'exécutent de la même façon pour n'importe quel format de données
- Les requêtes Linq utilisent toujours le modèle objet pour interroger les données

Syntaxe d'une requête Linq

List<Personne> per = new List<Personne> ();

Sélectionner des éléments de la collection	var x = from p in per select p;
Sélectionner des éléments de la collection selon une condition	var x = from p in per where p.age==17 select c
Trier les éléments de la collection	var x= from p in per Order by p.age ascending select p;

Syntaxe d'une requête Linq

Parcourir les éléments du
résultat de select

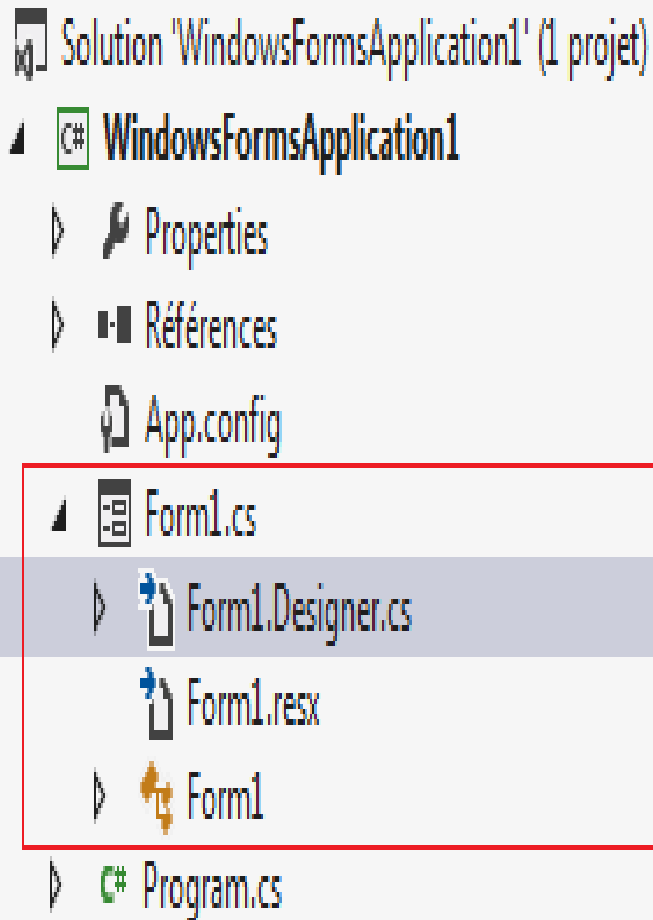
```
foreach (var i in x)  
{ console.WriteLine(i.nom); }
```

WindowsForms

Définition

- **WindowsForms**" est le nom de l'interface graphique qui est incluse dans le framework .NET.
- C'est ce qui nous permet de faire des applications en fenêtres (appelées **forms**).
- Il s'agit de client lourd

Structure de projet WindowsForms



-Form1.Designer:contient la liste des composants;

Affecte des propriétés aux composants

Affecte des événements aux composants

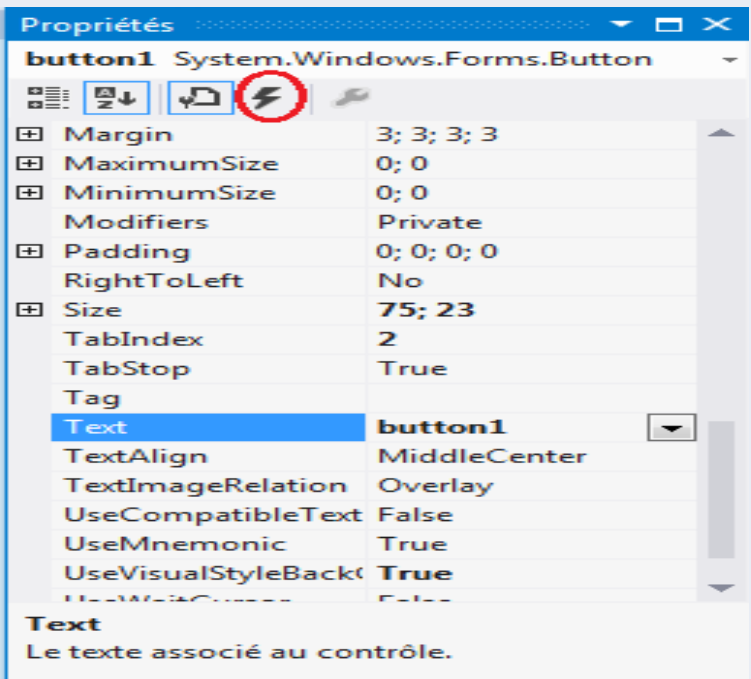
-Form1.cs: gestion des événements

-Program.cs:le main de projet

Les contrôles

-Les contrôles sont des objets que nous pouvons déposer dans une fenêtre par un simple glisser-déposer ;

Exemple: ajouter un bouton à l'interface graphique



Propriété	Valeur
Margin	3; 3; 3; 3
MaximumSize	0; 0
MinimumSize	0; 0
Modifiers	Private
Padding	0; 0; 0; 0
RightToLeft	No
Size	75; 23
TabIndex	2
TabStop	True
Tag	
Text	button1
TextAlign	MiddleCenter
TextImageRelation	Overlay
UseCompatibleText	False
UseMnemonic	True
UseVisualStyleBackg	True
UseWaitCursor	False

Text
Le texte associé au contrôle.

-Propriétés: size, le contenu, aspect visuel de composant

-Événements: actions à exécuter si un événement est déclenché: click, MouseEnter

...

La méthode InitializeComponent

-C'est la méthode qui initialise la liste des composants de l'interface graphique par leurs propriétés

```
private void InitializeComponent()
{
    this.button1 = new System.Windows.Forms.Button();
    //
    // button1
    //
    this.button1.Location = new System.Drawing.Point(64, 70);
    this.button1.Name = "button1";
    this.button1.Size = new System.Drawing.Size(75, 23);
    this.button1.TabIndex = 0;
    this.button1.Text = "button1";
    this.button1.UseVisualStyleBackColor = true;
    //
    // Form2
    //
    this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
    this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
    this.ClientSize = new System.Drawing.Size(593, 307);
    this.Controls.Add(this.button1);
    this.Name = "Form2";
    this.Text = "Form2";
}
```

La méthode InitializeComponent

```
public partial class Form2 : Form
{
    public Form2()
    {
        InitializeComponent();
    }
}
```

La méthode Dispose

-C'est une méthode qui permet de libérer toutes les ressources de forme

```
protected override void Dispose(bool disposing)
{
    if (disposing && (components != null))
    {
        components.Dispose();
    }
    base.Dispose(disposing);
}
```

Les événements

-On appelle **évènement** une action particulière qui a lieu lorsque quelque chose se produit sur un objet
(click, expiration d'un délai...)

Fonctionnement des événements

- Afin d'effectuer une action lorsqu'un certain événement est déclenché, il faut déjà savoir quand est déclenché l'événement dont il est question
- Avoir les gestionnaires d'événements (Eventhandlers) qui servent à indiquer que l'événement surveillé vient d'être déclenché.
- on attache un gestionnaire d'événements à un événement en spécifiant une méthode. Cette méthode est appelée lorsque l'événement est déclenché.

Fonctionnement des événements

-Form1.Designer.cs

```
this.button1.Click += new System.EventHandler(this.button1_Click);
```

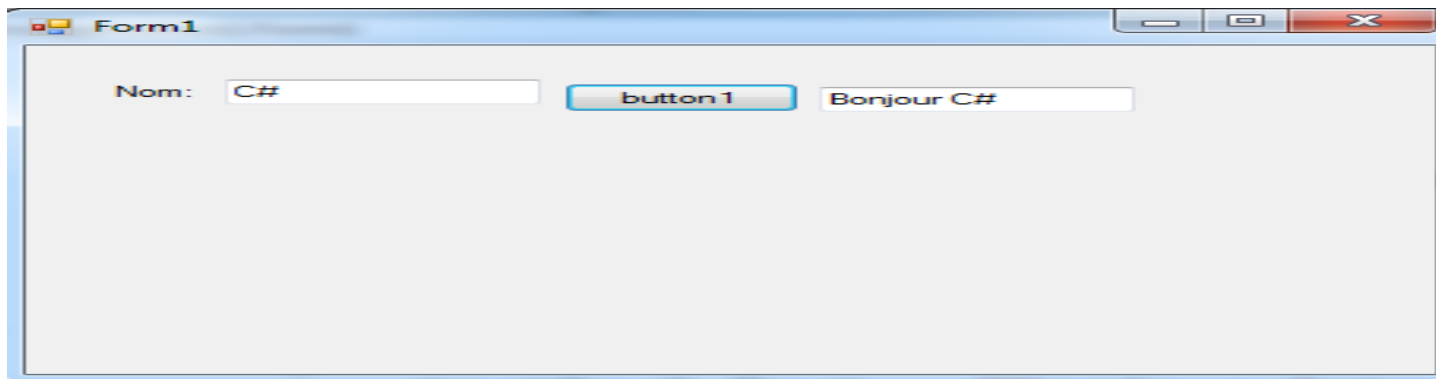
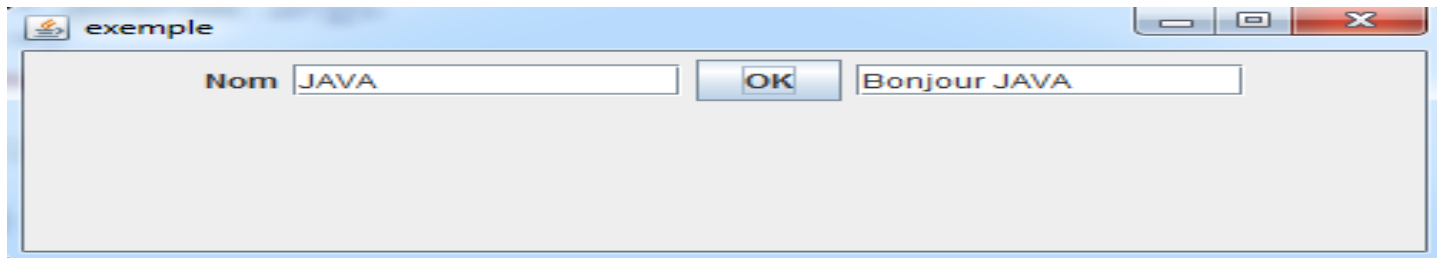
-Form1.cs

```
private void button1_Click(object sender, EventArgs e)
{
    textBox1.Text = "Bonjour";
}
```

Approche entre swing et windowsForms

Exemple:

créer l'interface graphique suivante en (C# ,windowsForms) et en (Java,Swing)



Approche entre swing et windowsForms

-Solution en Swing:

```
public class Classe1 implements ActionListener
{
    JFrame frame =new JFrame("exemple");
    JButton jb=new JButton("OK");
    JLabel jl=new JLabel("Nom");
    JTextField jt1=new JTextField(" ",12);
    JTextField jt2=new JTextField(" ",12);

    //constructeur
    Classe1(){
        jb.addActionListener (this);
        JPanel jp=new JPanel();
        jp.add(jl);
        jp.add(jt1);
        jp.add(jb);
        jp.add(jt2);
        frame.getContentPane().add(jp,BorderLayout.CENTER);
        frame.show();
    }
}
```


Approche entre swing et windowsForms

-Solution en Swing:

```
public void actionPerformed (ActionEvent e)
{
    jt2.setText("Bonjour "+jt1.getText()) ;
}
```

Approche entre swing et windowsForms

-Solution en WindowsForms:

- Glisser les composants dans la fenêtre(Label,TextBox et button)
- Changer le texte des composants dans la partie propriété
- Double clique sur le bouton pour avoir accès à la méthode qui traite l'événement click
- Voir le code généré automatiquement

Approche entre swing et windowsForms

-Solution en WindowsForms:

```
private void InitializeComponent()
{
    this.label1 = new System.Windows.Forms.Label();
    this.textBox2 = new System.Windows.Forms.TextBox();
    this.button1 = new System.Windows.Forms.Button();
    this.textBox1 = new System.Windows.Forms.TextBox();

    //
    // label1
    //
    this.label1.AutoSize = true;
    this.label1.Location = new System.Drawing.Point(13, 18);
    this.label1.Name = "label1";
    this.label1.Size = new System.Drawing.Size(32, 13);
}
```

```
private void button1_Click(object sender, EventArgs e)
{
    textBox2.Text = "Bonjour "+textBox1.Text;
}
```

Approche entre swing et windowsForms

Remarque: Si vous trouvez une difficulté dans une propriété d'un composant quelconque; Ecrire en google le nom de composant+propriété cherchée il va vous guider vers la page learn.microsoft de ce composant

Approche entre swing et windowsForms

-Résumé

- Tous les deux ont le même principe de fonctionnement.
- En java,il faut avoir plus des connaissances sur les composants afin de bien contrôler l'interface graphique
 - Il faut apprendre au début les propriétés de ces contrôles
- En windowsForms:il y'a une facilité de créer l'interface graphique en utilisant la technique Glisser/Déposer
 - On peut passer directement et spontanément dans le projet sans savoir au préalable tous les composants