مقدمةٌ عن البرمجة الكائنية
object oriented programming
OOP

بسم الله، و الصلاة و السلام علي رسول الله: الشرح التالي هو تعديلٌ غير كبير لشرح كنتُ قد كتبتُه لمجموعةٍ من زملائي في كلية الهندسة (في السنة قبل الأخيرة لنا فيها)، أشرح فيه نمط البرمجة المسمي بالبرمجة المَقُودة بالكائنات object oriented programming أو اختصاراً (البرمجة الكائنية). و قد وجدتُه بعد كل تلك الفترة التي نسيتُه فيها بسيطاً و ربما جيداً، فقلتُ أضعُه للناس ليستفيدوا منه. و ربما أقوم في المستقبل بإذن الله تعالي بتكملته و/أو التعديل فيه.

ماهية البرمجة الكائنية:

لكي نفهم ماهية و خصائص نمط البرمجة الكائنية oop paradigm يجب علينا أن نتفهم المثال التالي:

فلنفترض أننا نريد كتابة برنامج يقوم بإنشاء خَمس حساباتٍ لخمسة أشخاصٍ في خمس شركاتٍ مختلفة، و يقوم بحساب مستحقاتهم المالية لدي كل شركةٍ بعد مرور ثلاث سنوات بمعرفة نسبة الربح السنوي المُعتادة من رأس المال. مع ملاحظة أن الكود سيكون بنمط البرمجة العادي (أي بدون استخدام الأصناف classes و الكائنات objects).

```
double account1 = 100, profit1 = 0.1;
double account2 = 150, profit2 = 0.15;
double account3 = 200, profit3 = 0.20;
double account4 = 175, profit4 = 0.25;
double account5 = 200, profit5 = 0.30;


double after_years(double old_account, double profit, int years){
    for(int i = 1; i <= years; i++){
        old_account = old_account * profit + old_account;
    }
    return old_account;
}

void main(){
    account1 = after_years(account1, profit1, 3);
    account2 = after_years(account2, profit2, 3);
    account3 = after_years(account3, profit3, 3);
    account4 = after_years(account4, profit4, 3);
    account5 = after_years(account5, profit5, 3);
}
```

بملاحظة البرنامج السابق سوف نري أننا لتمثيل كل حسابٍ من الحسابات المالية استخدمنا متغيرين من نوع double:

- الأول لتمثيل النقود التي يحتويها الحساب account
- الثاني لتمثيل نسبة الربح profit

و تم تكرار هذا مع كل حسابٍ من الحسابات المالية، بالطبع مع تغيير أسماء المتغيرات للتمييز بينها.
إذاً فالسؤال المنطقي الآن: هل يمكنني كمبرمجٍ أن أختصر كل ذلك الكود ما دام الكثير منه مكرراً بالفعل ؟

و الجواب: نعم، و هذا هو نمط البرمجة الكائنية.
فكل ما نفعله في البرمجة الكائنية أننا نضم الكود الذي يُشكل مع بعضه وحدةً بنائيةً متكاملة (سواء أكانت متغيرات أم دوال) و نضعها في مكانٍ واحدٍ و نُطلق عليها إسماً مُعيناً، ثم نكتفي بعد ذلك بأن نأمر جهاز الحاسب بصنع نسخةٍ من ذلك الكود و نُطلق علي تلك النسخة إسماً خاصاً بها.
و بالتطبيق علي المثال السابق نري أن التقسيم الأوَّلي لما سنحصل عليه سيكون كالتالي:

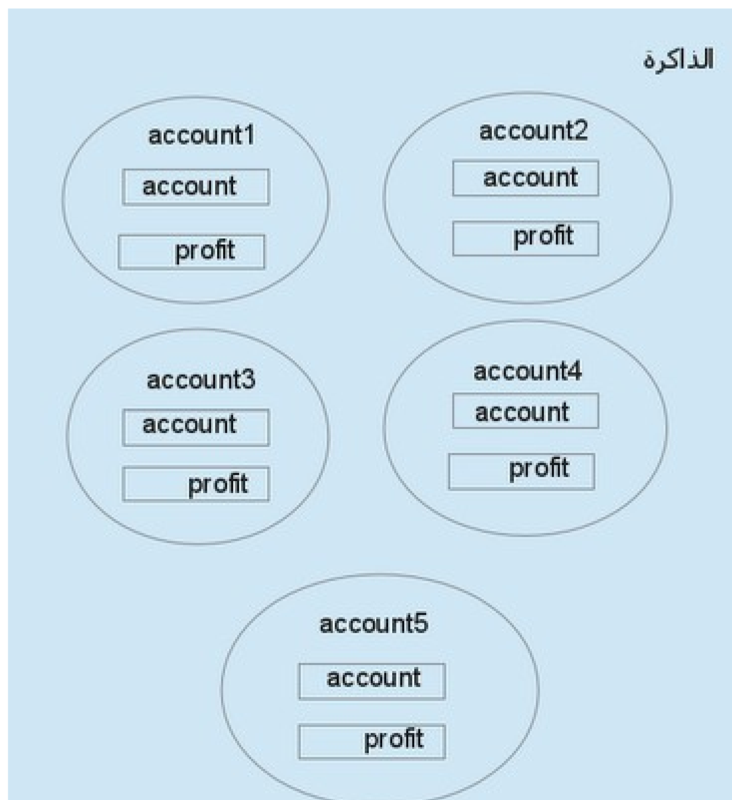
```
double account, profit;
double after_years(double old_account, double profit, int years){
for(int i = 1; i <= years; i++){
          old_account = old_account * profit + old_account;
     }
     return old_account;
}
```

يمكن أن نُطلق علي الكود السابق إسم caccount اختصاراً لـ company account و هو اسمٌ معبرٌ جداً عن وظيفته.

و في البرنامج الأساسي سنكتفي بطلب نسخ الكود caccount خمس مرات من مترجم اللغة بأسماء account1 account2 account3 account4 account5.

```
caccount account1;
caccount account2;
caccount account3;
caccount account4;
caccount account5;
```

فماذا سيفعل الحاسوب بعد حجز أماكن لتلك الكائنات في الذاكرة ؟
سوف يقوم بعمل ما يلي تقريباً:



و بداخل كل وحدةٍ من الوحدات المحجوزة تُوجد المتغيرات الموجودة في الكود المُكرر الذي كتبناه من قبل و وضعناه جانباً.
و حينما نكتب في البرنامج:

account1.account

فإننا نعني المتغير account الذي هو من نوع double و المحجوز للنسخة المُسماة account1.
و هكذا مع:

account2.account

و غيرها.

و لو أردنا أن ننفذ دالة علي النسخة account1 بحيث تعملُ باستخدام متغيراتها الخاصة فإننا نكتب الكود التالي:

```
account1.methodname();
```

مثال:

```
account1.newaccount();
```

إلي الآن و الأمر مفهوم: يوجد لدينا كود مكرر في البرنامج فقمنا بكتابته مرةً واحدةً و أطلقنا عليه اسماً مُعيناً، ثم صنعنا منه ما نريد من نسخ.
و لكن هل هذا كل شيء ؟

لا.
فماذا لو أنني أردتُ إعطاء قيم ابتدائية للمتغيرات الموجودة داخل النسخ الجديدة في نفس سطر تعريف كل نسخةٍ منها، كيف يمكنني فعل هذا ؟
و ماذا لو أنني أردتُ منع الوصول المباشر للمتغيرات داخل النسخ، مثل الأمر:

```
account1.account = 1000;
```

و أردتُ أن تكون هذه المتغيرات متاحةً فقط للدوال الموجودة معها في الكود المُكرر فقط ؟
إلي آخر تلك الأسئلة التي تُوضح إجاباتُها خصائصَ نمط البرمجة الكائني.

سوف نقوم بتوضيح إجابات تلك الأسئلة فيما يلي، و لكن أولاً سوف نقوم بشرح المُصطلح العلمي المقابل لبعض الكلمات التي ذكرناها من قبل أثناء الشرح:

التوصيف	المُصطلح العلمي العربي	المُصطلح العلمي الإنقليزي
كود مكرر	صِنْف	class
نُسخة	نسخة، كائن	Object, instance
متغير داخل صنف	حقل بيانات	Data field
دالة داخل صنف	دالة حالة	Instance method

و لنجب الآن عن الأسئلة التي سبق و طرحناها.
أما كيفية إعطاء حقول البيانات قيماً ابتدائية في نفس جملة التعريف فيتم عن طريق ما يُسمي بالمُشَيِّد constructor، و هو دالةٌ لها نفس اسم الصنف class بدون أي نوعٍ من المُخرجات (ليس حتي void).
و ينقسم المُشَيِّد إلي نوعين:

•المُشَيِّد الافتراضي default constructor
•المُشَيِّد ذو المُدخَلات parameterized constructor

و هذا هو المثال السابق بالكامل (مع إضافة مُشَيِّداتٍ من ذوات المُدخَلات):

```
public class caccount {
   double account, profit;

   public caccount(double account, double profit){
      this.account = account;
      this.profit = profit;
   }
   public static void main(String[] args){
      caccount account1 = new caccount(100, 0.1);
      caccount account2 = new caccount(150, 0.15);
      caccount account3 = new caccount(175, 0.20);
      caccount account4 = new caccount(200, 0.25);
      caccount account5 = new caccount(250, 0.3);
   }

   double after_years(double old_account, double profit, int years){
      for(int i = 1; i <= years; i++){
         old_account = old_account * profit + old_account;
      }
      return old_account;
   }
}
```

و يقوم البرنامج بحجز مكانٍ في الذاكرة للكائن الجديد ثم تنفيذ المُشَيِّد الافتراضي لو كتبنا ما يلي:

```
caccount account1 = new caccount();
```

و لاحظ عدم وجود قيم بين قوسي المُشَيِّد المُستدعَي.
بينما يقوم البرنامج بحجز مكان الذاكرة ثم تنفيذ المُشَيِّد ذي المُدخلات لو كتبنا ما يلي:

```
caccount account1 = new caccount(100, 1.0);
```

و لاحظ وجود قيمٍ بين قوسي المُشَيِّد المُستدعَي. و في هذه الحالة يكون:

```
account2.account = 100
account2.profit = 0.1
```

أما منع الوصول المّباشر لحقول البيانات فيكون ببساطة بإعطائها الخاصية private عند تعريفها، مثل:

```
private double account;
```

و لو أننا قمنا بتعريف كل حقول البيانات الموجودة في الصنف و أعطيناها الخاصية private فسنجد أننا وصلنا إلي جعل الصنف أشبه بالكبسولة؛ حيث لا يُمكن الوصول لحقول البيانات إلا من خلال الدوال الموجودة في الصنف، و هو ما يُعرف بالكبسلة encapsulation.

و كذلك ينطبق مفهوم الكبسلة علي كود الدوال الموجودة داخل الصنف، حيث لا يُمكن للمستخدم في البرنامج النهائي أن يُغير من ذلك الكود، بل هو مجرد مستخدمٍ له فقط.

أين تتم كتابة الأكواد:

في هذه المرحلة نكون قد وصلنا إلي تصورٍ جيدٍ لمفهوم البرمجة الكائنية oop و كيفية الاستفادة منها في تصغير حجم البرنامج و حماية حقول البيانات من تدخل المستخدم النهائي، و لكننا حتي الآن لا ندري أين تتم كتابة كود الصنف الفرعي و أين سنكتب الكود الذي سيستخدم هذا الصنف ؟
و هذه نقطةٌ هامةٌ للغاية تختلف فيها لغات البرمجة؛ ففي لغة الـ++C مثلاً سوف نجد أننا سوف نستخدم ملفات الـ header files (أو ملفات التَرْويسَة) لكي نكتب فيها كود الصنف و يكون لها الامتداد (h.)، و سيكون علينا أن نكتب الكود بنفس طريقة الـ++C التي تشترط كتابة مُلخص الصنف specification ثم بعد ذلك يأتي كوده التفصيلي (أو ما يُسمي بالبناء implementation).
و كذلك فإنه يمكننا أن نكتب التلخيص في ملف ترويسة و البناء داخل ملف آخر له الامتداد cpp. مع كتابة الجملة التالية في بداية هذا الملف الجديد:

#include "caccount.h";

لجعل المُترجم يفهم أن الملفين مرتبطان ببعضهما البعض.
هذا بالنسبة للـ++C، أما في حالة الـ java فيمكننا كتابة كود الصنف في نفس ملف البرنامج الأصلي، أو في ملفٍ مستقل لكن بحيث:

- يشتمل كل ملفٍ مستقلٍ علي صنفٍ واحدٍ فقط يماثله في الاسم، و بإمكاننا كتابة أصنافٍ أخري داخل ذلك الصنف الواحد.
- يكون امتداد الملف المستقل(java.)

مفهوم الوراثة inheritance و علاقته بمبدأ إعادة الاستخدام:

بعد الانتهاء من كتابة برنامجٍ ضخم بنمط البرمجة الكائنية سوف نجد أنه قد تم كتابة عددٍ كبيرٍ من الأصناف المُستقلة التي سهلت علينا العمل و أسرعته بمعدلاتٍ فائقة.
لكن السؤال الحيوي هنا: هل يمكنني الاستفادة منها بصورةٍ أكبر ؟

بالطبع نعم؛ حيث يمكنني إعادة استخدام reuse هذه التصنيفات الموجودة عندي في كتابة برامجي الجديدة دون الحاجة إلي إعادة كتابة كودها مرةً أخري أي أنني أصنع مكتبتي الخاصة بي، تماماً كالمكتبات القياسية للغات البرمجة، حيث أن الأصناف التي نستخدمها في برامجنا التي نكتبها بتلك اللغات ليست إلا أكواداً قام بكتابتها متخصصون في شركة البرمجيات و أراحونا من عناء كتابتها منذ البداية.
و علي سبيل المثال لو أنني احتجتُ للصنف caccount أثناء كتابتي لبرنامجٍ جديدٍ بلغة الـ++C فسوف أقوم فقط بضمه للبرنامج بالعبارة:

#include "caccount.h";

و بذلك يمكنني استخدامه بمنتهي الحرية.

لكن ماذا لو أردتُ استعمال هذا الصنف مع إحداث بعض التغييرات فيه (كتابة حقول بياناتٍ إضافية، أو كتابة دوالٍ إضافية، أو حتي تعديل أكواد دوالٍ موجودةٍ من قبل) مع الاحتفاظ بهيكل و مكونات الصنف الأساسي كما هي، فما العمل ؟
الحل هنا يكمنُ في مفهوم الوراثة؛ فحينما أُخبر اللغة أنني أريد كتابة تصنيفٍ جديدٍ فلنفترض أنه new_caccount يرث الصنف caccount فإنه يكون مفهوماً لديها أن الصنف الجديد يحتوي علي كل حقول البيانات و الدوال التي يحتويها الصنف القديم مع وجود إمكانية التعديل كما سبق و شرحنا.
و تختلف اللغات فيما بينها في مدي و كيفية الوراثة: أهي وراثة متعددة multiple inheritance بما يعني أن الصنف الواحد يمكنه وراثة عددٍ غير محدودٍ من التصنيفات الأخري كما في الـ++C و الـ eiffel، أم هي وراثة أحادية أي أن الصنف لا يمكنه وراثة أكثر من صنفٍ واحدٍ فقط كما في لغات الـ java و الـ#C. و ربما نتحدث عن هذا بالتفصيل في شروحاتٍ قادمةٍ بإذن الله تعالي.

إلي هنا نكون قد أنهينا المفاهيم الأساسية في عالم البرمجة الكائنية، و لكن أشياء كثيرة للغاية ما زالت تنتظر و لكنها تتأثر

بماهية اللغة التي سنبرمج بها، لذلك ستكون هذه الأشياء في الــ++C مختلفـة الشـكل و الخصـائص عنهـا فـي الــ java و عنهما في الـ#C.
و منها:

- الواجهات interfaces
- الهياكل structs
- التعدادات enumerations
- الوسائط delegates
- الاحداث و تعهد الأحداث events and events handling

و غيرهن.

م. وائل حسن محمد علي – أبو إياس
http://afkar-abo-eyas.blogspot.com/