



Redéfinition et surcharge

Objectifs

- Appréhender la surcharge de méthodes
- Appréhender la notion de redéfinition de méthodes

Surcharge de méthodes

- La surcharge de méthode (*overload*) consiste à déclarer et définir plusieurs méthodes qui ont toutes le même nom mais une signature différente
- La signature comprend :
 - Le nom de la méthode
 - Le nombre et les types des paramètres
- **! Le type de retour ne fait pas parti de la signature !**

Surcharge – Variation du nombre de paramètres

```
public void VariationNbParametres()
{
    Console.Out.WriteLine("VariationNbParametres()");
}

public void VariationNbParametres(int p_entier1)
{
    Console.Out.WriteLine("VariationNbParametres(int p_entier1)");
}

public void VariationNbParametres(int p_entier1, int p_entier2)
{
    Console.Out.WriteLine("VariationNbParametres(int p_entier1, int p_entier2)");
}
```

```
UneClasse uneClasse = new UneClasse();
uneClasse.VariationNbParametres();
uneClasse.VariationNbParametres(1);
uneClasse.VariationNbParametres(1, 2);
```

```
VariationNbParametres()
VariationNbParametres(int p_entier1)
VariationNbParametres(int p_entier1, int p_entier2)
```

Surcharge – Variation des types de paramètres

```
public void VariationTypeParametres2(long p_entier1)
{
    Console.Out.WriteLine("VariationTypeParametres(long p_entier1)");
}

public void VariationTypeParametres2(double p_reel1)
{
    Console.Out.WriteLine("VariationTypeParametres(double p_reel1)");
}
```

```
uneClasse.VariationTypeParametres2(1);
uneClasse.VariationTypeParametres2(1.0);
uneClasse.VariationTypeParametres2(1.0f);
uneClasse.VariationTypeParametres2(1L);
```

```
VariationTypeParametres2(long p_entier1)
VariationTypeParametres2(double p_reel1)
VariationTypeParametres2(double p_reel1)
VariationTypeParametres2(long p_entier1)
```

Surcharge – Variation des types de paramètres

```
public void VariationTypeParametres3(int p_entier1, double p_reel2)
{
    Console.Out.WriteLine("VariationTypeParametres3(int p_entier1, double p_reel2)");
}

public void VariationTypeParametres3(double p_reel1, int p_entier2)
{
    Console.Out.WriteLine("VariationTypeParametres3(double p_reel1, int p_entier2)");
}

public void VariationTypeParametres3(double p_reel1, double p_reel2)
{
    Console.Out.WriteLine("VariationTypeParametres3(double p_reel1, double p_reel2)");

uneClasse.VariationTypeParametres3(1, 1.0);
uneClasse.VariationTypeParametres3(1.0, 1.0);
uneClasse.VariationTypeParametres3(1, 1);
```

⊕ void UneClasse.VariationTypeParametres3(int p_entier1, double p_reel2) (+ 2 surcharges)

L'appel est ambigu entre les méthodes ou propriétés suivantes : 'UneClasse.VariationTypeParametres3(int, double)' et 'UneClasse.VariationTypeParametres3(double, int)'

uneClasse.VariationTypeParametres3(1, 1);

Surcharge – Valeurs par défaut

- Les méthodes avec des valeurs par défaut sont moins prioritaires que les méthodes avec les bons types et le bon nombre de paramètres

```
public void ParametresValeursDefaut(int p_entier1)
{
    Console.Out.WriteLine("ParametresValeursDefaut(int p_entier1)");
}

public void ParametresValeursDefaut(int p_entierA = 0, int p_entierB = 0)
{
    Console.Out.WriteLine("ParametresValeursDefaut(int p_entierA = 0, int p_entierB = 0)");
}
```

uneClasse.ParametresValeursDefaut();
uneClasse.ParametresValeursDefaut(1);
uneClasse.ParametresValeursDefaut(1, 2);
uneClasse.ParametresValeursDefaut(p_entierA: 2);
uneClasse.ParametresValeursDefaut(p_entierB: 2);

ParametresValeursDefaut(int p_entierA = 0, int p_entierB = 0)
ParametresValeursDefaut(int p_entier1)
ParametresValeursDefaut(int p_entierA = 0, int p_entierB = 0)
ParametresValeursDefaut(int p_entierA = 0, int p_entierB = 0)
ParametresValeursDefaut(int p_entierA = 0, int p_entierB = 0)

Surcharge et type de retour

```
0 références
public int VariationTypeRetour()
{
    return 0;
}
0 références
public double VariationTypeRetour()
{
    return 0.0;
}
```

Le type 'UneClasse' définit déjà un membre appelé 'VariationTypeRetour' avec les mêmes types de paramètre

Le type de retour n'est pas inclus dans la signature de la méthode pour la surcharge

Surcharge – Mixe (Exercice)

```
public void Mixe(int p_entier1, double p_reel2) // M1
{
    Console.Out.WriteLine("Mixe(int p_entier1, double p_reel2)");
}
public void Mixe(double p_reel1, int p_entier2) // M2
{
    Console.Out.WriteLine("Mixe(double p_reel1, int p_entier2)");
}
public void Mixe(double p_reel1 = 0.0, double p_reel2 = 0.0) // M3
{
    Console.Out.WriteLine("Mixe(double p_reel1 = 0.0, double p_reel2 = 0.0)");
}
```

```
uneClasse.Mixe();
uneClasse.Mixe(1, 1.0);
uneClasse.Mixe(1.0, 1);
uneClasse.Mixe(1.0, 1.0);
uneClasse.Mixe(1L, 1);
uneClasse.Mixe(1, 1);
```

Est-ce que l'appel est possible ?
Quelle méthode est appelée ?

Redéfinition de méthodes

- La redéfinition de méthodes est possible dans un contexte d'héritage (vu un peu plus loin dans le cours)
- **Pour le moment**, retenez que chacune de vos classes hérite des méthodes de la classe *Object*
- Votre classe aura donc automatiquement certaines méthodes

Redéfinition de méthodes

- Les méthodes que vous pouvez redéfinir sont :
 - `ToString()` : permet d'obtenir la représentation sous forme de chaîne de caractères d'un objet
 - `Equals(Object)` : permet de savoir si l'objet passé en paramètre est égal à l'objet sur lequel elle est appelée
 - `GetHashCode()` : permet d'obtenir la clé de hachage d'un objet
- Pour redéfinir une méthode en C#, on ajoute le mot clé « `override` » dans la déclaration : elle doit avoir exactement la même signature que la méthode à redéfinir

ToString() : string

```
public override string ToString()
{
    return "Voiture(couleur: " + this.Couleur.ToString()
        + ", nombrePortes: " + this.NombrePortes.ToString()
        + ", vitesse: " + this.Vitesse.ToString();
}
```

ToString() : string – Autres versions

```
public override string ToString()
{
    StringBuilder sb = new StringBuilder();

    sb.Append("Voiture(couleur: ");
    sb.Append(this.Couleur);
    sb.Append(", nombrePortes: ");
    sb.Append(this.NombrePortes);
    sb.Append(", vitesse: ");
    sb.Append(this.Vitesse);
    sb.Append(")");

    return sb.ToString();
}
```

```
public override string ToString()
{
    return $"Voiture(couleur: {this.Couleur}, nombrePortes: {this.NombrePortes}, vitesse: {this.Vitesse}";
}
```

Equals(Object) : bool

```
public override bool Equals(object p_obj)
{
    bool egaux = false;
    Voiture conversionVoiture = p_obj as Voiture;
    if (conversionVoiture != null)
    {
        egaux =
            this.Couleur.CompareTo(conversionVoiture.Couleur) == 0
            && this.NombrePortes == conversionVoiture.NombrePortes
            && this.Vitesse == conversionVoiture.Vitesse;
    }
    return egaux;
}
```

Opérateur qui renvoie l'objet dans le bon type ou null si p_obj n'est pas compatible avec Voiture

Redéfinition des opérateurs == et !=

```
public static bool operator ==(Voiture p_gauche, Voiture p_droit)
{
    return object.ReferenceEquals(p_gauche, p_droit)
        ||
        (
            !object.ReferenceEquals(p_gauche, null)
            && !object.ReferenceEquals(p_droit, null)
            && p_gauche.Equals(p_droit)
        );
}

public static bool operator !=(Voiture p_gauche, Voiture p_droit)
{
    return !(p_gauche == p_droit);
}
```

Ne pas faire p_gauche
== null sinon vous allez
appeler l'opérateur que
vous définissez

GetHashCode() : int

```
public override int GetHashCode()
{
    return HashCode.Combine(Couleur, NombrePortes, Vitesse);
}
```

Aller plus loin

- <https://docs.microsoft.com/en-us/dotnet/csharp/methods>
- <https://docs.microsoft.com/en-us/dotnet/api/system.object?view=netcore-3.1>