



# Héritage II

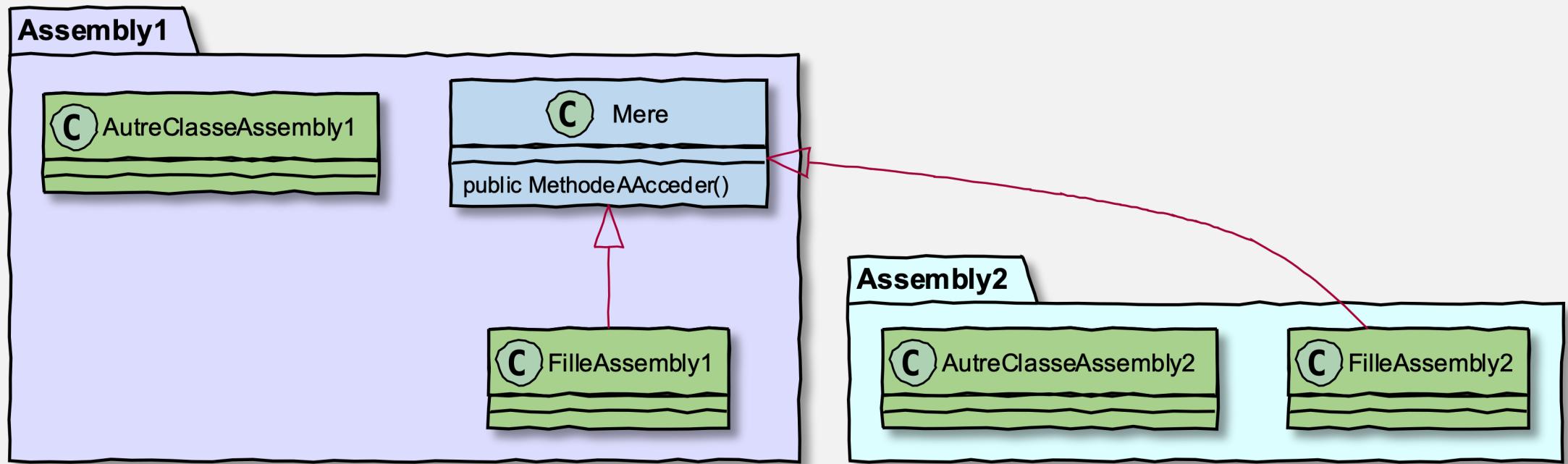
## Accès, abstractions

# Objectifs

- Comprendre les modificateurs d'accès
- Retour sur l'héritage
- Abstractions

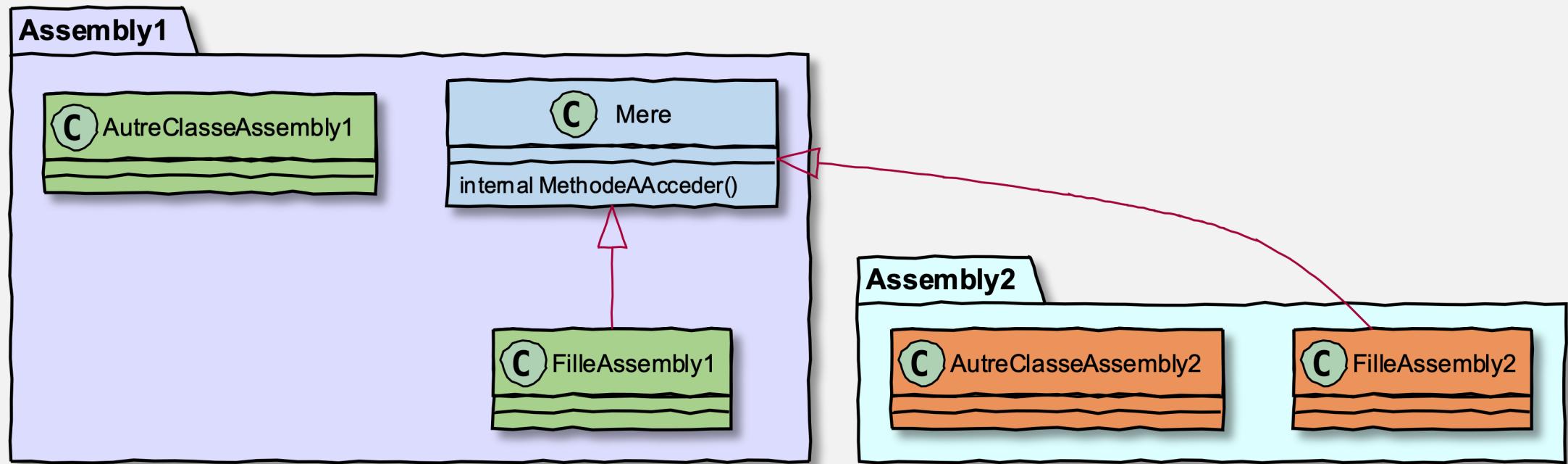
# Accès « public »

- Exemple : accès à la méthode « *MethodeAAcceder* »



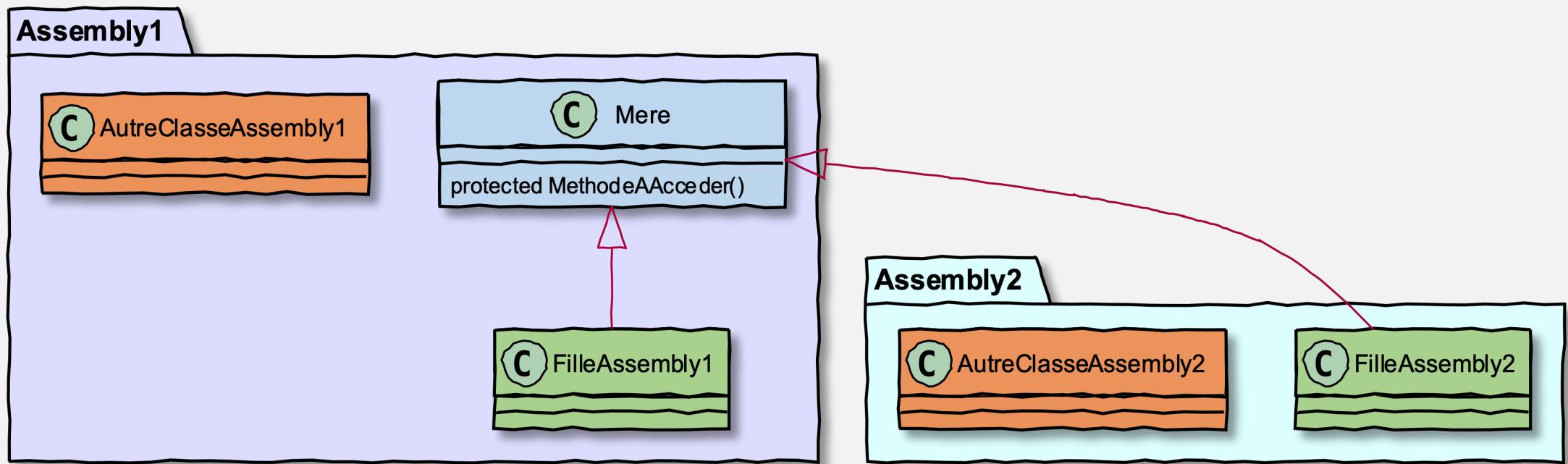
# Accès « internal »

- Exemple : accès à la méthode « *MethodeAAcceder* »



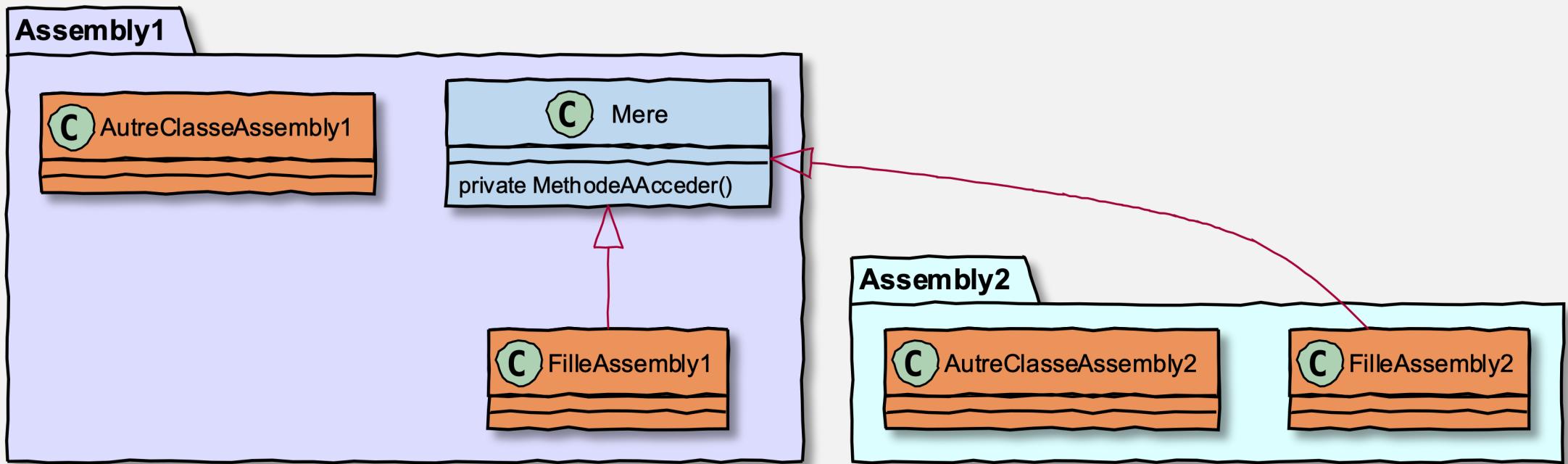
# Accès « protected »

- Exemple : accès à la méthode « *MethodeAAcceder* »



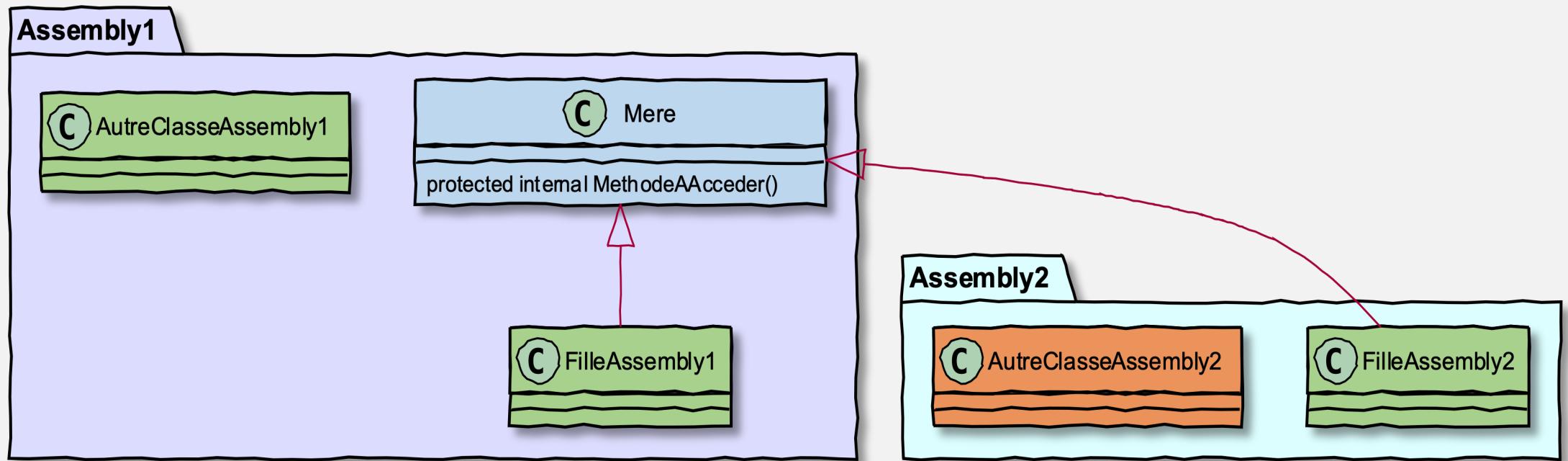
# Accès « private »

- Exemple : accès à la méthode « *MethodeAAcceder* »



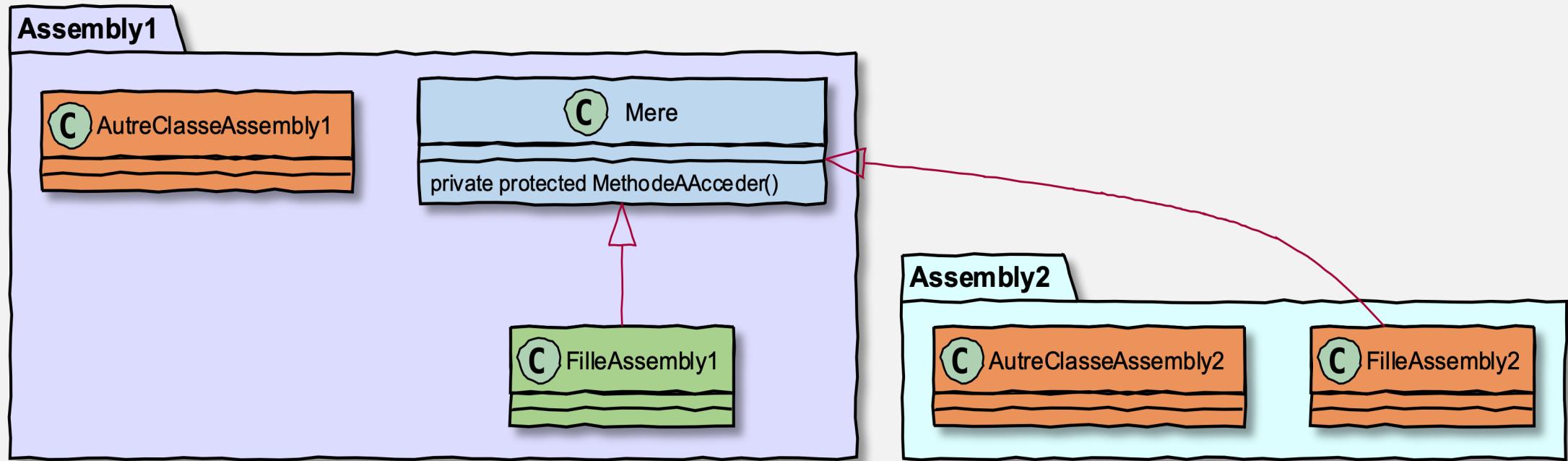
# Accès « protected internal »

- Exemple : accès à la méthode « *MethodeAAcceder* »

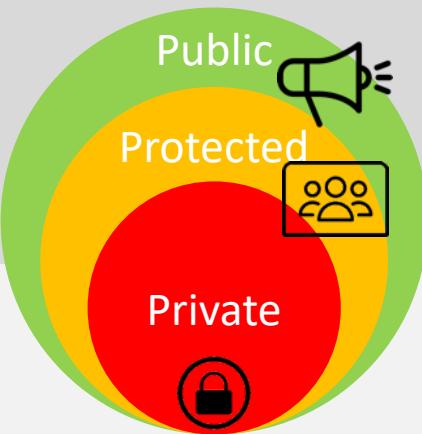


# Accès « private protected »

- Exemple : accès à la méthode « *MethodeAAcceder* »



# Accès aux classes et membres



	Même « assembly » / projet			Autre « assembly » / projet	
	Même classe	Classe fille	Autres classes	Classe fille	Autres classes
public	✓	✓	✓	✓	✓
protected internal	✓	✓	✓	✓	✗
internal	✓	✓	✓	✗	✗
protected	✓	✓	✗	✓	✗
private protected	✓	✓	✗	✗	✗
private	✓	✗	✗	✗	✗

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/access-modifiers>

# Rappels / Compléments : Héritage

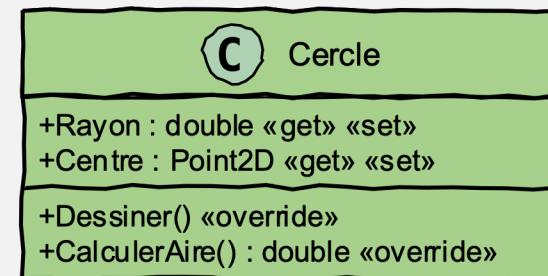
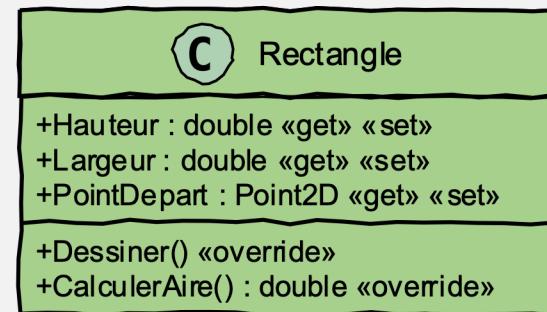
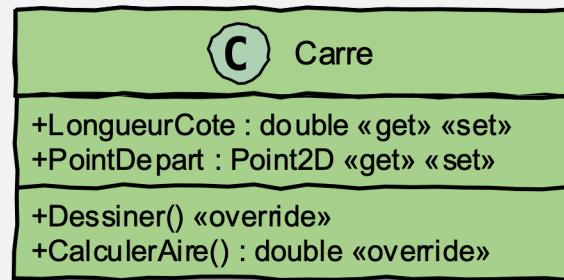
- En programmation orientée objet, l'héritage est un mécanisme qui permet, lors de la déclaration d'une nouvelle classe, d'y inclure les caractéristiques d'une autre classe. (Wikipédia)
- Les caractéristiques sont les données membres et les méthodes
- En C#, Java, etc., comme dans la plupart des langages modernes, l'héritage est simplifié en ne permettant que l'héritage simple
- Si vous n'explicitez aucun héritage pour une classe, elle hérite de la classe de base « Object »
- On peut redéfinir des méthodes existantes avec le mot clef « override » si elles sont marquées « virtual »

# Rappels / Compléments : Polymorphisme

- Le polymorphisme est la possibilité d'avoir plusieurs formes
- Il existe plusieurs formes de polymorphisme :
  - Ad-hoc : surcharge de méthodes (plusieurs méthodes avec le même nom mais des paramètres différents)
  - Par sous-typage (héritage) : redéfinition de méthodes avec les mêmes paramètres. Exemple : `ToString`, `Equals`, etc. et vos propres méthodes virtuelles
  - Paramétrisé (programmation générique) : vous l'utilisez déjà avec des collections paramétrées par un type (ex. une liste), vous allez le programmer en algorithmique avancée

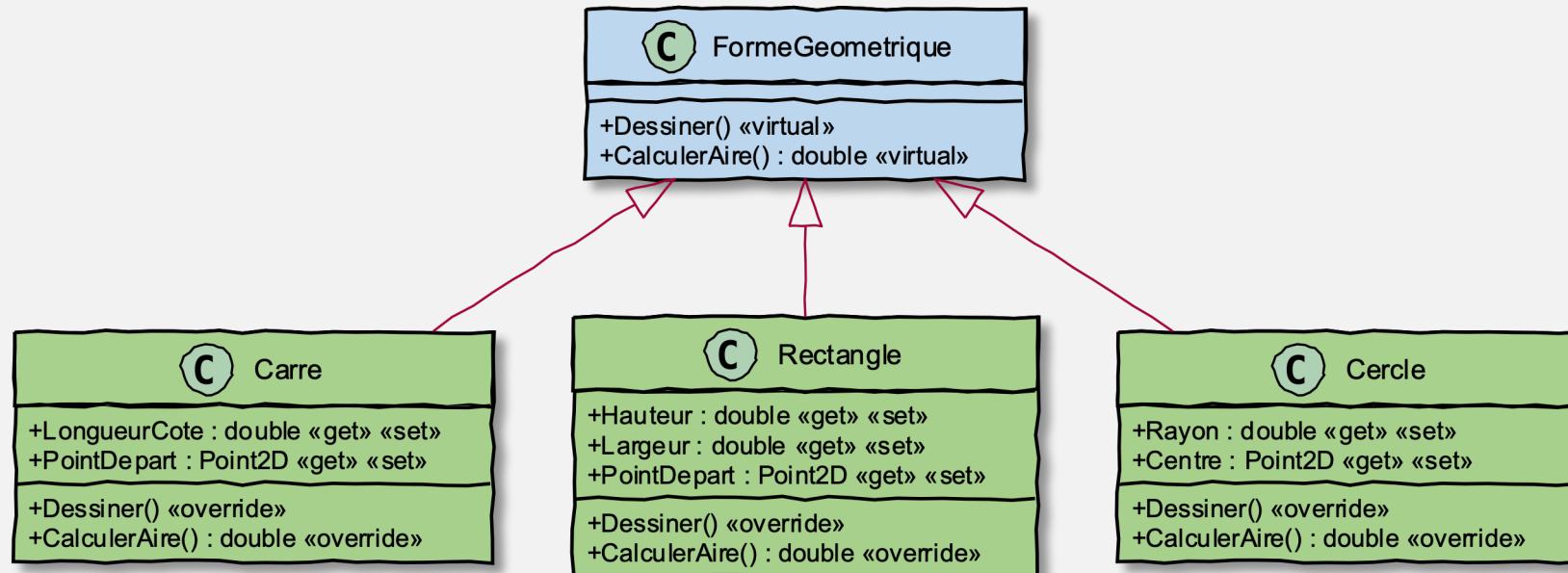
# Exemple pratique : rajoutons un peu d'abstraction

- Nous devons concevoir un programme permettant de manipuler des formes :
  - Des cercles : définis par un centre ( $x, y$ ) et un rayon
  - Des rectangles : définis par un point de départ ( $x, y$ ), une hauteur et une largeur
  - Des carrés : définis par un point de départ ( $x, y$ ) et une longueur de côté
- Nous voulons pouvoir réaliser les opérations suivantes sur les formes :
  - Dessiner() : affichage à l'écran des propriétés
  - CalculerAire() : calculer l'aire de chaque forme

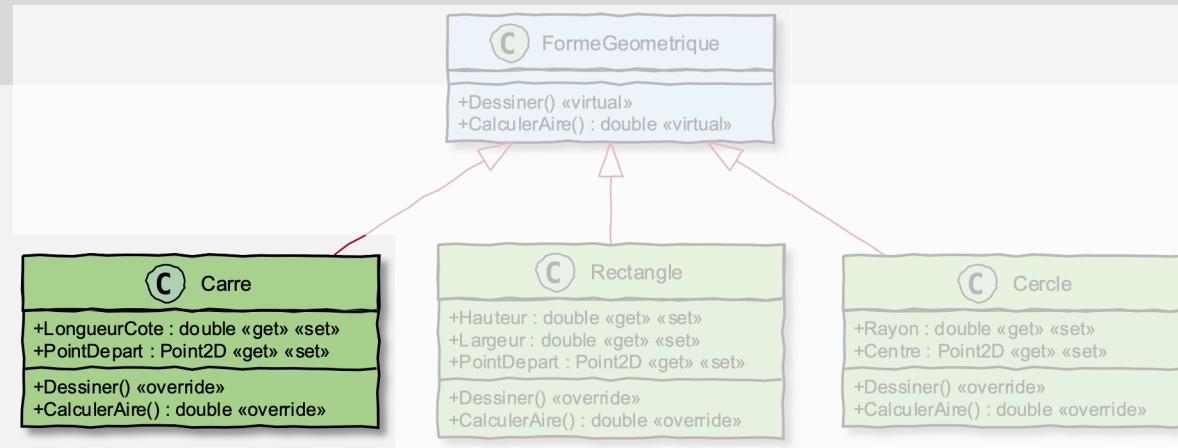


# Exemple pratique : rajoutons un peu d'abstraction

- Ces formes doivent pouvoir être manipulées quelque soit leurs types
- Pour résoudre ce cas, nous allons ajouter la classe artificielle « FormeGeometrique », les classes « Cercle », « Rectangle » et « Carre » vont hériter de « FormeGeometrique »
- Dans le programme, on manipulera donc des « FormeGeometrique »



# Exemple pratique : rajoutons un peu d'abstraction

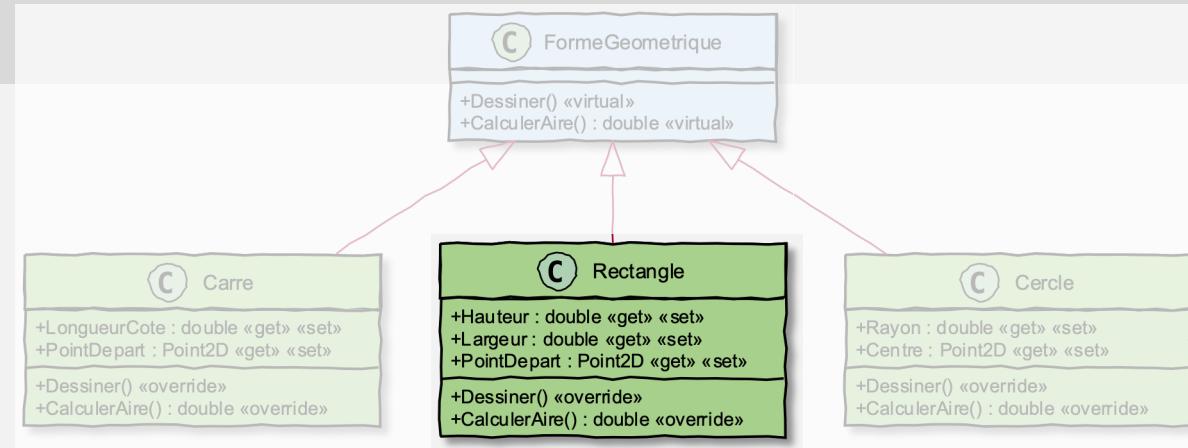


```
public class Carré : FormeGeometrique
{
    public Point2D PointDepart { get; set; }
    public double LongueurCote { get; set; }

    public override void Dessiner()
    {
        Console.Out.WriteLine($"Carre(PointDepart: {this.PointDepart}, LongueurCote: {this.LongueurCote})");
    }

    public override double CalculerAire()
    {
        return Math.Pow(this.LongueurCote, 2);
    }
}
```

# Exemple pratique : rajoutons un peu d'abstraction

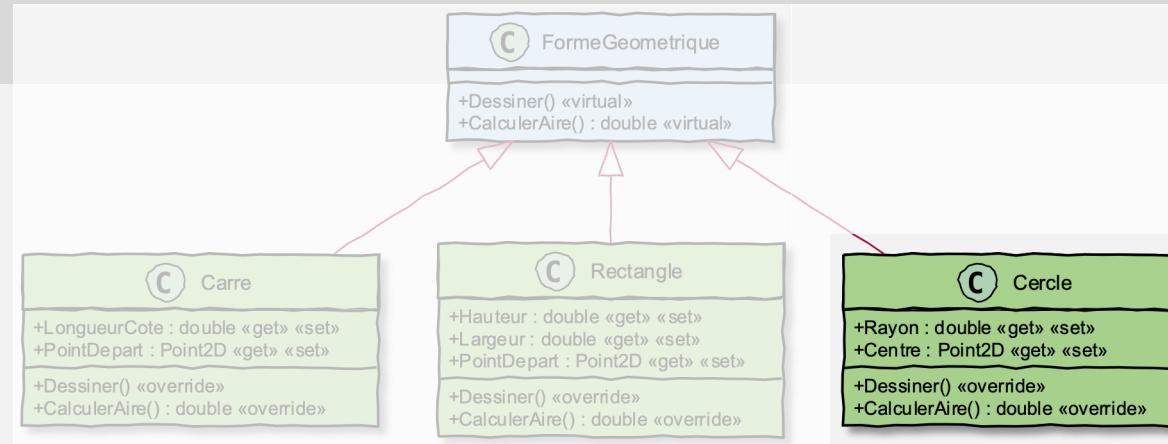


```
public class Rectangle : FormeGeometrique
{
    public Point2D PointDepart { get; set; }
    public double Hauteur { get; set; }
    public double Largeur { get; set; }

    public override void Dessiner()
    {
        Console.Out.WriteLine($"Rectangle(PointDepart: {this.PointDepart}, Hauteur: {this.Hauteur}, Largeur: {this.Largeur})");
    }

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

# Exemple pratique : rajoutons un peu d'abstraction



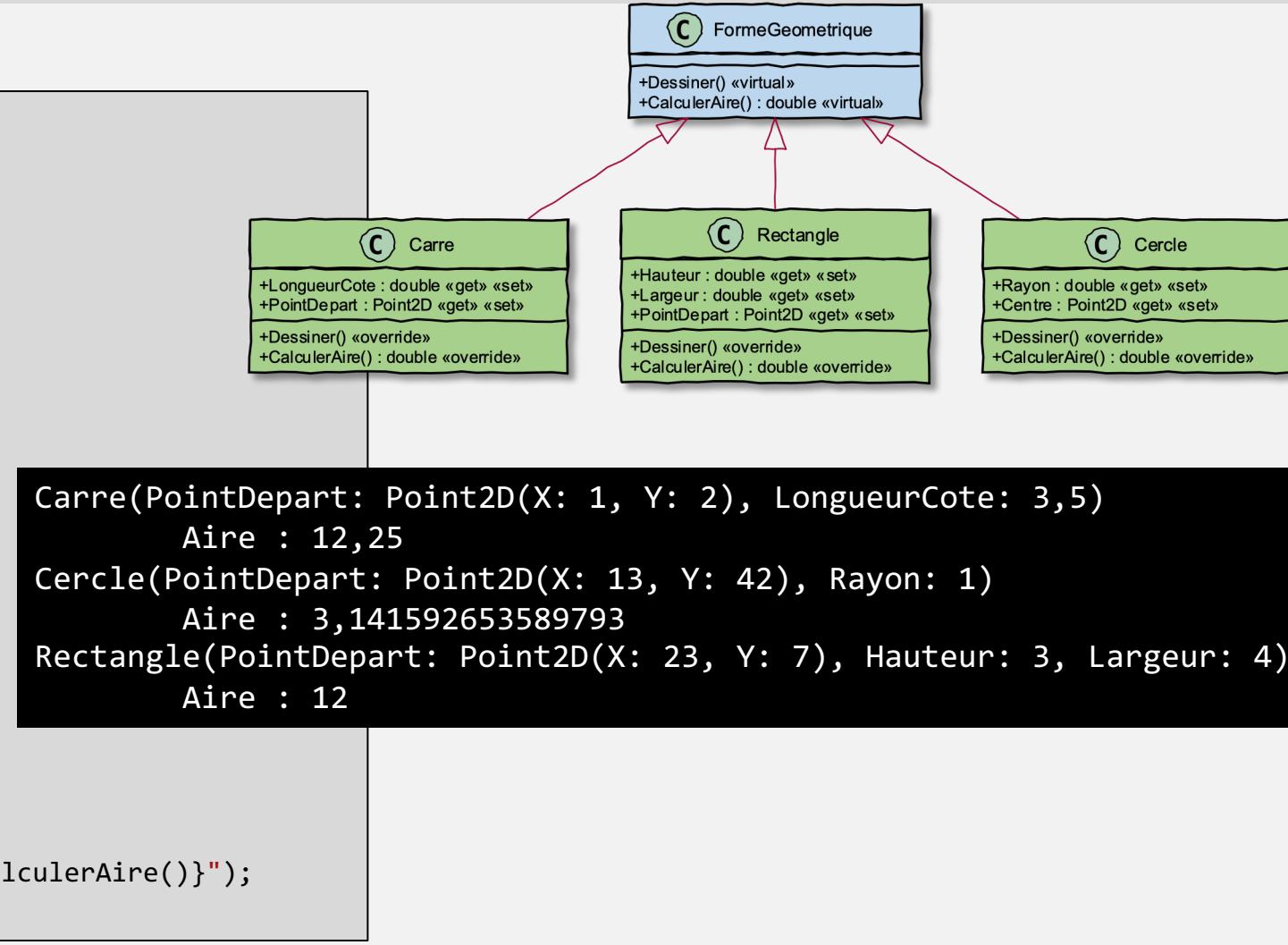
```
public class Cercle : FormeGeometrique
{
    public Point2D Centre { get; set; }
    public double Rayon { get; set; }

    public override void Dessiner()
    {
        Console.Out.WriteLine($"Cercle(PointDepart: {this.PointDepart}, Rayon: {this.Rayon})");
    }

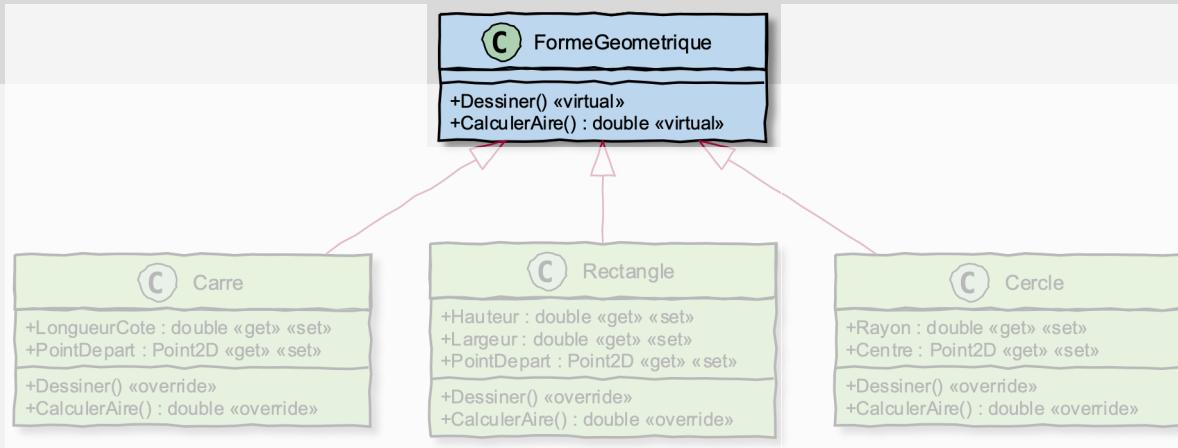
    public override double CalculerAire()
    {
        return Math.PI * Math.Pow(this.Rayon, 2);
    }
}
```

# Exemple pratique : rajoutons un peu d'abstraction

```
FormeGeometrique fg1 = new Carre() {
    PointDepart = new Point2D { X = 1, Y = 2 },
    LongueurCote = 3.5
};
FormeGeometrique fg2 = new Cercle() {
    Centre = new Point2D { X = 13, Y = 42 },
    Rayon = 1
};
FormeGeometrique fg3 = new Rectangle() {
    PointDepart = new Point2D() { X = 23, Y = 7 },
    Hauteur = 3,
    Largeur = 4
};
List<FormeGeometrique> fgs = new List<FormeGeometrique>()
{
    fg1,
    fg2,
    fg3
};
foreach (FormeGeometrique formeGeometrique in fgs)
{
    formeGeometrique.Dessiner();
    Console.Out.WriteLine($"\\tAire : {formeGeometrique.CalculerAire()}");
}
```



# Exemple pratique : rajoutons un peu d'abstraction



```
public class FormeGeometrique
{
    public virtual void Dessiner()
    {
        // ???
        throw new NotImplementedException();
    }

    public virtual double CalculerAire()
    {
        // ???
        throw new NotImplementedException();
    }
}
```

Quel code mettre dans ces méthodes ?

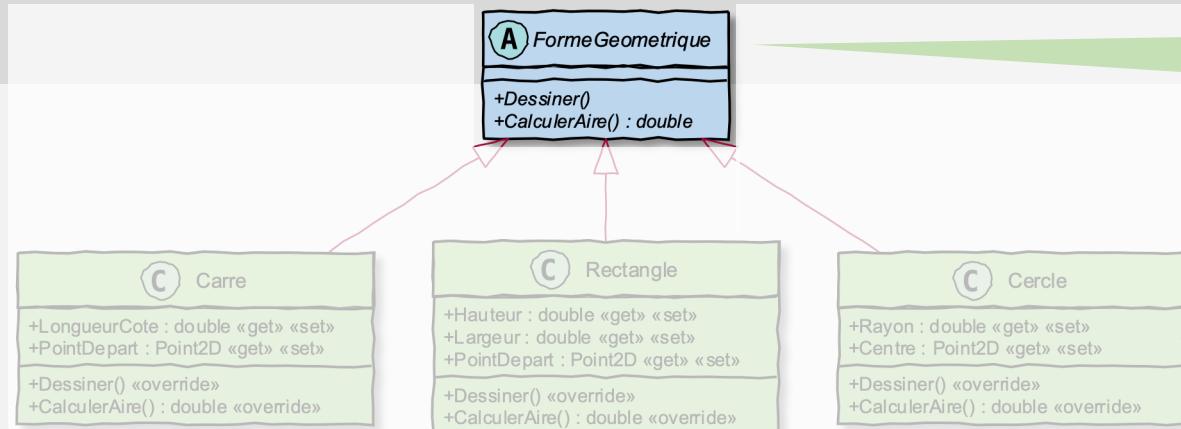
FormeGeometrique formeGeometrique = **new FormeGeometrique();**

Que représente un objet de ce type ?

# Méthodes abstraites et classes abstraites

- Une **méthode abstraite** est une méthode **déclarée** mais **non définie** (le code n'est pas donné)
- Pour qu'une méthode soit abstraite, il suffit d'utiliser le mot clef « **abstract** »
- Si au moins une méthode d'une classe **est abstraite**, la classe est **abstraite** et doit être marquée avec le mot clef « **abstract** »

# Exemple pratique : rajoutons un peu d'abstraction



Notez le « A » et  
L'italique

Dans notre exemple, le code des méthodes « Dessiner » et « CalculerAire » ne peut pas être écrit. Ces méthodes doivent donc être abstraites :

```
public abstract class FormeGeometrique
{
    public abstract void Dessiner();

    public abstract double CalculerAire();
}
```

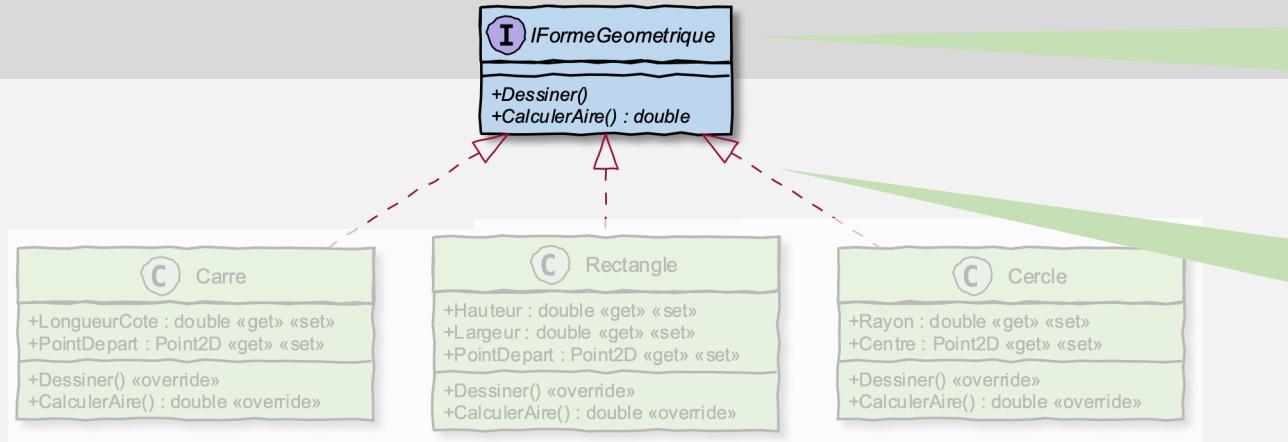
Maintenant  
interdit !

FormeGeometrique formeGeometrique = ~~new FormeGeometrique();~~

# Interface

- Quand tout est abstrait dans une classe, on peut la transformer en « interface »
- Pour cela, nous allons remplacer les mots clefs « abstract class » par « interface »
- Une classe peut implanter plusieurs interfaces
- En C#, le nom des interfaces est, par convention, préfixé par « I »
- Quand on peut, on va privilégier les interfaces

# Exemple pratique : rajoutons un peu d'abstraction



Notez le « I » et L'italique

Notez les pointillés

```
public interface IFormeGeometrique
{
    public abstract void Dessiner();

    public double CalculerAire();
}
```

« abstract » est optionnel ici

```
FormeGeometrique formeGeometrique = new IFormeGeometrique();
```

Toujours interdit !