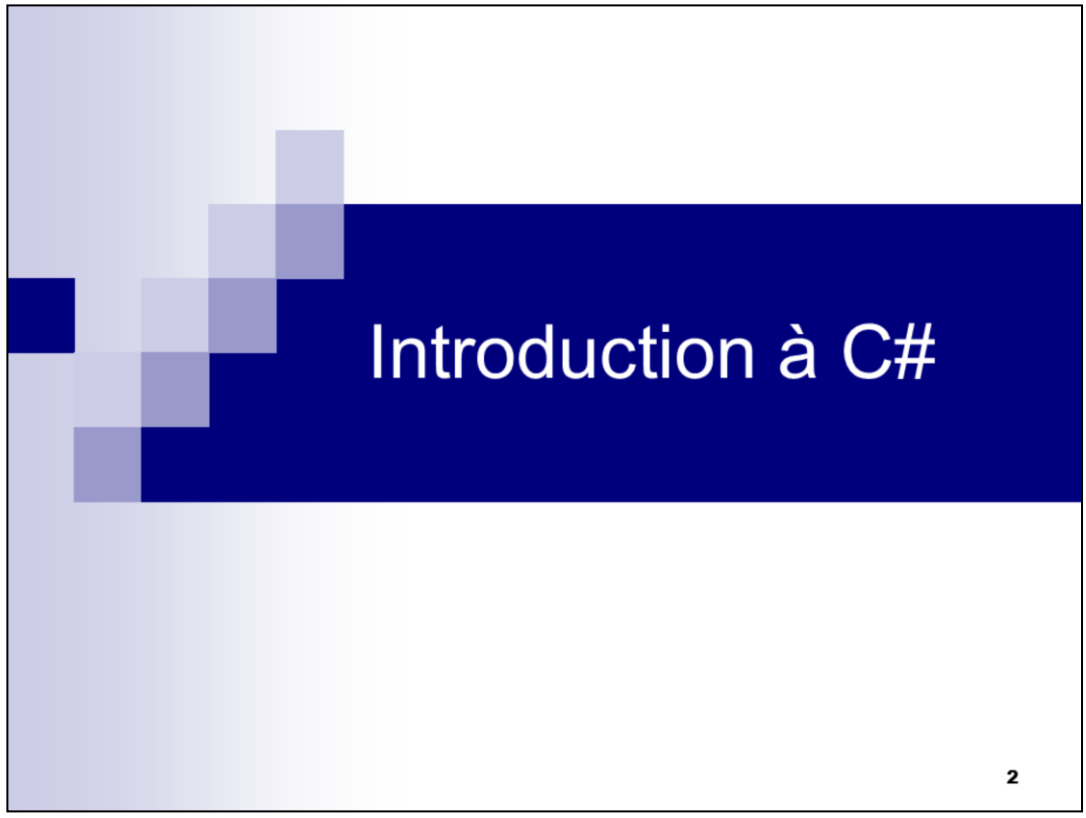
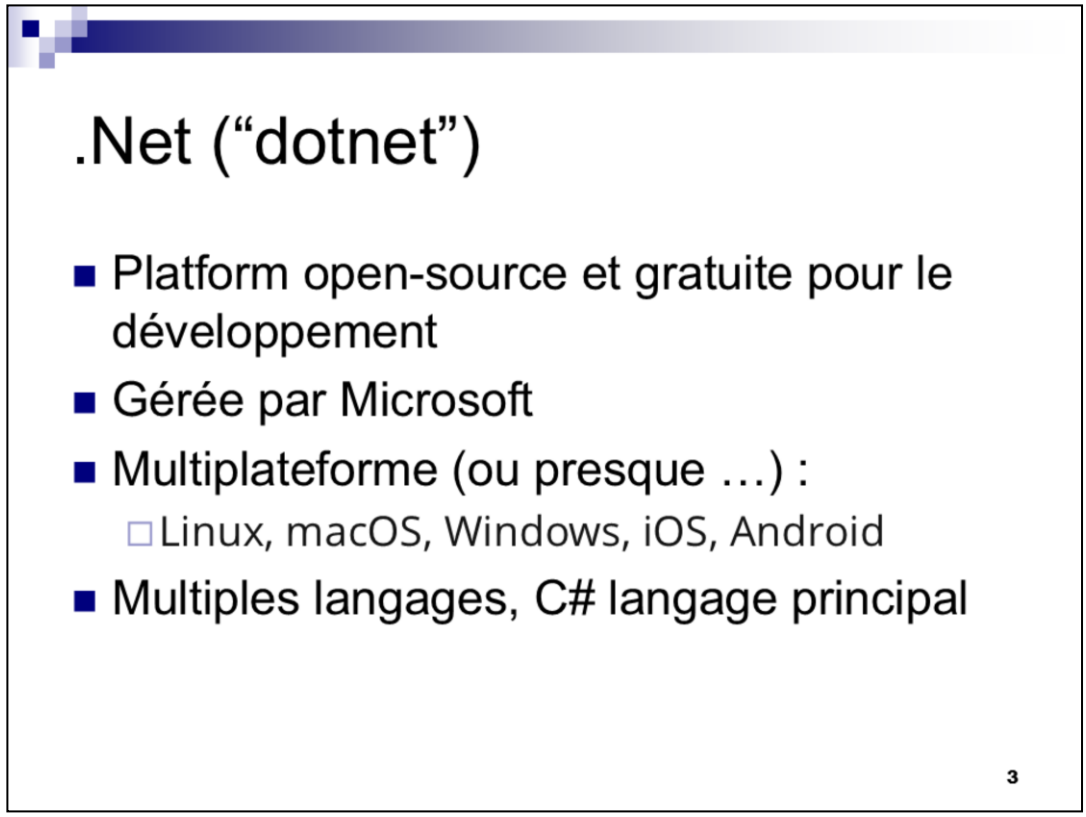
Interfaces Interactives Avancées

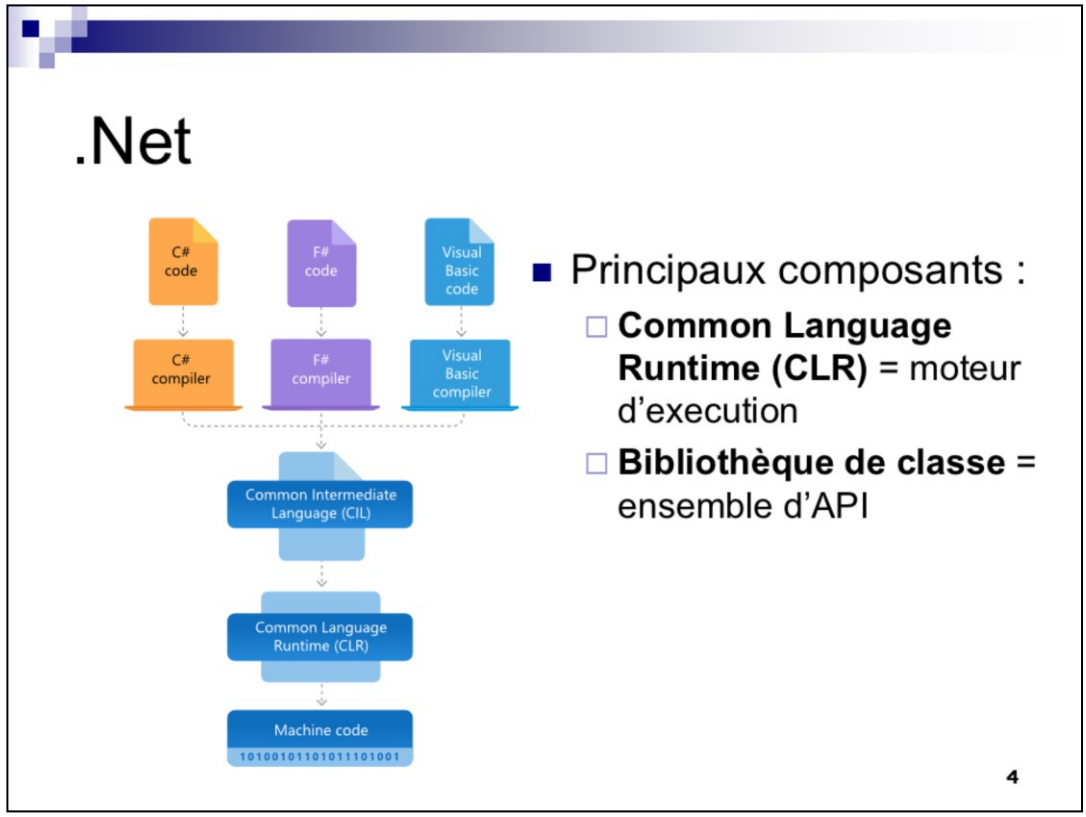
Ouriel Grynszpan
Professeur, Université Paris-Saclay
LISN lisn.upsaclay.fr

1


Introduction à C#
2

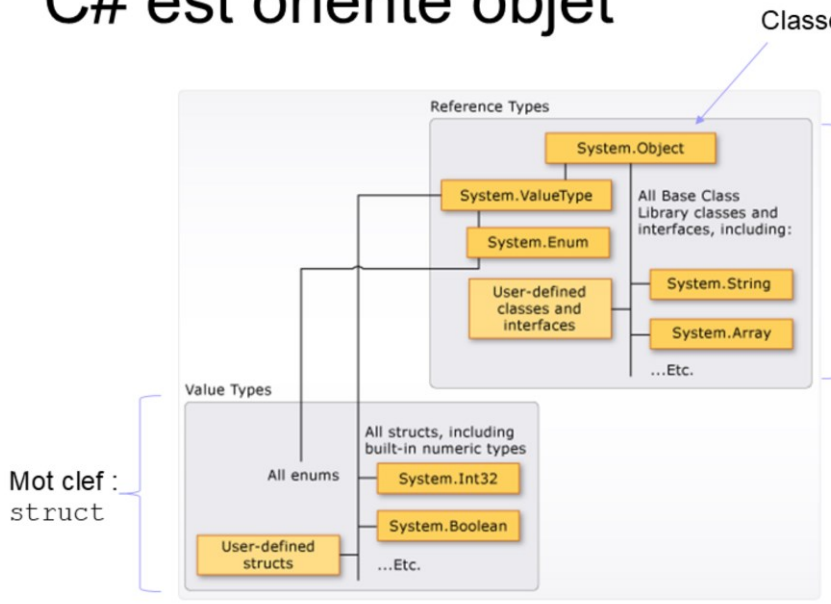


La librairie que nous utiliserons pour créer des interfaces n’est malheureusement disponible que pour Windows.



Les applications .NET sont écrites en langage de programmation C#, F# ou Visual Basic. Le code C#, F# ou Visual Basic est compilé dans un Common Intermediate Language indépendant du langage (CIL). Ce code est stocké dans des « assembly ». Lorsqu'un programme s'exécute, le CRL prend l'assembly et le convertit en code machine spécifique à la plateforme sur laquelle le programme s'exécute.

C# est orienté objet



Mots clefs : `class`

Mot clef : `struct`

5

Syntaxe ~ Java et C++

- Mêmes intructions conditionnelles (if, else, switch)
- Mêmes instructions de boucles (for, while, do…while) + foreach

```
foreach (type variable in collection)
{
      instruction…
}
```

6

Types primaires

C# type keyword	.NET type
bool	System.Boolean
byte	System.Byte
sbyte	System.SByte
char	System.Char
decimal	System.Decimal
double	System.Double
float	System.Single

C# type keyword	.NET type
int	System.Int32
uint	System.UInt32
nint	System.IntPtr
nuint	System.UIntPtr
long	System.Int64
ulong	System.UInt64
short	System.Int16
ushort	System.UInt16

7

Tableaux et listes

- Tableau :

```
int[] T = new int[5]
int[][] M = new int[5][10]
```

Base abstract class:
`System.Array`

- Liste :
  - Classe `List<T>` où `T` est le type d'éléments
  - Namespace:
    `System.Collections.Generic`

8

Point d'entrée

- Possibilité d'instructions 'top-level' :

```
// This line prints "Hello, World"
Console.WriteLine("Hello, World");
```

Nous ne l'utiliserons pas

- Ou plus classiquement :

```
using System;

class Hello
{
    static void Main()
    {
        // This line prints "Hello, World"
        Console.WriteLine("Hello, World");
    }
}
```

Point d'entrée

9

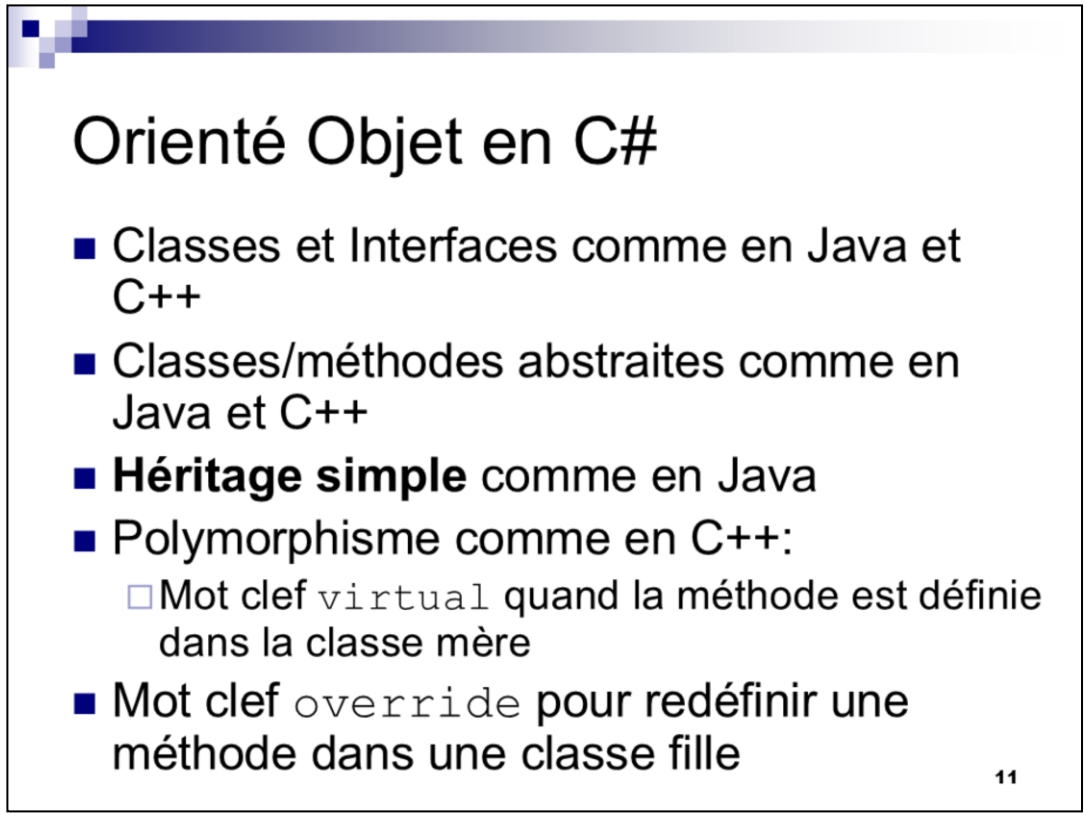
Les deux modes sont équivalentes : Le compilateur synthétise les instructions 'top-level et la méthode Main de la classe principale.

Lecture/écriture dans fichier

```
using System.IO;

string a_ecrire = "Il était une fois...";
File.WriteAllText("nom_fichier.txt", a_ecrire);

string a_lire =
File.ReadAllText("nom_fichier.txt");
Console.WriteLine(a_lire);
```

10

On utilise la classe File du namespace System.IO



En C#, une classe ne peut hériter que d'une seule classe.

Classe simple

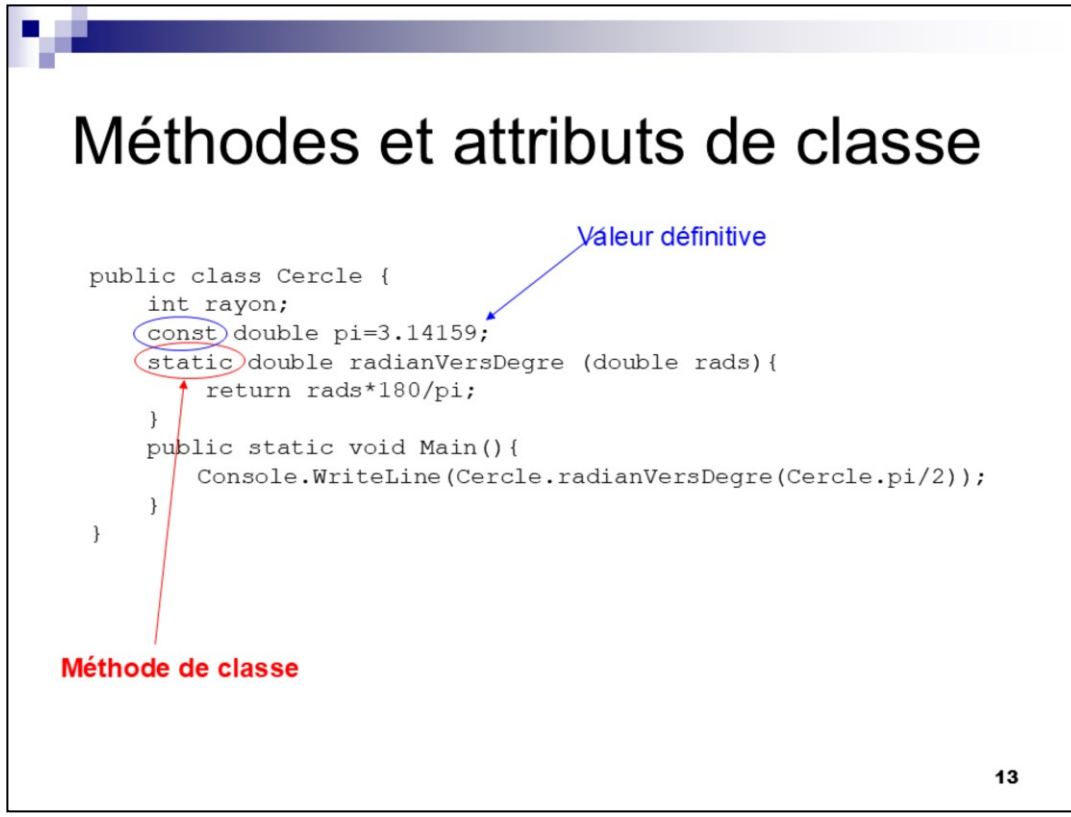
```
public class Rectangle {
    int hauteur;
    int largeur;
    void setHauteur(int h){
        this.hauteur=h;
    }
    void setLargeur(int l){
        this.largeur=l;
    }
    double getAire(){
        return hauteur*largeur;
    }

    public static void Main(String[] args){
        Rectangle rectangle=new Rectangle();
        rectangle.setHauteur(5);
        rectangle.setLargeur(12);
        double aire=rectangle.getAire();
        Console.WriteLine("L'aire du rectangle est " + aire);
    }

}
```

Champs

Méthodes

instanciation

12



Pour rappel, des champs ou méthodes peuvent être définis directement au niveau de la classe, avec le mot clef ‘static’. Ils n’ont donc pas besoin d’une instance de la classe pour être utilisés. Le modificateur static n’est pas autorisé dans une déclaration constante.

Hiérarchie

```
class Rectangle : Forme
{
    int hauteur;
    int largeur;
    void setHauteur(int h){
        this.hauteur=h;
    }
    void setLargeur(int l){
        this.largeur=l;
    }
    double getAire(){
        return hauteur*largeur;
    }
    public static void Main(String[] args){
        Rectangle rectangle=new Rectangle();
        rectangle.setHauteur(5);
        rectangle.setLargeur(12);
        rectangle.setX(30);
        rectangle.setY(40);
        double aire=rectangle.getAire();
        Console.WriteLine("L'aire du rectangle est "
        + aire);
    }

}
```

```
class Forme {
    int x;
    int y;
    protected void setX(int x){
        this.x=x;
    }
    protected void setY(int y){
        this.y=y;
    }
}
```

Constructeur

```
class Rectangle : Forme
{
    int hauteur;
    int largeur;
    void setHauteur(int h){
        this.hauteur=h;
    }
    void setLargeur(int l){
        this.largeur=l;
    }
    double getAire(){
        return hauteur*largeur;
    }
    public static void Main(String[] args){
        Rectangle rectangle=new Rectangle();
        rectangle.setHauteur(5);
        rectangle.setLargeur(12);
        rectangle.setX(30);
        rectangle.setY(40);
        double aire=rectangle.getAire();
        Console.WriteLine("L'aire du rectangle est "
        + aire);
    }

}
```

```
class Forme {
    int x;
    int y;
    protected void setX(int x){
        this.x=x;
    }
    protected void setY(int y){
        this.y=y;
    }
    protected Forme (){
        this.x=0;
        this.y=0;
    }
}
```

constructeur

Constructeurs multiples

```
class Rectangle : Forme
{
    int hauteur;
    int largeur;
    void setHauteur(int h){
        this.hauteur=h;
    }
    void setLargeur(int l){
        this.largeur=l;
    }
    double getAire(){
        return hauteur*largeur;
    }

    public static void Main(String[] args){
        Rectangle rectangle=new Rectangle();
        rectangle.setHauteur(5);
        rectangle.setLargeur(12);
        rectangle.setX(30);
        rectangle.setY(40);
        double aire=rectangle.getAire();
        Console.WriteLine("L'aire du rectangle est "
         + aire);
    }

}
```

```
class Forme {
    int x;
    int y;
    protected void setX(int x){
        this.x=x;
    }
    protected void setY(int y){
        this.y=y;
    }

    protected Forme (){
        this.x=0;
        this.y=0;
    }

    protected Forme (int x, int y)
{
        this.x=x;
        this.y=y;
    }
}
```

16

Les constructeurs se différencient par leurs paramètres.

Héritage des constructeurs

```
class Rectangle : Forme
{
    int hauteur;
    int largeur;
    void setHauteur(int h){
        this.hauteur=h;
    }
    void setLargeur(int l){
        this.largeur=l;
    }
    double getAire(){
        return hauteur*largeur;
    }

    public static void Main(String[] args){
        Rectangle rectangle=new Rectangle();
        rectangle.setHauteur(5);
        rectangle.setLargeur(12);
        rectangle.setX(30);
        rectangle.setY(40);
        double aire=rectangle.getAire();
        Console.WriteLine("L'aire du rectangle est "
         + aire);
    }
}
```

Attention bug →

```
class Forme {
    int x;
    int y;
    protected void setX(int x){
        this.x=x;
    }
    protected void setY(int y){
        this.y=y;
    }

    /*protected Forme (){
        this.x=0;
        this.y=0;
    }*/

    protected Forme (int x, int y)
{
        this.x=x;
        this.y=y;
```

error CS7036: Parmi les arguments spécifiés, aucun ne correspond au paramètre obligatoire 'x' de 'Forme.Forme(int, int)'

17

Appel méthodes classe parent

```
class Rectangle : Forme
{
    int hauteur;
    int largeur;
    Rectangle (int x, int y , int h, int l) : base (x,y)
    {
        this.hauteur = h;
        this.largeur = l;
    }
    void setHauteur(int h){
        this.hauteur=h;
    }
    void setLargeur(int l){
        this.largeur=l;
    }
    double getAire(){
        return hauteur*largeur;
    }

    public static void Main(String[] args){
        Rectangle rectangle=new Rectangle(30,40,5,12);
        double aire=rectangle.getAire();
        Console.WriteLine("L'aire du rectangle est "
        + aire);
    }
}
```

Mot clef : **base**

```
class Forme {
    int x;
    int y;
    protected void setX(int x){
        this.x=x;
    }
    protected void setY(int y){
        this.y=y;
    }

    protected Forme (){
        this.x=0;
        this.y=0;
    }

    protected Forme (int x, int y) {
        this.x=x;
        this.y=y;
    }
}
```

18

Pour faite appel à une méthode de la classe parent, on utilise le mot clef « base (paramètres) » pour les constructeurs et base.method(..) pour des méthodes particulières.

Réécriture de méthodes (1/2)

Un changement de type de l'instance sur la classe mère ne modifie pas les méthodes réécrites ➔ polymorphisme

```
class Rectangle : Forme
{
    int hauteur;
    int largeur;
    void setHauteur(int h){
        this.hauteur=h;
    }
    void setLargeur(int l){
        this.largeur=l;
    }
    double getAire(){
        return hauteur*largeur;
    }
    protected override void setX(int x){
        this.x=x+this.largeur/2;
    }
    protected override void setY(int y){
        this.y=y+this.hauteur/2;
    }
```

```
class Forme {
    protected int x;
    protected int y;
    protected virtual void setX(int x){
        this.x=x;
    }
    protected virtual void setY(int y){
        this.y=y;
    }
    public int getX(){return this.x;}
    public int getY(){return this.y;}
}
```

Associé directement à l'instance

19

Les polymorphismes sont associés directement à l'instance de classe. Donc si on change le type d'une instance, ses méthodes restent les mêmes.

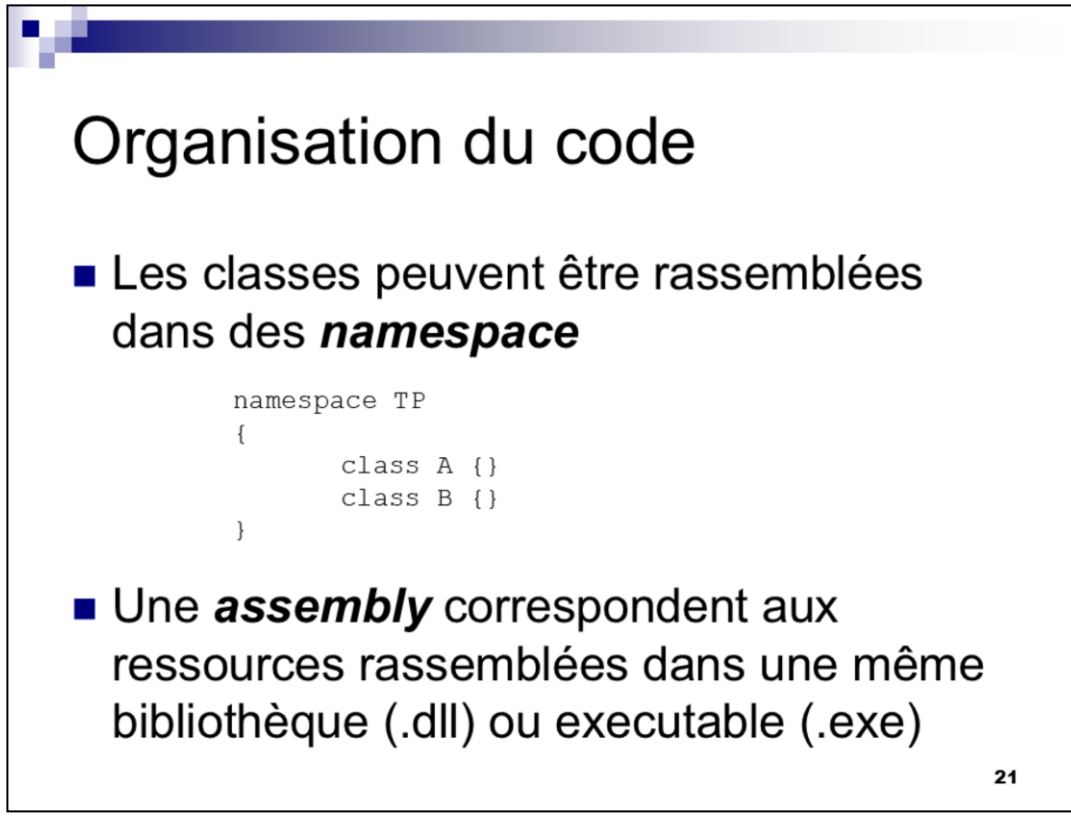
Réécriture de méthodes (2/2)

Un changement de type de l'instance sur la classe mère ne modifie pas les méthodes réécrites ➔ polymorphisme

```
    public static void Main(String[] args){
        Rectangle rectangle=new Rectangle();
        rectangle.setLargeur(12);
        rectangle.setHauteur(6);
        rectangle.setX(30);
        rectangle.setY(40);
        Forme forme=rectangle;
        Console.WriteLine("Abscisse du centre de la forme = " + forme.getX());
        Console.WriteLine("Ordonnée du centre de la forme = " + forme.getY());
    }

}
```

Résultat :

```
Abscisse du centre de la forme = 36
Ordonnée du centre de la forme = 43
```



L'assembly se définit avec des outils de développement, typiquement Visual Studio.

using

- Permet d'utiliser des types définis dans un *namespace* sans spécifier le nom complet du *namespace*
- Un peu comme *import* en Java ou Python

En début de fichier →

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

Modificateurs d'accessibilité

Emplacement de l'appelant	public	protected	internal	private
Dans la classe	✓	✓	✓	✓
Classe dérivée (même assembly)	✓	✓	✓	✗
Classe non dérivée (même assembly)	✓	✗	✓	✗
Classe dérivée (assembly différent)	✓	✓	✗	✗
Classe non dérivée (assembly différent)	✓	✗	✗	✗

Modificateur « Private »

```
class Rectangle : Forme
{
    int hauteur;
    int largeur;
    void setHauteur(int h){
        this.hauteur=h;
    }
    void setLargeur(int l){
        this.largeur=l;
    }
    double getAire(){
        return hauteur*largeur;
    }
    protected override void setX(int x){
        this.x=x+this.largeur/2;
    }
    protected override void setY(int y){
        this.y=y+this.hauteur/2;
    }
```

```
class Forme {
    private int x;
    private int y;
    protected virtual void setX(int x){
        this.x=x;
    }
    protected virtual void setY(int y){
        this.y=y;
    }
    public int getX(){return this.x;}
    public int getY(){return this.y;}
}
```

```
error CS0122: 'Forme.x' est inaccessible en raison de son niveau de protection
error CS0122: 'Forme.y' est inaccessible en raison de son niveau de protection
```

24

Modificateur « Protected »

```
class Rectangle : Forme
{
    int hauteur;
    int largeur;
    void setHauteur(int h){
        this.hauteur=h;
    }
    void setLargeur(int l){
        this.largeur=l;
    }
    double getAire(){
        return hauteur*largeur;
    }
    protected override void setX(int x){
        this.x=x+this.largeur/2;
    }
    protected override void setY(int y){
        this.y=y+this.hauteur/2;
    }
```

```
class Forme {
    protected int x;
    protected int y;
    protected virtual void setX(int x){
        this.x=x;
    }
    protected virtual void setY(int y){
        this.y=y;
    }
    public int getX(){return this.x;}
    public int getY(){return this.y;}
}
```

Pas d'erreur

25

Propriétés

- Propriété

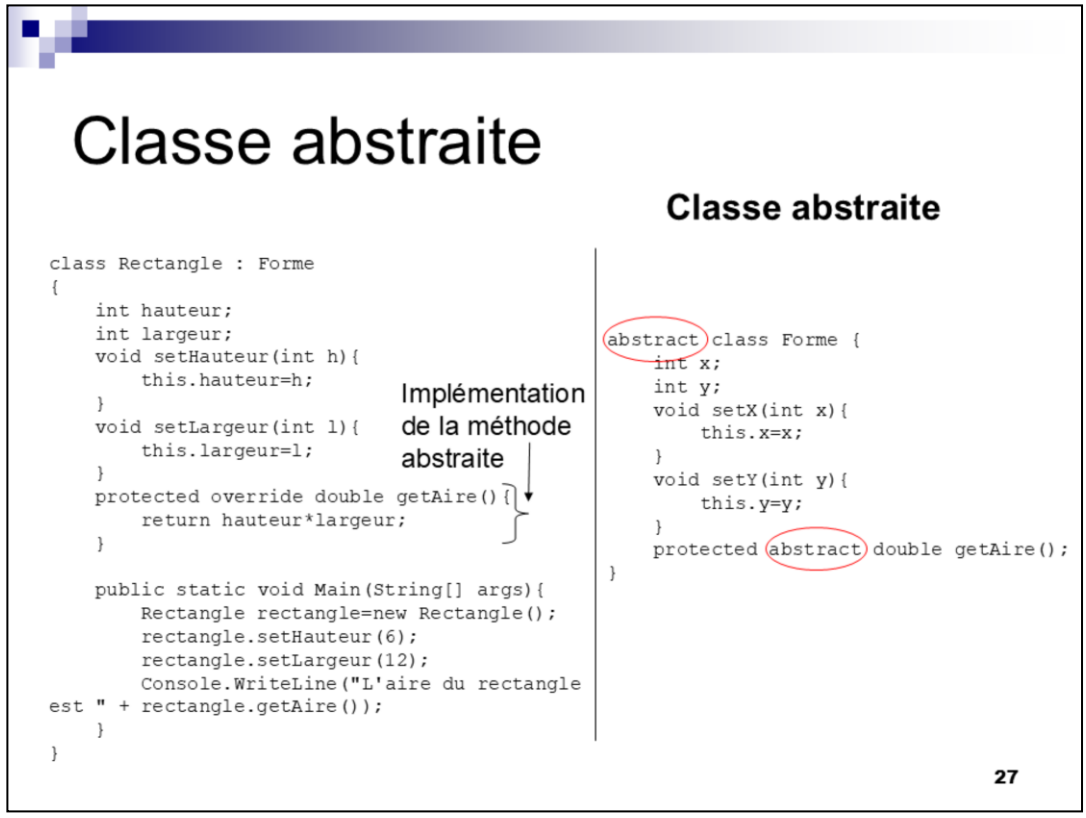
```
class Person{
       private int age; // Field "age" declared
       public int Age { // Property for age
              get{return age;}
              set{age = value;}
       }
}
```

- Propriété créée automatiquement

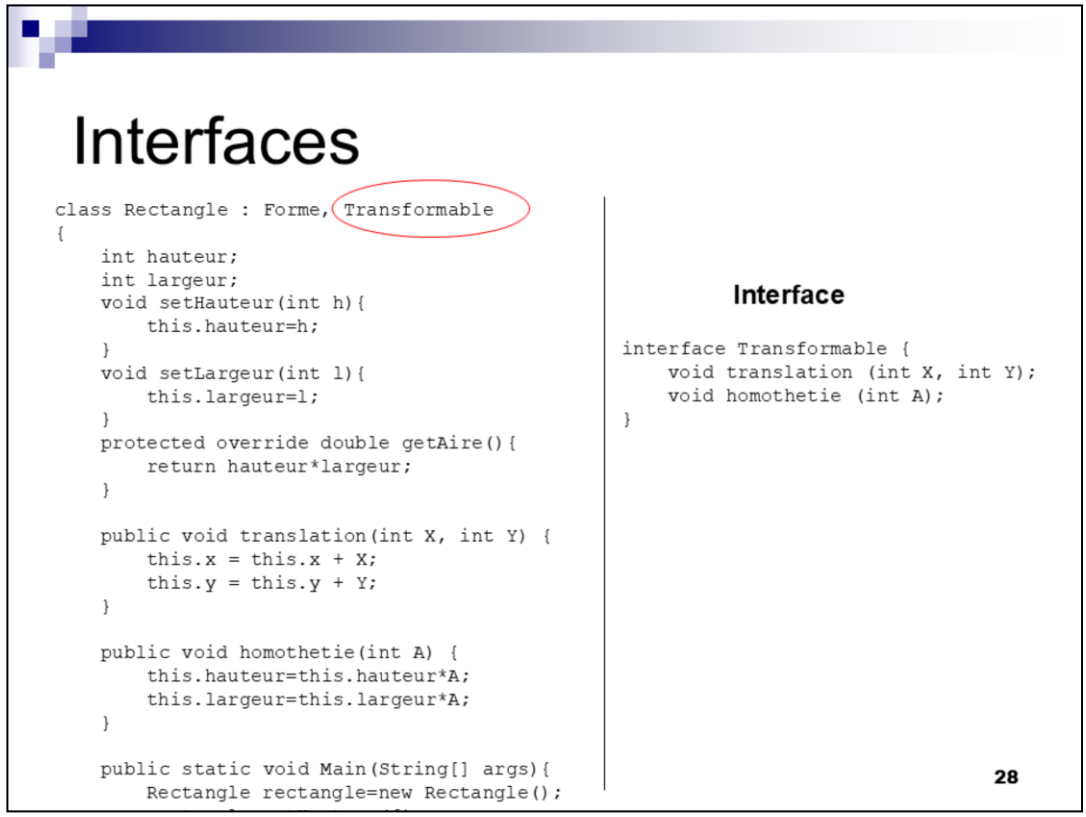
```
class Person{
       public int Age // Property for age
       { get; set; }
}
```

26

Classe abstraite



Si une classe contient une méthode abstraite, elle est abstraite et doit être déclarée abstraite. Une sous-classe d'une classe abstraite doit implémenter les méthodes déclarées abstraites pour pouvoir être instanciée.



Une interface est un peu comme une classe abstraite dont toutes les méthodes sont abstraites. Une classe peut implémenter plusieurs interfaces. Les membres d'une interface sont publics par défaut.

Exception

- Signale d'une condition exceptionnelle ou une erreur
- Se propage à travers les blocs et méthodes jusqu'à ce qu'elle soit « attrapée », sinon termine l'exécution avec erreur.
- Classes dérivant de *System.Exception*
- Mots clefs:
  - *throw* (lance l'exception)
  - *try/catch* (pour traiter l'exception)

29

```
class Rectangle
{
    int hauteur;
    int largeur;
    void setHauteur(int h){
        if (h<0) {
            throw new Exception("Hauteur négative non permise!");
        }
        this.hauteur=h;
    }
    void setLargeur(int l){
        this.largeur=l;
    }
    double getAire(){
        return hauteur*largeur;
    }

    public static void Main(String[] args){
        Rectangle rectangle=new Rectangle();
        try {
            rectangle.setHauteur(-6);
        }
        catch(Exception e){
            Console.WriteLine("Exception attrapée");
            Console.WriteLine(e.Message);
        }
        rectangle.setLargeur(12);
    }

}
```

Exception lancée

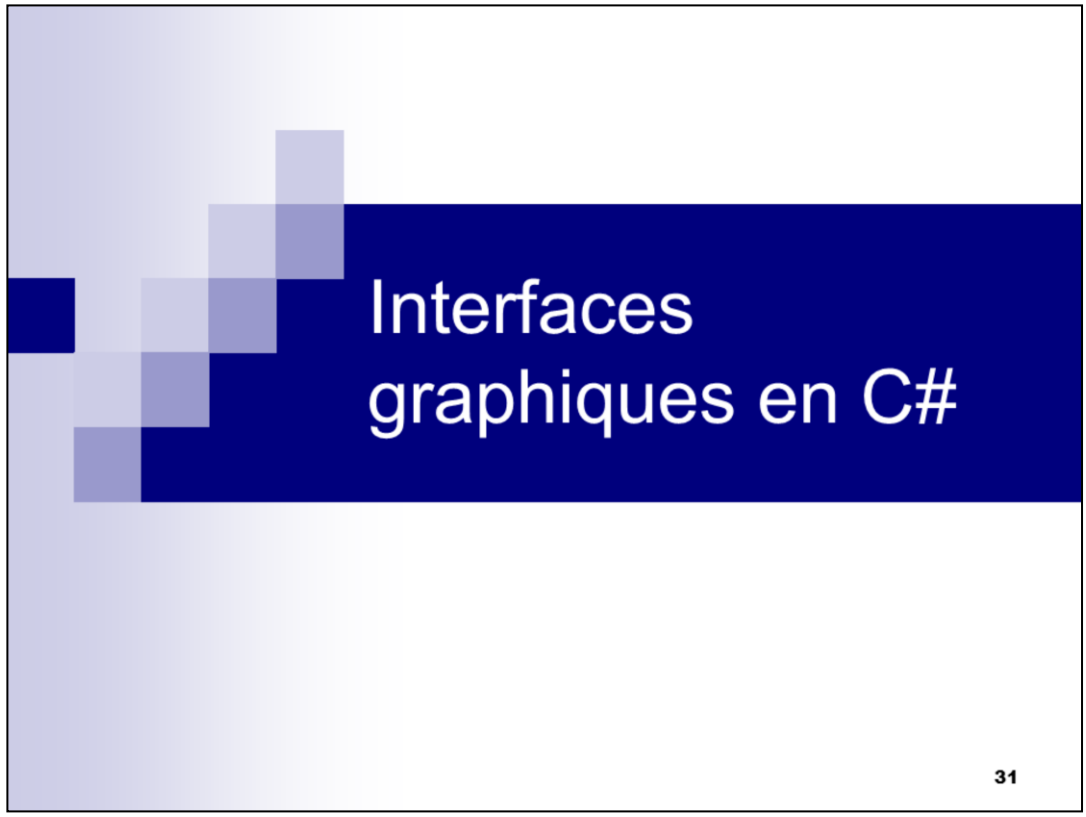
Instruction risquée

Traitement de l'exception

À l'exécution

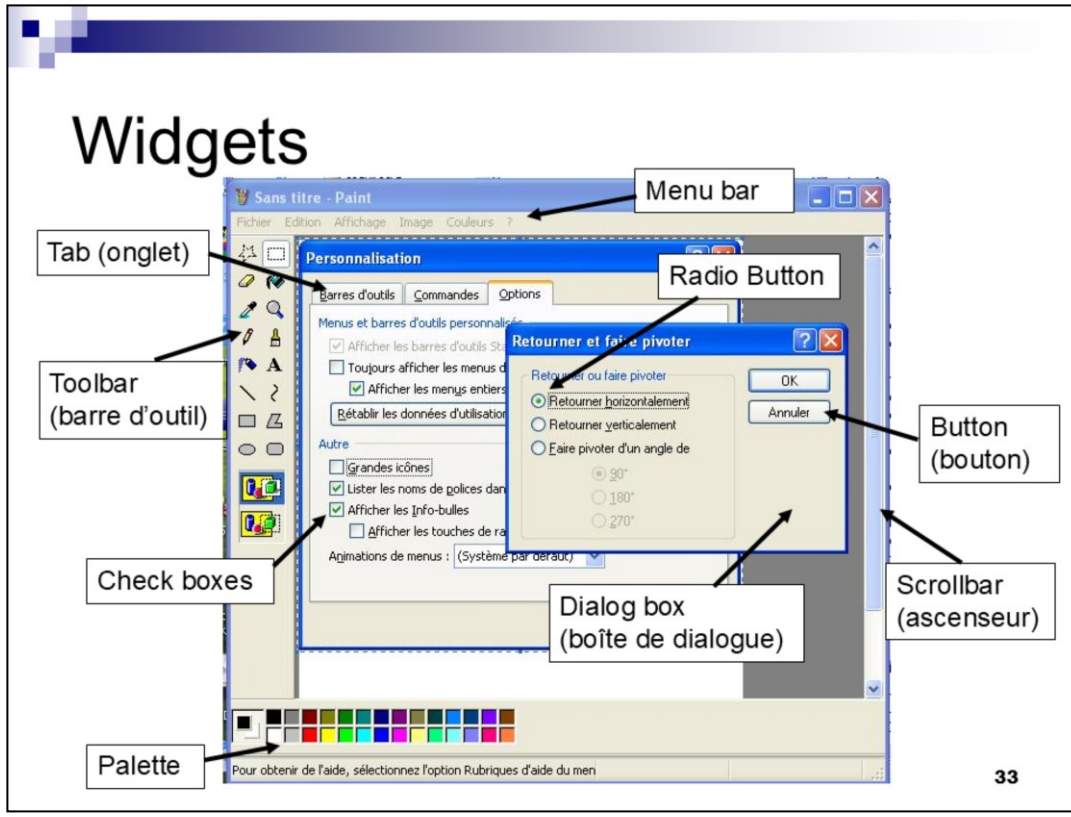
```
Exception attrapée
Hauteur négative non permise!
```

30


Interfaces
graphiques en C#
31

Terminologie

- WIMP: Windows Icons Menus Pointers
- GUI: Graphical User Interface
- Widget:
  - contraction « window + gadget »
  - Élément graphique de base
- API: Application Programming Interface

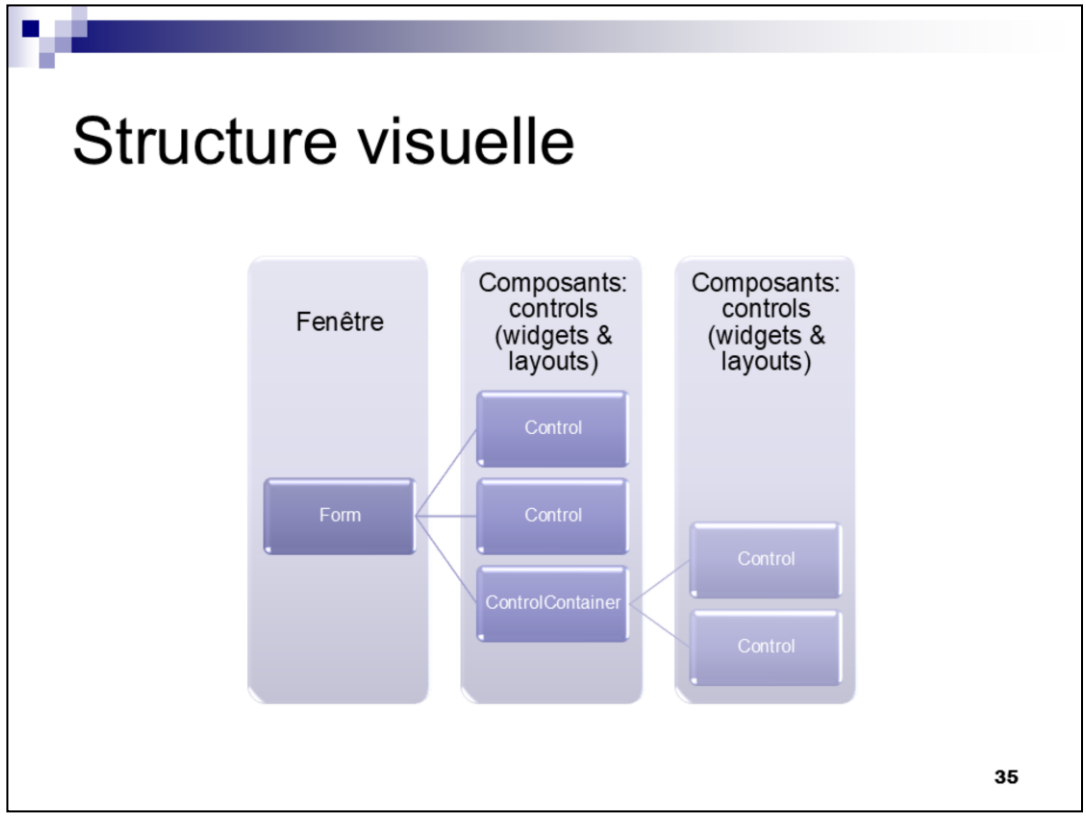


Une interface WIMP est constituée d'éléments appelés widgets : fenêtres (windows) possiblement associée à un ascenseur (scrollbar), icônes, pointeur représenté par un curseur qui peut prendre différentes formes selon le mode courrant, menus, boutons (radio buttons, check boxes), barre d'outils (toolbar), palettes qui sont des collections d'icônes servant à indiquer les modes ou options actuellement sélectionnés, boîtes de dialogue (dialog boxes) repésentées par des fenêtres simples dont le but consiste à attirer l'attention sur un point particulier.

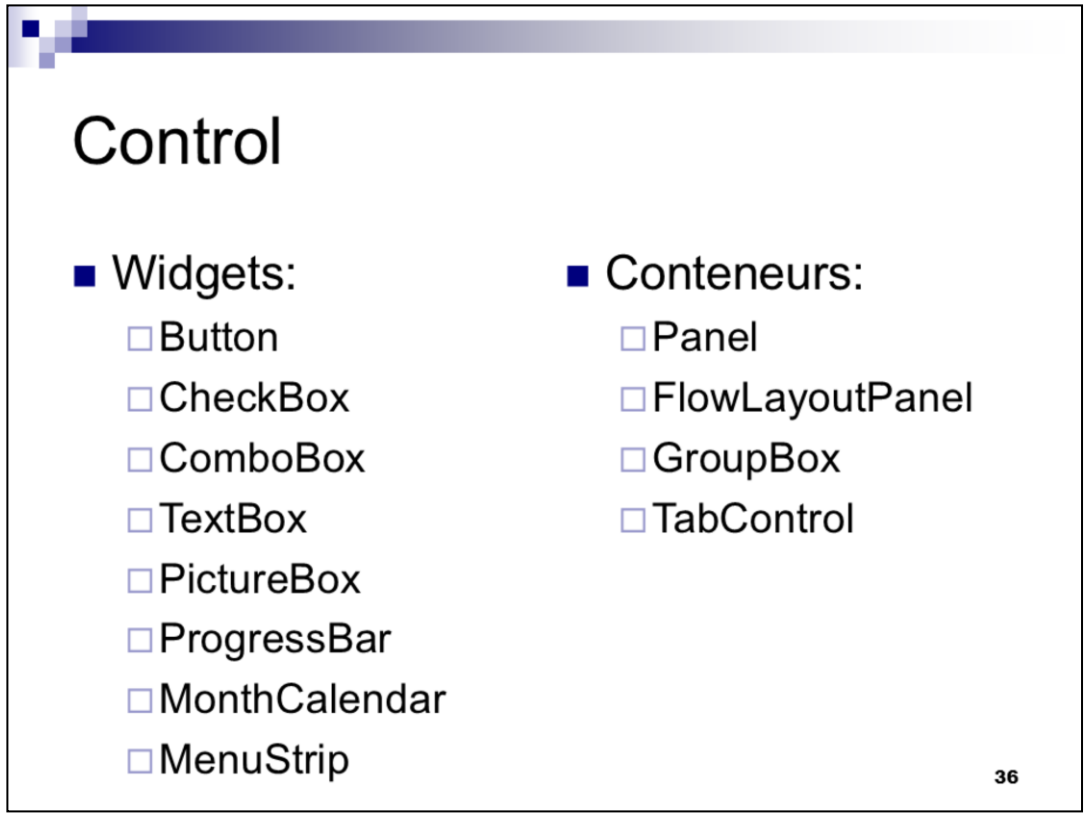
Librairie graphique en C# ?

- **Windows Forms**
  - Réservé pour Windows™
  - Utilise des bibliothèques natives Windows
- Visual Studio
  - Outils d'aide à la conception
  - Génération automatique de code
- Namespace : System.Windows.Forms

34



Les éléments d'une interface graphique s'assemblent dans une hiérarchie de composants. En haut de la hiérarchie, on trouve la fenêtre dont la classe de base est Form. Dans celle-ci, on peut placer différents composants graphiques appelé « contrôles » et qui dérive d'une classe de base Control. Ces contrôles peuvent être des conteneurs d'autres contrôles et ainsi de suite.

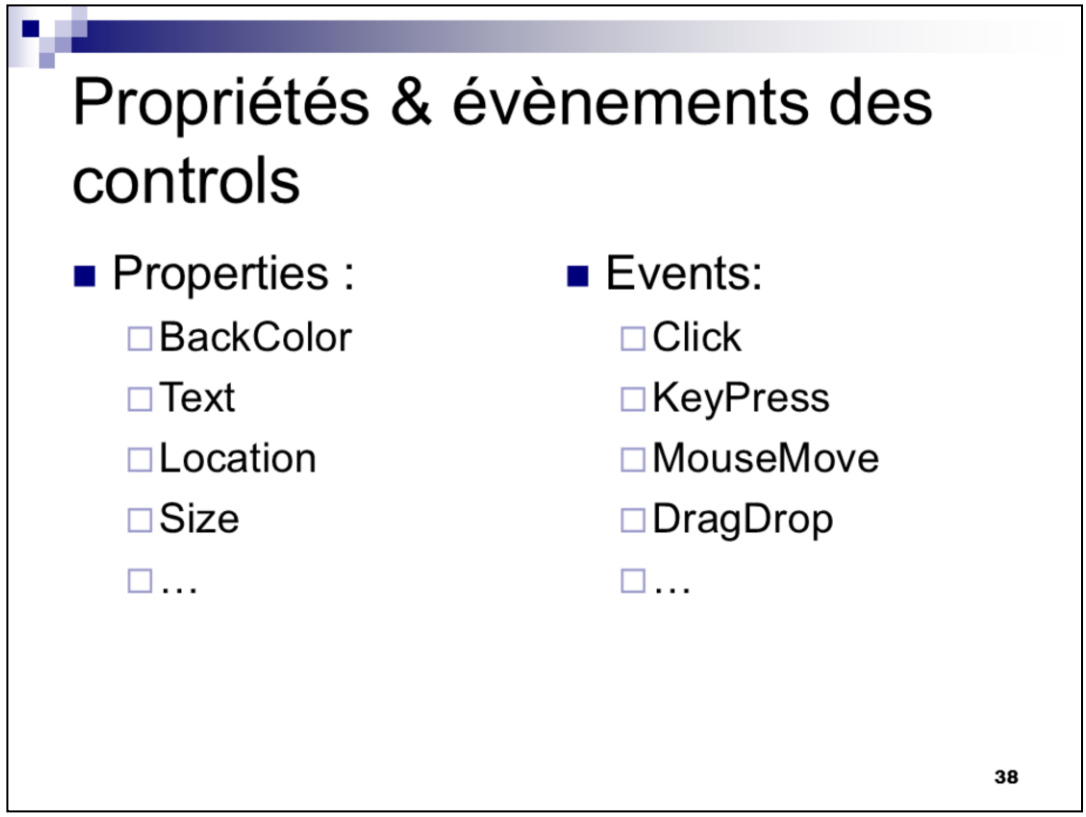


Ci-dessus des exemples de controls et conteneurs. Il s'agit de noms de classes.

Exemple

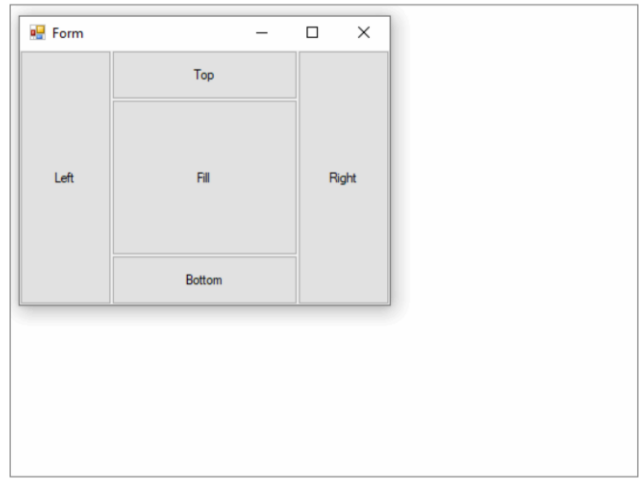

Titre
Délimitation
GroupBox
TabControl
Employee Data
Personal Info
Job Info
Skill
Employee
Select your name
Esther
George
Alan
Eric
Show Summary

37



L'aspect des controls est défini par des propriétés propres aux controls, par exemple BackColor, Text, Location, Size …

Amarage des controls

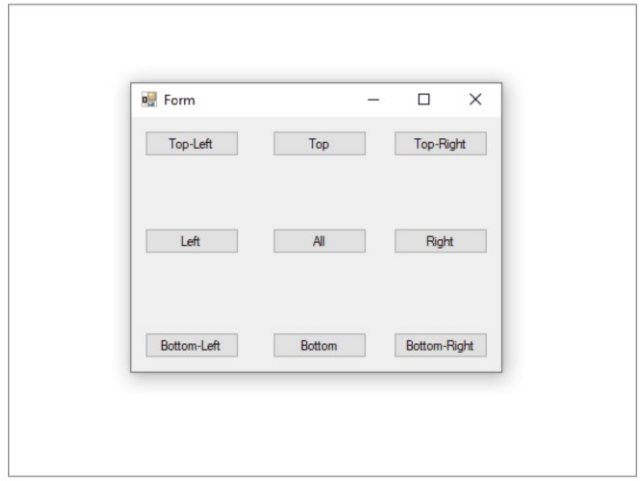


`Dock` property

39

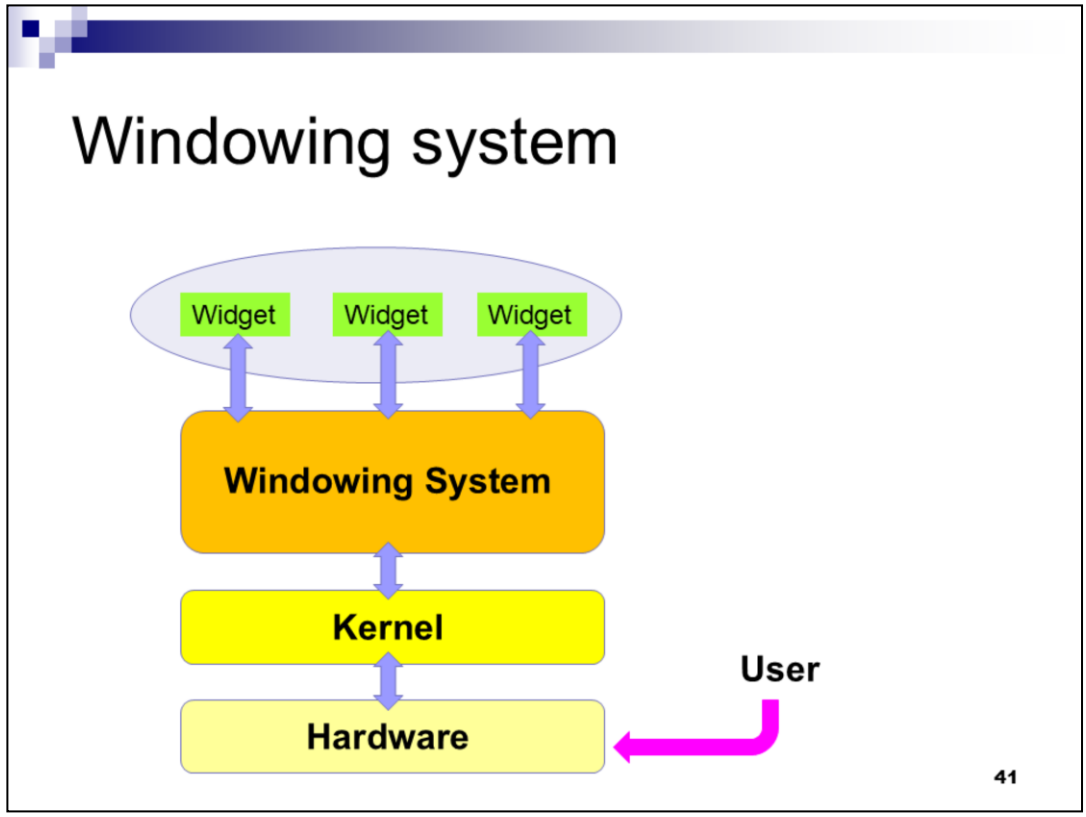
La disposition des contrôles en fonction des modifications de la taille de la fenêtre principale est régulée grâce à des propriétés associées

Encrage des controls



`Anchor` property

40



En IHM, il faut suivre une approche de programmation évènementielle. Les programmes ne suivent pas une logique séquentielle, mais doivent être capables de réagir en fonction des évènements causés par l'utilisateur. On ne peut pas prédire ce que va faire l'utilisateur mais simplement réagir à ses requêtes. Les évènements recueillis par les périphériques d'entrée sont transmis au système d'exploitation et seront gérés par le système de multifenêtrage qui distribuera les requêtes aux différentes instances concernées. Les évènements utilisateur sont mis en files d'attente puis dirigés vers les guichets adéquats par le système de multifenêtrage. Les différents évènements ne seront donc pas forcément traités dès qu'ils sont émis. Cela dépend des autres évènements simultanément émis.

Gestion des évènements

- Widgets ➔ où se déclenchent des évènements
- Windowing System
  ➔ répertorie les évènements
  ➔ gère les files d'attente d'évènements
  ➔ attribue les évènements
- Handler
  ➔ produit les réponses aux évènements
  ➔ reçoit en paramètre un objet représentant l'évènement

42

Traitement des évènements

- Enregistrer une méthode « handler » sur le widget

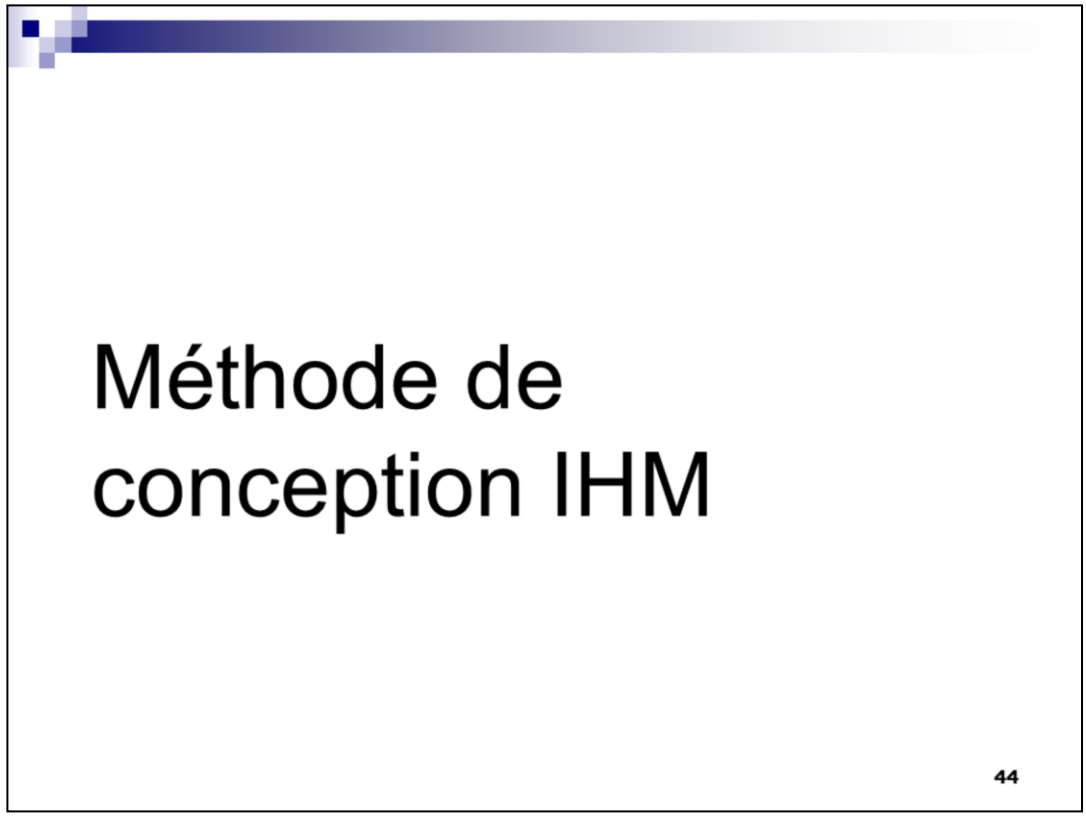
```
private void InitializeComponent()
{
        ...
        button1.Click += button1_Click;
        ...
}
...
private Button button1;

private void button1_Click(object sender, EventArgs e)
{

}
```

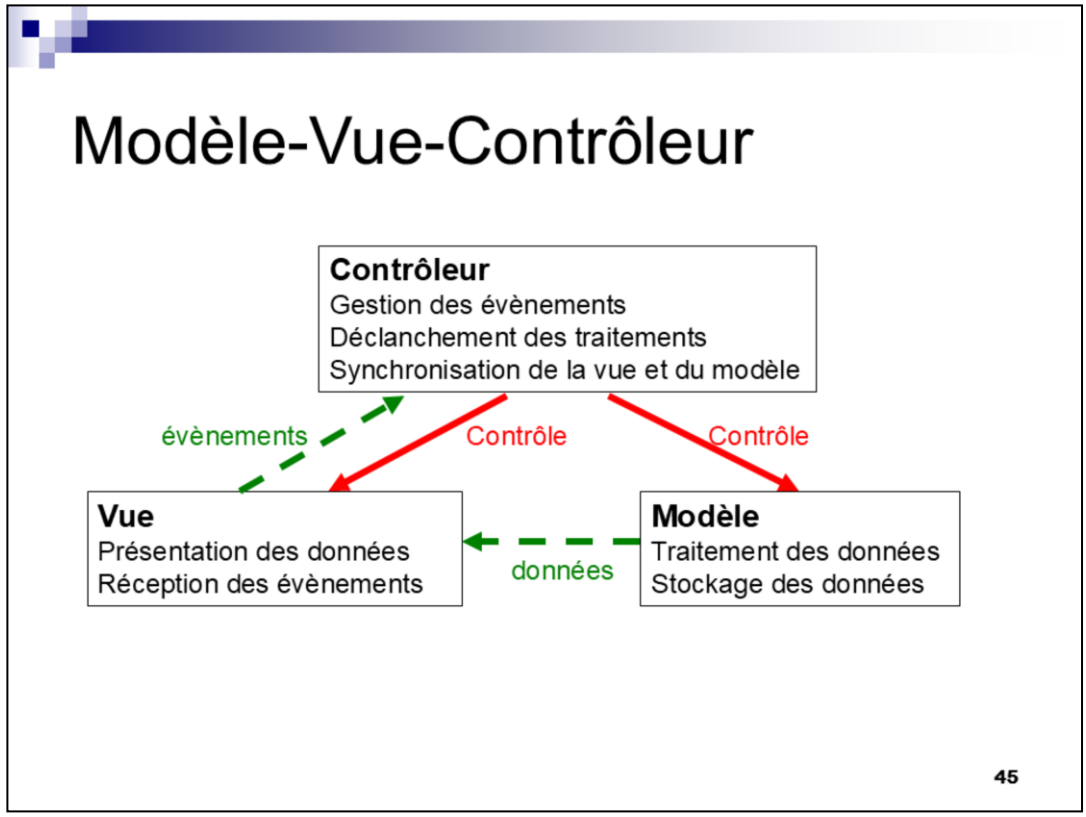
① Enregistrer le handler

② Définir le comportement du handler

43

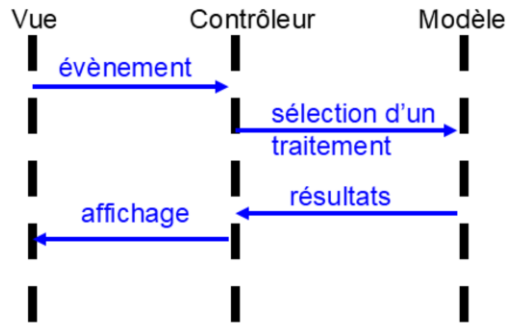
Pour traiter des évènements associés à un widget, il faut ajouter une méthode « handler » à l'évènement correspondant du widget. Les handler d'évènements sont des membres de la classe du widget. Ils sont de type « délégué » qui correspond à une référence à une fonction. On peut additionner plusieurs méthodes sur ces handlers délégués. Lorsqu'un évènement est notifié, ces méthodes sont exécutées dans l'ordre dans lequel elles ont été additionnées.


Méthode de
conception IHM
44



Dans ce cours, nous allons adopter la méthode de conception MVC (Modèle-Vue-Contrôleur). La Vue correspond généralement à l'interface graphique.

Flux des traitements



Avantages du MVC

- Séparer traitement des données et présentation de celles-ci
- Associer plus d'une vue pour un même modèle
- Facilite la maintenance et correction de bugs

47