Héritage

L'héritage permet une réutilisation de classes déjà écrites, c'est-à-dire une réutilisation de la structure des données et des méthodes créées.
Lorsque plusieurs objets ont des caractéristiques et/ou des comportements communs, la création d'une classe de base (ou parent) permet de regrouper ce qui est semblable.

Prenons l'exemple d'une classes représentant des employés.

```
public class Employee
{
    private string _name;

    public Employee(string name)
    {
        Console.WriteLine("Employee constructor");
        _name = name;
    }

    public void Show()
    {
        Console.WriteLine("Employee Name : " + _name);
    }
}
```

Si maintenant nous désirons créer des employés responsables de département, nous pourrions créer la classe suivante :

```
public class Manager
{
    private string _name;
    private string _department;

    public Manager(string name, string department)
    {
        Console.WriteLine("Manager constructor");
        _name = name;
        _department = department;
    }

    public void Show()
    {
        Console.WriteLine("Manager Name : " + _name);
        Console.WriteLine("Department : " + _department);
    }
}
```

Nous pouvons facilement nous apercevoir que ces deux classes ont une caractéristique identique, la variable privée « name » et un comportement commun, la méthode « Show ». Cette manière de faire duplique du code, la bonne méthode pour ce cas-là, consiste à créer une classe de base et une classe dérivée de la classe précédemment créée.
Cela permet de réduire la complexité du code car nous obtenons moins de membres et méthodes à gérer.

Pour indiquer qu'une classe doit dériver d'une autre classe (ou hériter), il faut utiliser la syntaxe suivante :

```
public class Parent
{
    public Parent()
    {
    }

    public void Test()
    {
        Console.WriteLine("Test");
    }
}

public class Child : Parent
{
    public Child()
    {
    }
}
```

Sur le même principe que l'exemple précédent implémentons, une solution améliorée pour représenter les employés responsables des départements :

```
public class Employee
{
    // protected permet l'accès depuis une classe dérivée
    protected string name;

    public Employee(string name)
    {
        Console.WriteLine("Employee constructor");
        this.name = name;
    }

    public void Show()
    {
        Console.WriteLine("Employee Name : " + name);
    }
}

public class Manager : Employee
{
    private string _department;
    // :base(...) permet de transmettre le paramètre requis
    // par le constructeur de la classe de base
    public Manager(string name, string department) : base(name)
    {
        Console.WriteLine("Manager constructor");
        _department = department;
    }

    // le mot clef "new" permet de créer une méthode propre à la classe Manager
    public new void Show()
    {
        Console.WriteLine("Manager Name : " + name);
        Console.WriteLine("Department : " + _department);
    }
}

namespace ConsAppInheritance
{
    class Program
    {
        static void Main(string[] args)
        {
            Employee Emp1 = new Employee("Wilson");
            Emp1.Show();

            Manager Manag1 = new Manager("Peverelli", "Marketing");
            Manag1.Show();

            Console.ReadLine();
        }
    }
}
```

L'exécution de ce programme donnera l'affichage suivant :

```
Employee constructor
Employee Name : Wilson
Employee constructor
Manager constructor
Manager Name : Peverelli
Department : Marketing
```

Nous constatons que le constructeur de base y est appelé deux fois, une fois pour la création de l'employé et la seconde fois pour la création du manager. En effet, cela est fort utile car la structure de données héritée de la classe de base nécessite souvent d'être initialisée ou mise à jour.

Le principe d'héritage est extrêmement puissant car il permet de structurer très efficacement son code. Il faut toutefois faire très attention à l'utiliser de manière judicieuse car il n'est pas adapté à toutes les situations. Un premier test à faire lorsque l'on fait hériter une classe d'une autre et d'utiliser le verbe « être ». Notre exemple ci-dessus est judicieux parce que « un manager **est** un employé. »

Examinons un exemple qui illustre la puissance de ce mécanisme. Si je veux faire une liste de tous les employés d'une entreprise, je serais tenté de me servir en fait de deux listes :

```
using System;
using System.Collections.Generic;
public class Employee
{
    // …
}

public class Manager : Employee
{
    // …
}

List<Employee> employees = new List<Employee>();
List<Manager> managers = new List<Manager>();

Employee Emp1 = new Employee("Wilson");
employees.Add(Emp1);

Manager Manag1 = new Manager("Peverelli", "Marketing");
managers.Add(Manag1);

// Show company
foreach (Employee emp in employees)
{
    emp.Show();
}
foreach (Manager manager in managers)
{
    manager.Show();
}
Console.ReadLine();
```

Nous allons maintenant tirer profit du fait qu'un manager est un employé. Cela nous permet de mettre un manager dans une liste d'employés :

```
List<Employee> employees = new List<Employee>();

Employee Emp1 = new Employee("Wilson");
employees.Add(Emp1);

Manager Manag1 = new Manager("Peverelli", "Marketing");
employees.Add(Manag1);

// Show company
foreach (Employee emp in employees)
{
    emp.Show();
}
```

L'exécution de ce programme donnera l'affichage suivant :

```
Employee constructor
Employee constructor
Manager constructor
Manager Name : Wilson
Manager Name : Peverelli
```

Ceci nous embête un peu, parce que l'on ne voit pas l'information

Nous avons donc bien simplifié le code en passant de deux listes à une seule. Mais nous avons un problème : on ne voit pas à l'affichage que Peverelli est un manager et encore moins le département dont il est le responsable.

C'est tout à fait normal parce que nous avons une liste d'employés lorsque nous affichons les informations ce sont les informations d'employé qui sont affichées.

Selon le contexte dans lequel j'utilise cette liste, ce n'est peut-être pas un problème (par exemple si je ne veux que la liste des noms des employés). Mais si j'ai besoin de traiter différemment les managers des employés à l'intérieur de ma boucle for, je vais avoir besoin de connaître le type exact de l'objet et peut-être de le transformer avant usage :

```
List<Employee> employees = new List<Employee>();

Employee Emp1 = new Employee("Wilson");
employees.Add(Emp1);

Manager Manag1 = new Manager("Peverelli", "Marketing");
employees.Add(Manag1);

// Show company
foreach (Employee emp in employees)
{
    if (emp.GetType() == typeof(Manager)) // Identify the type of employee
    {
        Manager man = (Manager)emp; // Create a Manager object using type casting
        man.Show();
    }
    else
    {
        emp.Show();
    }
}
```