جامعة البعث
الكلية التطبيقية
قسم تقنيات الحاسوب

برمجة متقدمة
السنة الثانية
عملي
محاضرة سادسة

م. زكي نقولا م. بتول الليوس م. سارة معروف

الخصائص Properties

للخصائص ولحقول البيانات المعنى المنطقيّ نفسه في البرمجة كائنيّة التوجّه، إلّا أنّ سي شارب تميّز بينهما من الناحية العمليّة. للخصائص في سي شارب مرونة أكبر، حيث من الممكن تنفيذ عبارات برمجيّة عند إسناد قيمة إلى خاصيّة أو حتى عند القراءة منها. كما من الممكن أن نجعل إحدى الخواص قابلة للقراءة فقط أو حتى قابلة للكتابة فقط (ولو أنّه أمر نادر الحدوث). لفهم هذه المزايا بشكل جيّد سأستعير البرنامج من الدرس السابق، وأجري عليه بعض التعديلات لإدخال مفهوم الخصائص

```
1   using System;
2
3   namespace Lesson07_01
4   {
5
6       class Employee
7       {
8           private string firstName;
9           private string lastName;
10          private double salary;
11
12          public string FirstName
13          {
14              get
15              {
16                  return this.firstName;
17              }
18              set
19              {
20                  this.firstName = value;
21              }
22          }
23
24          public string LastName
25          {
26              get
17              {
28                  return this.lastName;
29              }
30              set
31              {
32                  this.lastName = value;
33              }
34          }
35
36
37          public double Salary
38          {
39              get
40              {
41                  return this.salary;
42              }
```

```
            set
            {
                this.salary = value;
            }
        }

        public string DisplayInfo()
        {
            string result = string.Format("{0} {1} - Salary: {2:N0}",
                this.FirstName, this.LastName, this.Salary);

            return result;
        }

        public Employee(string firstName, string lastName, double salary)
        {
            this.FirstName = firstName;
            this.LastName = lastName;
            this.Salary = salary;
        }

        public Employee()
        {

        }
    }


    class Program
    {
        static void Main(string[] args)
        {
            Employee employee1, employee2;

            employee1 = new Employee("Mohammad", "Mansoor", 1000);
            employee2 = new Employee("Saleh", "Mahmoud", 2500);

            Console.WriteLine("First Employee: {0}", employee1.DisplayInfo());
            Console.WriteLine("Second Employee: {0}", employee2.DisplayInfo());
        }
    }
}
```

عند تنفيذ البرنامج سنحصل على نفس الخرج الذي حصلنا عليه في البرنامج . لقد أجرينا في الحقيقة بعض التعديلات التي تبدو للوهلة الأولى أنّها ليست ذات مغزى. لقد استبدلنا محدّد الوصول للحقول في الأسطر من 8 حتى 10 ليصبح `private` بدلًا من `public` (كما كان الوضع في البرنامج السابق). يفيد مُحدّد الوصول هذا في جعل الحقل خاصًّا بالصنف ولا يمكن الوصول إليه من خارج الكائن المُنشَأ من هذا الصنف، وبالتالي لا يمكن لأحد أن يُعدّل عليه إلّا التوابع الموجودة ضمن نفس الصنف حصرًا. الأمر الآخر أنّنا قد جعلنا أسماء الحقول تبدأ بحرف طباعي صغير وذلك لتمييزها عن الخصائص التي ستأتي بعدها والتي تحمل نفس الاسم ولكن بحرف طباعي كبير.

يبدأ التصريح عن الخاصيّة `FirstName` في السطر 12 ويمتدّ حتى السطر 22. للخصائص في سي شارب فوائد عظيمة سنختبرها بالتدريج في هذا الدرس وفي الدروس اللّاحقة. لاحظ وجود النوع `string` قبل اسم الخاصيّة في السطر 12، يُشير ذلك إلى أنّ هذه الخاصيّة تقبل وتعطي قيمًا نصيّة فحسب. في الحقيقة يمكن استبدال `string` بأيّ نوع نحتاجه. من الواضح أنّ التصريح عن الخاصيّة `FirstName` يتألّف من قسمين: قسم القراءة `get` (بين السطرين 14 و 17) وقسم الإسناد `set` (بين السطرين 18 و 21). في الواقع لا تتعدّى الخاصيّة كونها وسيلة للوصول إلى الحقول الخاصّة private fields الموجودة ضمن الكائن سواءً بالقراءة أو الإسناد، ولكن مع إمكانيّة معالجة القيم سواءً قبل إسنادها إلى هذه الحقول أو بعد القراءة منها. عندما نحاول إسناد قيمة إلى الخاصيّة `FirstName` ستُنفّذ العبارات البرمجيّة الموجودة في قسم set، وهي عبارة برمجيّة واحدة فقط في مثالنا هذا:

```
this.firstName = value;
```

الكلمة `value` هي كلمة محجوزة تحتوي على القيمة المُسندة إلى الخاصيّة `FirstName`. العبارة السابقة واضحة للغاية فهي تعمل على إسناد القيمة المخزّنة ضمن `value` إلى الحقل `firstName`. لاحظ كيف يمكننا الوصول إلى هذا الحقل من الكلمة `this`، كما ويمكننا إغفالها. نستطيع الوصول إلى الحقل `firstName` رغم أنّه ذو محدّد وصول `private` لأنّنا نصل إليه من تابع يقع في نفس الصنف.

أمّا عندما نحاول قراءة الخاصيّة `FirstName` فسيتمّ تنفيذ العبارات البرمجيّة الموجودة ضمن القسم `get`. في هذا المثال يحتوي القسم `get` على عبارة برمجيّة واحدة وهي:

```
return this.firstName;
```

حيث تعمل على إرجاع القيمة المخزّنة ضمن الحقل `firstName` إلى الشيفرة التي طلبت قراءة الخاصيّة `FirstName`. يطبّق نفس الأمر تمامًا على الخاصيّتين `LastName` و`Salary`

الخصائص المطبقة تلقائيا

يبدو البرنامج السابق طويلًا بلا مبرّر، فنحن لم نقم بأيّ عمل ضمن الخصائص سوى الإسناد أو القراءة. إذا كان الأمر كذلك في برامجك الحقيقيّة فيمكنك الاستغناء عن هذا الشكل من الخصائص واللجوء إلى شكل أكثر حداثةً وعصريّة، والذي يتمثّل بالخصائص المطبّقة تلقائيًّا auto implemented properties. انظر الشكل العام لها فيما يتعلّق بالخاصيّة FirstName:

```
public string FirstName
{
  get;
  set;
}
```

لم تعد العبارات البرمجيّة في قسميّ get و set موجودة. ينحصر دور هذه الخاصيّة في شكلها الحالي في تخزين القيم ضمن الخاصيّة FirstName والقراءة منها فقط. ولكن يأتي السؤال هنا، أين ستخزّن الخاصيّة FirstName قيمها، فأنا لا أرى حقلًا للتخزين!

يعمل المترجم في هذه الحالة على إنشاء حقل خاص غير مُشاهد في شيفرة MSIL وظيفته الاحتفاظ بقيمة الخاصيّة FirstName. وهنا قد يجول بخاطرك سؤال آخر: لماذا كلّ هذا التعقيد، لماذا لا نستخدم الحقول كما كنّا نفعل في الدرس السابق وحسب؟

الإجابة بسيطة على هذا التساؤل المشروع. فنحن نستخدم الخصائص بهذا الشكل لغايات تصميميّة فحسب. فكلّما كنت بحاجة لأن تُضيف خاصيّة لأحد الأصناف يمكن الوصول إليها من خارجه فافعل ذلك عن طريق الخصائص (وليس الحقول) ولن تندم، وإن بدا ذلك يتطلّب المزيد من العمل.

سنطبّق هذه الخصائص الفريدة على برنامجنا المعدّل بحيث تستغني تمامًا عن الحقول firstName و lastName و salary.

```
using System;

namespace Lesson07_02
{

    class Employee
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public double Salary { get; set; }

        public string DisplayInfo()
        {
            string result = string.Format("{0} {1} - Salary: {2:N0}",
                this.FirstName, this.LastName, this.Salary);

            return result;
        }

        public Employee(string firstName, string lastName, double salary)
        {
            this.FirstName = firstName;
            this.LastName = lastName;
            this.Salary = salary;
        }

        public Employee()
        {

        }
    }


    class Program
    {
        static void Main(string[] args)
        {
            Employee employee1, employee2;

            employee1 = new Employee("Mohammad", "Mansoor", 1000);
            employee2 = new Employee("Saleh", "Mahmoud", 2500);

             Console.WriteLine("First Employee: {0}", employee1.DisplayInfo());
             Console.WriteLine("Second Employee: {0}", employee2.DisplayInfo());
        }
    }
}
```

أصبح هذا البرنامج الآن يُشبه البرنامج من الدرس السابق إلى حدّ كبير، باستثناء أنّنا نستخدم هنا الخصائص بدلًا من الحقول. لاحظ فقط أنّه يمكننا كتابة التصريح عن أيّ خاصيّة على نفس السطر مثل الأسطر 8 و 9 و 10

الخصائص ذات إمكانية القراءة فقط

هل تذكُر التمرين الضريبة من الدرس السابق

كان يطلب ذلك التمرين إضافة تابع جديد اسمه `GetSalaryAfterTax` للحصول على قيمة الراتب بعد خصم الضريبة. واتفقنا وقتها أن تكون هذه الضريبة 2%. سنضيف خاصيّةً لتقوم بهذه المهمّة بدلًا من هذا التابع، ولكنّنا سنجعلها للقراءة فقط read only. أي لا يمكن إسناد أي قيم لها.

ستكون الخاصيّة `SalaryAfterTax` الجديدة على الشكل التالي

```
public double SalaryAfterTax
{
  get
  {
    return 0.98 * this.Salary;
  }
}
```

من الواضح أنّه قد أزلنا القسم `set` المسؤول عن الإسناد من تصريح الخاصيّة `SalaryAfterTax` وبذلك تتحوّل للقراءة فقط. يحتوي القسم get على عمليّة حسابيّة بسيطة تطبّق عملية حسم الضريبة على الراتب Salary.

سنجري تعديلًا طفيفًا على التابع `DisplayInfo` لكي يُرفق قيمة الراتب بعد حسم الضريبة ضمن النصّ المنسّق الذي يرجعه. سنحصل في النتيجة على البرنامج المعدّل:

```
using System;

namespace Lesson07_03
{

    class Employee
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public double Salary { get; set; }
        public double SalaryAfterTax
        {
            get
            {
                return 0.98 * this.Salary;
            }
        }

        public string DisplayInfo()
        {
            string result = string.Format("{0} {1} \n Salary: {2:N0} \n Salary after tax: {3:N0}",
                this.FirstName, this.LastName, this.Salary, this.SalaryAfterTax);

            return result;
        }

        public Employee(string firstName, string lastName, double salary)
        {
            this.FirstName = firstName;
            this.LastName = lastName;
            this.Salary = salary;
        }

        public Employee()
        {

        }
    }


    class Program
     {
         static void Main(string[] args)
         {
             Employee employee1, employee2;

             employee1 = new Employee("Mohammad", "Mansoor", 1000);
             employee2 = new Employee("Saleh", "Mahmoud", 2500);

             Console.WriteLine("First Employee: {0}", employee1.DisplayInfo());
             Console.WriteLine("Second Employee: {0}", employee2.DisplayInfo());
         }
     }
}
```

أضفت الخاصيّة SalaryAfterTax(الأسطر من 11 حتى 17). وأجريت تعديلًا طفيفًا ضمن التابع DisplayInfoفي السطر 21 حيث أضفت المحرف \n والذي يُستخدم ضمن النص للإشارة إلى وجوب الانتقال إلى سطر جديد لأغراض تنسيقية فقط، كما أضفت مكانًا في النصّ التنسيقيّ {3:N0} لإدراج قيمة الراتب بعد خصم الضريبة this.SalaryAfterTax، وهذا كلّ ما في الأمر.

إذا حاولت في هذا البرنامج أن تُسند أيّ قيمة إلى الخاصيّة SalaryAfterTax ستحصل على خطأ يفيد أنّها للقراءة فقط read only.

الأعضاء الساكنة Static Members

هي أعضاء يمكن استدعاؤها مباشرةً من الصنف الذي صُرّحت ضمنه، وليس من كائن مُنشَأ من هذا الصنف. يمكن أن نجعل أيّ عضو ساكن وذلك بوسمه بالكلمة المحجوزة static. يوضّح البرنامج استخدام التوابع الساكنة. لاحظ وجود الكلمة المحجوزة staticبعد محدّد الوصول :public

```
using System;

namespace Lesson07_04
{
    class Calculator
    {
        public static double Addition(double x, double y)
        {
            return x + y;
        }

        public static double Minus(double x, double y)
        {
            return x - y;
        }

        public static double Division(double x, double y)
        {
            if (y == 0)
            {
                return double.NaN;
            }
            else
            {
                return x / y;
            }
        }

        public static double Multiplication(double x, double y)
        {
            return x * y;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            double x = 5;
            double y = 9;
```

```
42              double addition = Calculator.Addition(x, y);
43              double minus = Calculator.Minus(x, y);
44              double multiplication = Calculator.Multiplication(x, y);
45              double division = Calculator.Division(x, y);
46
47              Console.WriteLine("{0} + {1} = {2}", x, y, addition);
48              Console.WriteLine("{0} - {1} = {2}", x, y, minus);
49              Console.WriteLine("{0} * {1} = {2}", x, y, multiplication);
50              Console.WriteLine("{0} / {1} = {2}", x, y, division);
51          }
52      }
53  }
```

نفّذ البرنامج السابق لتحصل على الخرج التالي:

```
5 + 9 = 14
5 - 9 = -4
5 * 9 = 45
5 / 9 = 0.555555555555556
```

أنشأنا الصنف `Calculator` الذي يحتوي على التوابع الساكنة `Addition` و`Minus` و`Multiplication` و`Division`. إذا انتقلنا إلى التابع `Main` الذي هو بالمناسبة تابع ساكن بسبب وجود الكلمة `static`) انظر إلى السطر 42 كيف استدعينا التابع `Addition` من الصنف `Calculator` مباشرةً بدون إنشاء أي كائن من هذا الصنف. سنكرّر نفس العمليّة من أجل التوابع `Minus` و `Multiplication` و `Division`.

أمرٌ أخير. انظر إلى محتوى التابع الساكن `Division`، ستجد أنّنا نختبر قيمة `y` فيما إذا كانت تساوي الصفر أم لا. فإذا كانت قيمة `y` تساوي الصفر فإنّه لا يجوز القسمة على صفر. لذلك فنرجع `double.NaN` وهو عبارة عن ثابت يُعبّر عن عدم وجود قيمة عدديّة. وهذا أمر طبيعي لأنّ القسمة على صفر لن تعطينا عدد. إذا استبدلت قيمة `y` في السطر 40 من البرنامج السابق بالقيمة 0 سنحصل على الخرج التالي:

```
5 + 0 = 5
5 - 0 = 5
5 * 0 = 0
5 / 0 = NaN
```

لاحظ السطر الأخير من البرنامج كيف يبدو منطقيًّا تمامًا.

يمكننا تعميم نفس المفهوم السابق بالنسبة للخصائص والحقول ضمن الصنف بجعلها ساكنة وذلك بإضافة الكلمة المحجوزة `static` بعد محدّد الوصول مباشرةً.

بقي أن نشير إلى أنّه من غير الممكن استخدام الكلمة المحجوزة `this` ضمن أي عضو ساكن والسبب كما أعتقد واضح. يُشير `this` إلى الكائن الذي يحدث من ضمنه الاستدعاء. ولكن في الأعضاء الساكنة فإنّنا نجري الاستدعاءات للتوابع أو الخصائص من الصنف المصرّحة ضمنه مباشرةً، لذلك فاستخدام `this` لن يكون له أيّ معنى.

الوراثة Inheritance

سبق وأن قدّمنا للوراثة، واتفقنا على أنّها من أهمّ المفاهيم التي يمكن أن تدعمها لغات البرمجة كائنيّة التوجّه. في الحقيقة يرث أيّ صنف موجود في إطار عمل دوت نت أو أيّ صنف تنشئه بنفسك بشكل مباشر أو غير مباشر من الصنف Object حتى ولو لم نخبر مترجم سي شارب بذلك، حيث سيعمل المترجم على الوراثة منه بشكل ضمنيّ. هذا الصنف ذو دلالة عامّة، وهو غير مفيد كثيرًا كاستخدام بحدّ ذاته. يحتوي الصنف Object على عدد قليل من التوابع كأعضاء ضمنه مثل `Equals` و `GetHashCode` و `GetType` و `ToString`. التابع الأكثر استخدامًا هو التابع `ToString`، وهو يُستخدَم عادةً للحصول على التمثيل النصيّ لأيّ كائن.

لكي نفهم الوراثة بشكل عمليّ لا بدّ لنا من مثال تمهيديّ. سننشئ لهذا الغرض صنف أب سأسمّيه `Father`. سيكون هذا الصنف هو الأساس الذي نرث منه. لهذا الصنف الشكل التالي:

```
class Father
{

  public Father()
  {
    Console.WriteLine("Father: In Constructor");
  }

  public void MyMethod()
  {
    Console.WriteLine("Father: In MyMethod");
  }
}
```

يحتوي هذا الصنف على التابع MyMethod الذي يحوي الكلمة void قبل اسم التابع مباشرةً. تعني هذه الكلمة أنّ التابع MyMethod لن يُرجع أي قيمة للشيفرة التي استدعته. كما يحتوي الصنف Father على البانية عديمة الوسائط Father. وضعت عبارتي Writeline في كلّ تابع من باب التوضيح. الآن سنعرّف صنفًا جديدًا لنسمّه Child يرث من الصنف Father على الشكل التالي:

```
class Child : Father
{

}
```

قد يبدو الصنف Child فارغًا إلّا أنّه ليس كذلك. لاحظ من السطر الأوّل لتصريح هذا الصنف كيف وضعنا النقطتان الرأسيّتان (:) ومن ثمّ اسم الصنف Father. يخبر ذلك مترجم سي شارب أنّنا نريد من الصنف Child أن يرث من الصنف Father. في الحقيقة جميع الأعضاء المعرّفة ضمن الصنف Father ستصبح موجودة تلقائيًّا ضمن الصنف Child، بل ويمكن إضافة المزيد من الأعضاء إلى الصنف Child بحسب الحاجة.

أنشئ مشروعًا جديدًا وسمّه Lesson08_01، ضع الصنفين السابقين بجوار الصنف Program ضمن فضاء الاسم Lesson08_01 ثم اكتب الشيفرة التالية ضمن التابع Main:

```
Child c = new Child();

c.MyMethod();
```

نفّذ البرنامج لتحصل على الخرج التالي:

```
Father: In Constructor
Father: In MyMethod
```

من الواضح أنّ التنفيذ سيدخل إلى بانية الصنف Father (تذكّر أنّ البانية هي أوّل تابع يُستدعى عند إنشاء الكائن) وإلى التابع MyMethod وكلاهما موجودان ضمن الصنف الأب Father.

لنضيف بعض التعديلات على الصنف Child. عدّل الصنف Child ليصبح كما يلي:

```
class Child : Father
{

  public Child()
  {
    Console.WriteLine("Child: In Constructor");
  }
}
```

لاحظ أنّنا قد أضفنا بانية عديمة الوسائط للصنف Child وبداخلها التابع WriteLine لطباعة جملة توضيحيّة. أعد تنفيذ البرنامج السابق لتحصل على الخرج التالي:

```
Father: In Constructor
Child: In Constructor
Father: In MyMethod
```

الخرج السابق منطقيّ تمامًا. عند إنشاء كائن من الصنف Child باستخدام العبارة:

```
Child c = new Child();
```

فإنّ بانية الصنف Child ستُستدعى نتيجة لذلك، وبما أنّ الصنف Child يرث من الصنف Father لذلك فإنّ بانية الصنف Father هي من ستُنفّذ أولًا ومن ثمّ بانية الصنف Child. أمّا عند استدعاء التابع MyMethod من الكائن الموجود ضمن المتغيّر c فسنحصل على رسالة الخرج الثالثة كما هو متوقّع.

لنجرّب الآن شيئًا آخر. ماذا لو أردنا استبدال محتوى التابع MyMethod بمحتوى خاص بالابن، بمعنى آخر نريد "تجاوز" تعريف التابع MyMethod الموجود في الصنف الأب Father إلى تعريف آخر للتابع MyMethod ولكنّه خاص بالصنف Child. أضف التابع التالي إلى الصنف Child:

```
public new void MyMethod()
{
  Console.WriteLine("Child: In MyMethod");
}
```

لاحظ وجود الكلمة المحجوزة new بعد مُحدّد الوصول public. وظيفة هذه الكلمة في هذا المكان هي إخفاء التابع MyMethod الموجود في الصنف Father واستبداله بالتابع MyMethod الموجود في الصنف Child.

الآن بعد تنفيذ البرنامج ستحصل على الخرج التالي:

```
Father: In Constructor
Child: In Constructor
Child: In MyMethod
```

تمّ المطلوب، لقد أُخفي التابع MyMethodالموجود ضمن الصنف الأب Fatherلصالح التابع MyMethodالموجود ضمن الصنف الابن .Childيجب أن يبدو البرنامج بعد التعديلات الأخيرة شبيهًا بما يلي:

```
using System;

namespace Lesson08_01
{
    class Father
    {
        public Father()
        {
            Console.WriteLine("Father: In Constructor");
        }

        public void MyMethod()
        {
            Console.WriteLine("Father: In MyMethod");
        }
    }

    class Child : Father
    {
        public Child()
        {
            Console.WriteLine("Child: In Constructor");
        }

        public new void MyMethod()
        {
            Console.WriteLine("Child: In MyMethod");
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Child c = new Child();

            c.MyMethod();

        }
    }
}
```

محدد الوصول protected

يُستخدم محدّد الوصول protectedفي الوراثة. فعندما نُعرّف أحد أعضاء الصنف الأب باستخدام protectedفهذا يعني أنّه لا يمكن الوصول إليه مطلقًا إلّا من خلال أعضاء الصنف الأب نفسه، أو من خلال أعضاء الصنف الابن (أو الأحفاد).

```
using System;

 namespace Lesson08_02
 {
    class Car
    {
        protected string manufacturer;

        public Car()
        {
            this.manufacturer = "Car";
        }

        public string Manufacturer
        {
            Get
            {
                return this.manufacturer;
            }
        }
    }

    class Toyota : Car
    {
        public Toyota()
        {
            this.manufacturer = "Toyota";
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Toyota toyota = new Toyota();

            Console.WriteLine(toyota.Manufacturer);
        }
    }
}
```

عند تنفيذ البرنامج ستحصل على الكلمة Toyota في الخرج. السبب في ذلك أنّ بانية الصنف Toyota تصل إلى الحقل `manufacturer` في السطر 27، رغم أنّه مصرّح عنه في الصنف الأب `Car`، وذلك لأنّه ذو محدّد وصول `protected`. من الواضح أنّ الأعضاء المصرّح عنها باستخدام محدّد الوصول `private` في الأصناف الآباء تبقى مرئيّةً فقط ضمن أعضاء الصنف الأب فحسب.

التعددية الشكلية Polymorphism

سبق وأن تحدّثنا عن التعدديّة الشكليّة، وكيف أنّها مفهوم أساسيّ في البرمجة كائنيّة التوجّه. يتمحور مفهوم التعدديّة الشكليّة حول أنّه يحق للصنف الابن إعادة صياغة تابع (أو خاصيّة) موجود في صنف أب بصورةٍ تناسبه أكثر. لقد طبّقنا هذا المفهوم قبل قليل وذلك عندما "تجاوز" التابع `MyMethod` في الصنف الابن `Child`، التابع `MyMethod` الموجود في الصنف الأب `Father`، فأصبح التابع الموجود في الابن يُعبّر عن نفسه بشكل أكثر تخصّصًا. ولكن هذه ليست هي الطريقة المثلى لتنفيذ فكرة التعدديّة الشكلية، تزوّدنا سي شارب في الواقع بأسلوب أفضل بكثير لتحقيق هذا المفهوم. هل تذكر مثال الضفدع Frog والسمكة Fish والطائر Bird وسلوكيّة الانتقال Move التي يرثونها من الصنف `Animal`؟

تناولنا هذا المثال البسيط في درس سابق. وقد ذكرنا أنّ الصنف `Animal` هو الصنف الأب
للأصناف `Frog` و `Fish` و `Bird` وهو يحتوي على التابع `Move` الذي يُعبّر عن سلوكيّة الانتقال. وبما أنّ كلًّا من الأصناف الأبناء الثلاثة تُعبّر بشكل مختلف عن عمليّة الانتقال، لذلك فنحن أمام التعدديّة الشكليّة. يحتوي
البرنامج `Lesson08_03` على صنف أب `Animal` يحوي تابعًا وحيدًا اسمه `Move`ـــ موسوم بالكلمة
المحجوزة `virtual` التي تجعل منه تابعًا ظاهريًّا يسمح للتوابع الأخرى بتجاوزه. بالإضافة إلى وجود ثلاثة أصناف أبناء للصنف `Animal` وهي `Frog` و `Fish` و `Bird`. يحتوي كل صنف من الأصناف الأبناء على
التابع `Move` مع وسم خاص هو override. تسمح هذه الكلمة للتابع في الصنف الابن أن "يتجاوز" تعريف نفس التابع في الصنف الأب (موسوم بالكلمة virtual). أعني بكلمة "تجاوز" إعادة تعريف التابع بالشكل الذي يناسب الصنف الابن. فعند الحديث عن الانتقال، فالذي يناسب الضفدع `Frog` هو القفز، والذي يناسب السمكة `Fish` هو السباحة، والذي يناسب الطائر `Bird` بالطبع هو الطيران. وبالمناسبة فإنّ التابع ToString الموجود في الصنف Object هو تابع ظاهريّ (موسوم بالكلمة virtual) ليسمح لأي صنف آخر بتجاوزه.

```
using System;

namespace Lesson08_03
{
    class Animal
    {
        public virtual void Move()
        {
            Console.WriteLine("Animal: Move General Method");
        }
    }

    class Frog : Animal
    {
        public override void Move()
        {
            Console.WriteLine("Frog - Move: jumping 20 cm");
        }
    }

    class Bird : Animal
    {
        public override void Move()
        {
            Console.WriteLine("Brid - Move: flying 10 m");
        }

    }

    class Fish : Animal
    {
        public override void Move()
        {
            Console.WriteLine("Fish - Move: swimming 1 m");
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Frog frog = new Frog();
            Fish fish = new Fish();
            Bird bird = new Bird();
```

```
44
45              frog.Move();
46              fish.Move();
47              bird.Move();
48          }
49      }
50  }
```

نفّذ البرنامج السابق لتحصل على الخرج التالي:

```
Frog - Move: jumping 20 cm
Fish - Move: swimming 1 m
Brid - Move: flying 10 m
```

لاحظ كيف يُعبّر كلّ كائن من الأصناف الأبناء عن التابع Moveبالشكل الذي يناسبه. وواضح أنّ التابع Moveالموجود في الصنف الأب Animalلا يُستدعى مطلقًا. ولكن في بعض الحالات قد نرغب أن يُستدعى التابع المُتجاوَز الموجود في الصنف الأب لإنجاز بعض المهام ومن ثمّ نتابع العمل ضمن التابع المُتجاوِز. يمكننا ذلك ببساطة من خلال استخدام الكلمة المحجوزة baseالتي تُشير إلى الصنف الأب الذي يرث منه الابن. لاستدعاء التابع Moveالموجود في الصنف الأب Animalوذلك من خلال التابع Moveالموجود في الصنف Frogأضف العبارة التالية بعد السطر 16 مباشرةً قبل أي عبارة أخرى، ليصبح هذا التابع على الشكل:

```
public override void Move()
{
    base.Move();
    Console.WriteLine("Frog - Move: jumping 20 cm");
}
```

أعد تنفيذ البرنامج لتحصل على الخرج التالي:

```
Animal: Move General Method
Frog - Move: jumping 20 cm
Fish - Move: swimming 1 m
Brid - Move: flying 10 m
```

انظر كيف استُدعي التابع Move الموجود في الصنف الأب Animal ومن ثمّ استُدعي التابع Move الموجود في الصنف الابن Frog.

التحويل بين الأنواع

سنتناول في هذه الفقرة سلوكًا قد يبدو غريبًا بعض الشيء، ولكنّه مهم وأساسيّ وسيصادفك في معظم البرامج التي تكتبها باستخدام سي شارب. أعد البرنامج إلى حالته الأصلية (أي أزل العبارة base.Move()). امسح محتويات التابع Main واستبدلها بالشيفرة التالية:

```
Animal animal = new Frog();
animal.Move();
```

العبارة الأولى غريبة قليلًا أليس كذلك؟ في الحقيقة الوضع طبيعي تمامًا، فبما أنّ الصنف Animal هو صنف أب للصنف Frog لذلك فيستطيع أيّ متغيّر مصرّح عنه على أنّه من النوع Animal (في مثالنا هذا هو المتغيّر animal) أن يخزّن مرجع إلى كائن من الصنف Frog (تذكّر أنّ التعبير new Frog() يولّد مرجع لكائن من الصنف Frog). نفّذ البرنامج الآن لتحصل على الخرج التالي:

```
Frog - Move: jumping 20 cm
```

يبدو أنّ برنامجنا ذكيّ كفاية لكي يعرف أنّ الكائن الذي يشير إليه المتغيّر animal هو كائن من الصنف Frog. في الحقيقة يحصل هنا تحويل ضمني بين الكائنات، ولكن إذا فعلنا العكس، أي أسندنا مرجع لكائن من الصنف Animal إلى متغيّر من النوع Frog فسنحصل على خطأ أثناء ترجمة البرنامج. امسح محتويات التابع Main واستبدلها بالشيفرة التالية:

```
Animal animal = new Frog();
Frog frog = animal;
```

نحاول في السطر الثاني أن نُسند المتغيّر animal من النوع Animal إلى المتغيّر frog من النوع Frog، فنحصل على الخطأ التالي عند محاولة تنفيذ البرنامج:

```
Cannot implicitly convert type 'Lesson08_03.Animal' to
'Lesson08_03.Frog'. An explicit conversion exists (are you missing a
cast?)
```

يخبرنا هذا الخطأ أنّه لا يمكن التحويل بشكل ضمنيّ من النوع `Animal` إلى النوع `Frog` ويقترح علينا استخدام عامل التحويل بين الأنواع casting (هل تذكره؟). رغم أنّ المتغيّر `animal` يحمل مرجع إل كائن من الصنف `Frog` في حقيقة الأمر (انظر السطر الأوّل من الشيفرة السابقة) إلّا أنّنا عند محاولتنا إسناد المتغيّر `animal` إلى المتغيّر `frog` حصلنا على خطأ. السبب في ذلك هو أنّه لا يحدث تحويل ضمنيّ بين الأنواع implicit conversion وإنّما يتطلّب الأمر إجراء تحويل صريح باستخدام عامل التحويل بين الأنواع. إذا استبدلت السطر الثاني من الشيفرة السابقة بالسطر التالي، ستكون الأمور على ما يرام:

```
Frog frog = (Frog)animal;
```

لاحظ كيف وضعنا عامل التحويل (`Frog`) أمام المتغيّر `animal`. سيضمن ذلك حدوث التحويل المطلوب دون أيّ مشاكل.

لكي تريح نفسك من التفكير متى يحدث التحويل الضمنيّ ومتى يجب استخدام التحويل الصريح، تذكّر منّي القاعدة التالية: "في حياتنا اليوميّة، كثيرًا ما يحتوي الأب ابنه، ولكنّ العكس ليس صحيحًا".

أعلم أنّ لهذه القاعدة شواذ في واقعنا، ولكنّها في البرمجة لا تخيب! فالمتغيّر من النوع الأب يستطيع استقبال أي مرجع لكائن من صنف ابن، ولكنّ العكس ليس صحيح ما لم نستخدم التحويل الصريح بين الأنواع.

تحدث ظاهرة التحويل بين الأنواع بالنسبة للأنواع المضمّنة built-in أيضًا. فهناك تحويلات تحدث ضمنيًّا، وأخرى تحدث بتدخّل من المبرمج باستخدام عامل التحويل بين الأنواع، ولكن مع فرق جوهريّ. ففي هذه الحالة ليس بالضرورة أن يكون بين الأنواع التي تجري عمليّة التحويل فيما بينها أي علاقة وراثة. فمثلًا يمكن التحويل ضمنيًّا بين متغيّر من النوع `int` إلى آخر من النوع `double`:

```
int i = 6;
double d = i;
```

أمّا إذا حاولنا فعل العكس:

```
double d = 6;
int i = d;
```

فسنحصل على نفس الخطأ السابق الذي يخبرنا بوجوب استخدام التحويل الصريح بين الأنواع. يمكن حل هذه المشكلة ببساطة باستخدام عامل التحويل (`int`) ووضعه أمام المتغيّر `d` في السطر الثاني:

```
double d = 6;
int i = (int)d;
```

نخبر المترجم هنا أنّنا نريد التحويل فعليًّا من double إلى int. ستحتاج إلى مثل هذه التقنيّة دومًا إذا كانت عمليّة التحويل ستؤدّي إلى ضياع في البيانات. فالتحويل من double إلى int سيؤدّي إلى ضياع القيمة على يمين الفاصلة العشريّة لأنّ المتغيّرات من النوع int لا تقبلها. وكذلك الأمر عند التحويل من float إلى double لأنّ المتغيّرات من النوع float ذات دقّة أقل من المتغيّرات من النوع double، وهكذا.