جامعة حماه

المعهد التقاني للحاسوب

قسم شبكات

السنة الثانية

برمجة3

مدرسة المقرر:

م. رغد العمادي

الوراثة

إن الوراثة من أهم مرتكزات البرمجة غرضية التوجه, ومن خلالها يمكن إنشاء صفوف جديدة بالإعتماد على صفوف سابقة مما يسهل عمليات إعادة الإستخدام (reusability)

- وتكون الوراثة بلغة C# بالشكل التالي:

```
Class Base
{
}
Class Derived : Base
{
}
```

ان الحرف " : " يستخدم لتحديد بأن هذا الصف يرث من صف آخر,

حيث أن الصف Derived يرث من الصف Base وبالتالي فان الصف الابن Derived يرث كافة الأعضاء الموجودة في الصف الأب Base باستثناء التابع الباني فلا يتم توريثه.

ملاحظة: على الرغم من أن الصف الابن يرث كافة الأعضاء في الصف الأب ولكنه لا يمكن أن يصل الى الأعضاء من ذات المحدد private

مثال

ليكن لدينا صف يعبر عن الأشخاص اسمه "Person" يحوي حقلين للإسم وتاريخ الميلاد وتابع يعيد نص يعبر عن الجمع بين الإسم مع تاريخ الميلاد.

```
Class Person     {
string  name, dateOfBirth;

 public string Name

  { get { return name; } set { name = value; } }

Public string DateOfBirth

      { get { return dateOfBirth; } set { dateOfBirth = value; } }

Public string Greeting()

      { return"Hello My Name is:" + Name + "\n BirthDate" + DateOfBirth; }

                }
```

والآن سننشئ صف يعبر عن الطالب "Student" الذي سيملك نفس الحقول الموجودة لدى الصف "Person" من خلال عملية الوراثة بالإضافة إلى حقل جديد مميز يمثل رقم التسجيل للطالب:

```
Class Student : Person

   {

Private int registrationNum;

 Public int RegistrationNum

 { get { return registrationNum; } set { registrationNum = value; } }
    }
```

وبالتالي يمكننا انشاء غرض من صف الطالب وهذا الغرض سيحوي جميع الخاصيات والطرائق الموجودة بالصف الأب والصف الإبن:

```
Static void Main(string[] args)

      { Student s1 = newStudent();

        s1.Name = "New Student";

        s1.DateOfBirth = "17-07-1990";

        s1.RegistrationNum = 37;

        Console.WriteLine(s1.Greeting());

      }
```

التابع الباني في الوراثة

عند إنشاء غرض من صف مشتق فإنه يتم استدعاء التابع الباني للأب أولاً ثم التابع الباني للإبن, وعند عدم تحديد تابع باني للصف يكون هناك تابع باني افتراضي لا يأخذ أي معامل ولا يعيد أي نتيجة ولكن في حال تم تحديد تابع باني يأخذ معاملات, وكان هذا الصف أب لصف آخر, فيجب علينا في الصف الإبن تمرير هذه المعاملات, وإلا فسيحدث خطأ برمجي مفاده أن التابع الأب لا يملك تابع باني لا يقبل أي معاملات.

مثلاً لو عدلنا الصف Person ليملك التابع الباني الذي يأخذ معاملين كما يلي :

```
Class Person     {
string name, dateOfBirth;


public Person(string name,string dateOfBirth)
        {
this.name = name;
this.dateOfBirth = dateOfBirth;          }


Public string Name
        { get { return name; } set { name = value; } }

Public string DateOfBirth
        { get { return dateOfBirth; } set { dateOfBirth = value; } }

Public string Greeting()
        {
return"Hello My Name is:" + Name + "\n BirthDate" + DateOfBirth; }
                        }
```

فيجب علينا أن ننشئ تابع باني في الصف الإبن ليستدعي الباني في الأب ويمرر له المتحولات اللازمة. ويكون ذلك بالكلمة المفتاحية "base" كمايلي:

```
Class Student : Person
   {
Private int registrationNum;

public  Student(string name, string  dateOfBirth,  int  registrationNum) : base(name , dateOfBirth)
       {
this.registrationNum = registrationNum;        }


publicint RegistrationNum
       { get { return registrationNum; } set { registrationNum = value; } }
   }
```

الكلمة المفتاحية "base" تمثل استدعاء للتابع الباني للأب, وليست تعريف جديد للتابع, ولذلك نحن لا نعرف معاملات جديدة وإنما نمرر المتحولات اللازمة بحسب التعريف الخاص للتابع الباني في الصف الأب.

الكلمة المفتاحية Sealed

عند بناء صف وتمييزه بالكلمة المفتاحية sealed فهذا يعني أن هذا الصف "عقيم" ولا يمكن أن يرث منه أي صف آخر.

مثلاً لو كان لدي صفين أحدهما يرث من الآخر كما يلي:

```
classA
   {
}
classB : A
   {
   }
```

لو عدلنا الصف الأب A بالكلمة المفتاحية Sealed فسيحدث خطأ في عملية الوراثة :

```
sealedclassA
   {
    }
classB : A
   {

   }
```

'B': cannot derive from sealed type 'A'

Polymorphism تعددية الأشكال

هي قدرة الصفوف على تنفيذ مهام مختلفة للطرائق التي تملك نفس الإسم ويتم استدعاؤها بذلك الإسم

استخدام نمط الصف الأب لإنشاء غرض من نمط الصف الإبن

مثال :

```
Class A    {
Public void MethodA()
      {
      }
   }
Class B : A
   {
Public void MethodB()
      {
      }    }
```

وفي لغة #C يمكننا استخدام نمط الصف الأب لإنشاء غرض من نمط الصف الإبن:

```
A a = newB();
```

لاحظ انشاء غرض من الصف الابن ولكن من نوع الصف الأب وبشكل أوضح نقول أن مؤشر من النمط الأب يؤشر على غرض من النمط الابن وهذا ممكن في #C وهنا يجب أن نعامل هذا الغرض كغرض من الصف الأب وليس الابن أي أنه يمكننا أن نصل الى أعضاء الصف الأب مثلاً:

```
a.MethodA();
```

ولا يمكن أن نصل الى أعضاء الصف الابن أي:

```
a.MethodB(); // Error
```

استخدام الطرائق التي لها نفس الإسم في الصف الأب والصف الإبن لو كان لدينا طريقتين لهما نفس الاسم (نفس الترويسة) في الصف الأب والصف الابن وقمنا بانشاء غرض من الصف الابن باستخدام مؤشر من نوع الصف الأب , وقمنا باستدعاء هذا الطريقة (الموجودة في كلا الصفين) فانه سيتم تنفيذ الطريقة الموجودة في الصف الأب

مثال :

```
Class Shape     {
Public void Draw()
        {
Console.WriteLine("This is class Shape");
        }
    }
Class Circle : Shape
    {
Public void Draw()
        {
Console.WriteLine("This is class Circle");
        }
    }
```

ولو قمنا بتعريف غرض كما يلي :

```
staticvoid Main(string[] args)
        {           Shape s1 = newCircle();
                    s1.Draw();
        }
```

سيكون الناتج كما يلي:

```
This is class Shape
```

الكلمات المفتاحية Virtual و Override

لو أردنا تجاهل الطريقة الموجود في الصف الأب وتنفيذ الطريقة الموجودة في الصف الابن يتوجب علينا وضع الكلمة المفتاحية virtual عند الطريقة الموجودة في الصف الأب وكلمة Override عند الطريقة الموجودة في الصف الابن التي سيتم تنفيذها فعلياً وهذا يعني بأن الطريقة الموجودة Virtual بالصف الأب إفتراضية وفي حال وجد مقابل لها override في الإبن فسيتم تنفيذ الطريقة في الصف الإبن:

```
Class Shape     {
Virtual public void Draw()
        {
Console.WriteLine("This is class Shape");
        }
    }
Class Circle : Shape
    {
Override public void Draw()
        {
Console.WriteLine("This is class Circle");        }     }
```

وعند تنفيذ التعليمات التالية:

```
Static void Main(string[] args)
        {           Shape s1 = newCircle();
                    s1.Draw();
        }
```

سيصبح الناتج:

```
This is class Circle
```

ملاحظات:

يمكن استخدام أكثر من override لطريقة virtual واحدة وذلك اذا كان هناك أكثر من صف يرثوا من صف ما ويكون نتيجة التنفيذ حسب نوع الغرض المنشأ وليس حسب نوع المؤشر .

يمكن استخدام الكلمة المفتاحية New عند الطريقة في الصف الابن التي لها نفس اسم الطريقة في الصف الأب للدلالة على أنه لا يوجد تعددية أشكال .

مفهوم الواجهات (Interfaces)

- من المعروف أنه في لغة #c لا يمكن تطبيق مفهوم الوراثة المتعددة (أي كل صف ابن يجب يرث من صف أب واحد فقط), و لحل هذه المشكلة تم إيجاد الواجهات (Interfaces).

- الواجهة تشبه الصف و لكنها تحوي **فقط التصريح عن الخاصيات و الطرائق** و ليس تعريف كامل لها.

- يتم تعريف الواجهة في #c كما يلي:

```
namespace ConsoleApplication4
{
    interface myInterface
    {


    }
    class myClass : myInterface
    {


    }
```

- الواجهة لا تحوي أي تعليمة برمجية, و إنما فقط التصريح عن التوابع (أو الخاصيات).

- لا يمكن التصريح عن الحقول (المتحولات) ضمن الواجهة.

- لا يمكن تحديد معرفات وصول (access modifiers) ضمن الواجهة و إنما يتم تحديد معرف الوصول عند تعريف التابع أو الخاصية في الصف الابن, و يجب أن يكون public .

- لا يمكن إنشاء أغراض من الواجهات.

- يمكن للصف إن يرث من أكثر من واجهة.

- يمكن للواجهة أن ترث من واجهة أخرى, و في هذه الحالة الصف الابن يجب أن يطبق كل الطرائق و الخاصيات الموجودة في كلا الواجهتين.

- لا يمكن للواجهة أن ترث من صف..

- عندما يرث الصف من صف آخر و عدة واجهات, يجب دائماً أن نكتب الصف قبل الواجهات في عملية الوراثة.

مثال علمي :

- ليكن لدينا واجهة لتمثيل بيانات وجبة بيتزا باسم Ipizza تتضمن طريقة لطباعة معلومات الطلب.
- و صف لتمثيل بيانات وجبة سريعة باسم FastFood يتضمن اسم الوجبة Food_name و عدد السعرات الحرارية فيها calories.
- و صف لتمثيل بيانات نوع من أنواع البيتزا باسم pepperoniPizza يتضمن نفس بيانات الصف و الواجهة السابقين.
- قم بإنشاء البنية البرمجية السابقة محدداً العلاقة بينها إن وجدت, ثم قم بإنشاء غرض من الصف الأخير في التابع الرئيسي main و قم بإسناد قيم للمتحولات و استدعاء التوابع .

الحل :

```
interface Ipizza
{
    int Price {  get; }
    void order();
}
class fastfood
{
    string foodname;
    int calories;
    public string Foodname {
        set { foodname = value; }
        get { return foodname; } }
    public int Calories {
        set { calories = value; }
        get { return calories; } }
}
class peppronipizza :fastfood, Ipizza
{
    int price;
    public int Price { set { price = value; } get { return price; } }
    public void order()
    {
        Console.WriteLine("pepproni pizza order");
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        peppronipizza pizza = new peppronipizza();
        pizza.Calories = 120;
        pizza.Foodname = "pepproni";
        pizza.order();
    }
}
```