



Module 8

Hériter de Classes et Implémenter des Interfaces

Sommaire

- Utiliser l'héritage pour Définir de Nouveaux Types par Référence
- Définir et Implémenter des Interfaces
- Définir des Classes Abstraites

Leçon 1: Utiliser l'héritage pour Définir de Nouveaux Types par Référence

- Qu'est-ce que l'héritage?
- La Hiérarchie d'Héritage dans le Framework .NET
- Substituer et Masquer des Méthodes
- Appeler des Méthodes et des constructeurs de la classe de base
- Assigner et référencer des classes dans la Hiérarchie d'héritage
- Comprendre le Polymorphisme
- Définir des Classes et Méthodes Scellées

Qu'est-ce que l'héritage?

L'héritage vous permet de définir de nouveaux types basés sur les types existants :

- Par exemple, les classes de **Manager** et **ManualWorker** pourraient hériter de la classe **Employee**
- Les champs et méthodes de la classe **Employee** sont disponibles par **Manager** and **ManualWorker**
- **Manager** et **ManualWorker** peuvent leurs propres champs et comportements
- Définir les membres accessibles avec **protected** dans la classe de base

```
// Base class
class Employee
{
    protected string empNum;
    protected string empName;
    protected void DoWork()
    { ... }
}

// Inheriting classes
class Manager : Employee
{
    public void DoManagementWork()
    { ... }
}

class ManualWorker : Employee
{
    public void DoManualWork()
    { ... }
}
```



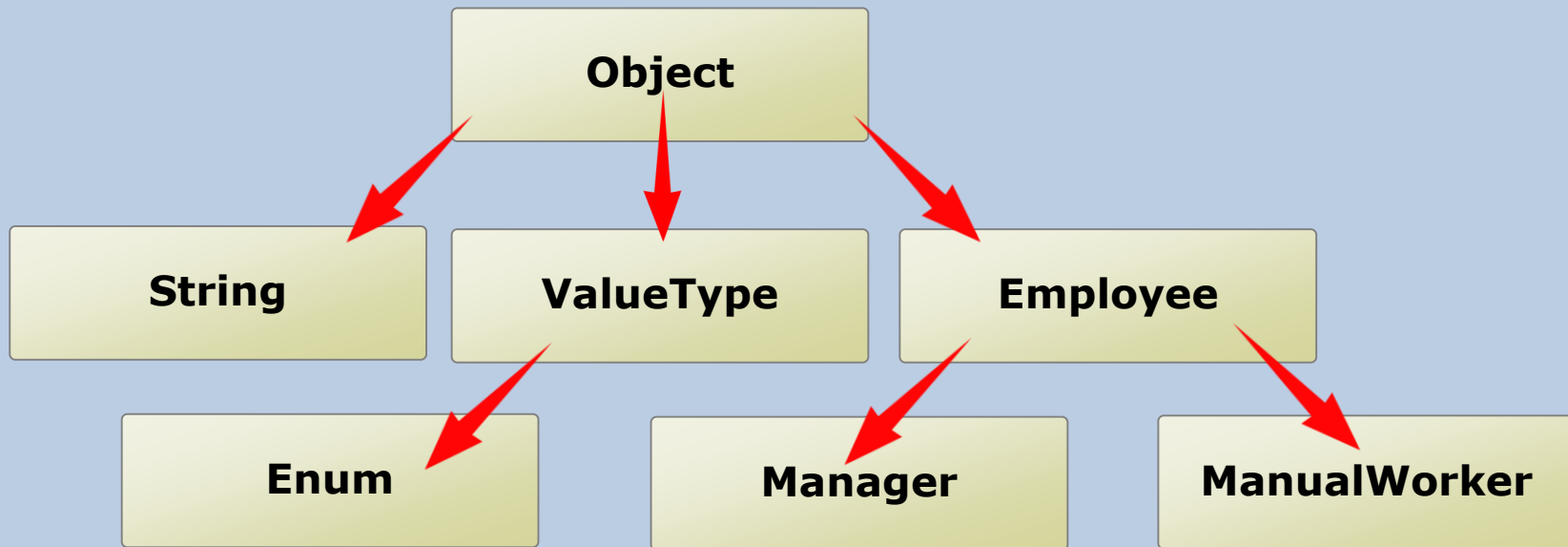
C# ne supporte que l'héritage unique



La Hiérarchie d'Héritage dans le Framework .NET

Tous les types héritent directement ou pas de la classe **System.Object**

- Pas besoin d'utiliser : **Object** dans la definition de la classe
- Les Structs et enums héritent de **ValueType**



Vous ne pouvez pas définir votre propre hiérarchie d'héritage en utilisant des structs et des enums



Substituer et Masquer des Méthodes



Substituer: Remplacer ou étendre fonctionnalité dans une classe de base avec un comportement sémantiquement équivalent (intentionnel)



```
class Employee
{
    protected virtual void
        DoWork()
    { ... }
}

class Manager : Employee
{
    protected override void
        DoWork()
    { ... }
}
```

```
class Employee
{
    protected void DoWork()
    { ... }
}

class Manager : Employee
{
    public new void DoWork()
    { ... }
}
```



Masquer: Remplacer une fonctionnalité dans une classe de base avec un nouveau comportement (possibilité d'erreur)



Appeler des Méthodes et des constructeurs de la classe de base

Utiliser le mot clé **base**

```
class Employee
{
    protected string empName;
    public Employee(string name)
    { this.empName = name; }
}

class Manager : Employee
{
    protected string empGrade;
    public Manager(string name,
                    string grade)
        : base(name)
    {
        this.empGrade = grade;
    }
}
```

```
class Employee
{
    protected virtual void
        DoWork()
    { ... }
}

class Manager : Employee
{
    protected override void
        DoWork()
    {
        ...
        base.DoWork();
    }
}
```

Les constructeurs appellent automatiquement le constructeur par défaut de la classe de base sauf si vous spécifiez autre chose



Assigner et référencer des classes dans la Hiérarchie d'héritage

La vérification des types C# vous empêche d'assigner une référence à type sur une variable d'un autre type ...



```
Manager myManager = new  
Manager(...);  
ManualWorker myWorker =  
    myManager;
```

... Masi vous pouvez assigner une référence à un type différent qui est plus élevé dans la hiérarchie d'héritage



```
Manager myManager = new  
Manager(...);  
Employee myEmployee =  
    myManager;
```

Utiliser les opérateurs **is** et **as** operators pour assigner de façon sécurisée une référence à un type qui est plus bas dans la hiérarchie d'héritage




```
Manager myManagerAgain =  
    myEmployee as Manager;
```




Comprendre le Polymorphisme

Dans une hiérarchie d'héritage, la même instruction peut appeler des implémentations différentes de la même méthode

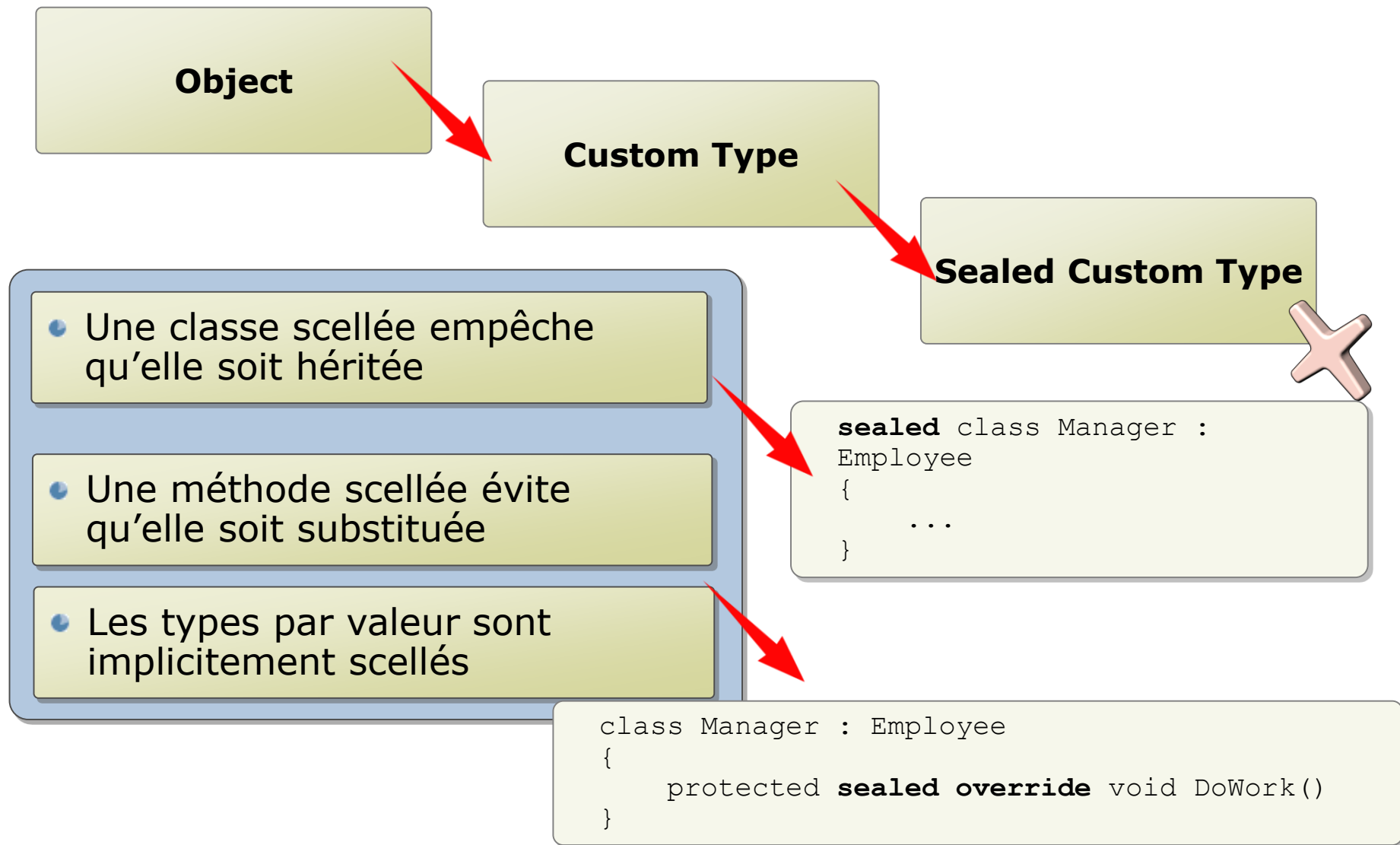


```
Employee myEmployee;  
Manager myManager =  
    new Manager(...);  
ManualWorker myWorker =  
    new ManualWorker(...);  
  
myEmployee = myManager;  
Console.WriteLine(  
    myEmployee.GetTypeName());  
  
myEmployee = myWorker;  
Console.WriteLine(  
    myEmployee.GetTypeName());
```



```
class Employee  
{  
    public virtual string  
    GetTypeName()  
    {  
        return "This is an  
            Employee";  
    }  
}  
  
class Manager : Employee  
{  
    public override string  
    GetTypeName()  
    {  
        return "This is a Manager";  
    }  
}  
  
class ManualWorker : Employee  
{  
    // Does not override GetTypeName  
}
```

Définir des Classes et Méthodes Scellées



Lesson 2: Définir et Implémenter des Interfaces

- Qu'est-ce qu'une Interface?
- Créer et implémenter une Interface
- Référencer un Objet via une Interface
- Implémenter une Interface implicitement et Explicitement

Qu'est-ce qu'une Interface?

Une Interface est:

- Un contrat qui spécifie quelles méthodes doivent être exposées par leur implementation dans une classe
- Indépendant de son implémentation

Comparable :
compareTo(...)

```
graph TD; Comparable[Comparable : compareTo(...)] --> String[String]; Comparable --> Int32[Int32]; Comparable --> Employee[Employee];
```

The diagram illustrates the Comparable interface and its implementations. At the top, a box labeled 'Comparable : compareTo(...)' has three red arrows pointing down to three separate boxes. The first box is for 'String', showing an example comparison 'BBB > AAA?' and 'Alphanumeric comparison'. The second box is for 'Int32', showing an example comparison '100 > 99 ?' and 'Numeric comparison'. The third box is for 'Employee', showing an example comparison 'VP > Worker ?' and 'Grade comparison'.

String

BBB > AAA?

**Alphanumeric
comparison**

Int32

100 > 99 ?

**Numeric
comparison**

Employee

VP > Worker ?

**Grade
comparison**

Créer et implémenter une Interface

Utiliser le mot clé **interface**

```
interface ICalculator
{
    double Add();
    double Subtract();
    double Multiply();
    double Divide();
}
```

Ne pas spécifier de
modificateur d'accès

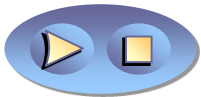
```
class Calculator : ICalculator
{
    public double Add() {}
    public double Subtract() {}
    public double Multiply() {}
    public double Divide() {}
}
```

Ajouter : suivi du nom de
l'interface à implémenter

S'assurer que les méthodes
sont accessibles de façon public



En C# les classes peuvent implémenter plusieurs interfaces



Référencer un Objet via une Interface

Vous pouvez définir une classe qui implémente une interface...

```
class Calculator : ICalculator
{
    ...
}
```

... puis utilisez cette interface pour faire référence à une instance de cette classe

```
Calculator myCalculator =
    new Calculator();
ICalculator iMyCalculator =
    myCalculator;
```

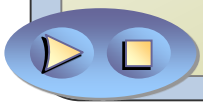
On peut également utiliser des interfaces comme paramètres

```
public int PerformAnalysis(ICalculator)
{
    ...
}
```

... et utiliser une logique pour tester si un objet implémente une interface ou pas

```
Calculator calc =
    iMyCalculator as Calculator;
bool isCalc =
    iMyCalculator is Calculator;
```

```
ICalculator iCalc =
    myCalculator as ICalculator;;
```



Implémenter une Interface implicitement et Explicitement

Implémentation Implicite:

- Convient pour des scénarios simples
- Peut être sujet à ambiguïté
- Signifie que toutes les méthodes sont visibles lorsque vous référencez un objet de classe

Explicit implementation:

- Convient pour des scénarios plus complexes
- Évite toute ambiguïté
- Est recommandé dans l'utilisation des interfaces
- Doit faire référence à des objets par le biais de l'interface appropriée pour appeler des méthodes

```
class Calculator : ICalculator
{
    double Add()
    {
    }

    double Subtract()
    {
    }

    ...
}
```

```
class Calculator : ICalculator
{
    double ICalculator.Add()
    {
    }

    double ICalculator.Subtract()
    {
    }

    ...
}
```



Leçon 3: Définir des Classes Abstraites

- Qu'est-ce qu'une classe abstraite?
- Qu'est-ce qu'une méthode abstraite?

Qu'est-ce qu'une classe abstraite?

Les classes Abstraites:

- Contiennent du code commun, réduisant ainsi la duplication de code
- Doivent être héritées
- Ne peuvent pas être instanciées
- Peuvent contenir des méthodes, des champs et autres membres
- Utilisent le mot clé **abstract**

```
abstract class SalariedEmployee : Employee, ISalaried
{
    void ISalaried.PaySalary()
    {
        Console.WriteLine("Pay salary: {0}", currentSalary);
        // Common code for paying salary.
    }
    int currentSalary;
}

class ManualWorker : SalariedEmployee, ISalaried
{
    ...
}

class Manager : SalariedEmployee, ISalaried
{
    ...
}
```

Qu'est-ce qu'une méthode abstraite?

Ajoutée à une classe abstraite

```
abstract class SalariedEmployee : Employee, ISalariedEmployee
{
    abstract void PayBonus();
    ...
}
```

Définie avec le modificateur **abstract**

Ne possède pas de corps



Les méthodes abstraites sont utiles lors du développement d'une classe abstraite qui implémente une interface ou repose sur une méthode où une implémentation par défaut n'est pas appropriée



Les classes qui héritent d'une classe avec une méthode abstraite doivent substituer cette méthode, sinon le code ne compilera pas



Atelier

- Exercise 1:
- Exercise 2:
- Exercise 3: