المحاضرة الأولى

10

مقدمة:

لم نكن أنا وزملائي في البداية متأكدين من جدوى كتابة محاضرات مادة لغات البرمجة وذلك لعدة أسباب منها أننا اعتقدنا أن الدكتور * محيي الدين مراد * سيقدم للطلاب الـ slides التي سيشرح عليها في المحاضرة.

ما جعلنا نغير رأينا هو الأمور التالية:

١- لم يقدم الدكتور الـ slides للطلاب.

٢- كان الانطباع العام عند الطلاب عن المحاضرة الأولى غير مشجع بسبب طريقة طرح الدكتور للمعلومات.

لهذه الأسباب وجدنا أن من الواجب علينا تقديم مادة علمية تساعد الزملاء على دراسة المادة وتخفف العبء الدراسي عنهم، ولكن أحب أن أنوه إلى أن هذه المحاضرات ليست بديلاً عن المرجع *علماً أني سأبذل قصارى جهدي لتكون غنية بالمعلومات وبسيطة في الأسلوب* وأتمنى أن أوفق لهذا الهدف..

أتمنى من زملائي أن لا يبخلوا علي بملاحظاتهم وانتقاداتهم على المحاضرة، وذلك في سبيل تطويرها قدر المستطاع لتكون إسهاماً في تقديم شيء مفيد لا مجرد محاضرة روتينية.

للاستفسار أو الملاحظات أو النقد يرجى المراسلة على البريد التالي:

lectures_team@hotmail.com

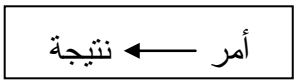
❖❖❖❖❖❖❖❖❖❖❖❖❖❖❖❖❖❖❖❖❖❖❖❖❖❖❖❖❖❖

مقدمة عن لغات البرمجة:

من المتعارف عليه في الكلية أن هذه المادة تهتم بلغة البرمجة Java ولكن هذا الكلام ليس دقيقاً تماماً، فالمادة تركز على مفاهيم البرمجة غرضية التوجه OOP وتقدمها من خلال لغة Java التي تتميز بأنها لغة غرضية التوجه تماماً أي لا يمكن كتابة برنامج غير غرضي التوجه باستخدام Java.

تصنف لغات البرمجة كالتالي:

- **اللغات اللاإجرائية:** هي لغات أوامر، أي أن المبرمج يتعامل مع اللغة عن طريق مجموعة من الواصفات تمثل الدخل، ووظيفة اللغة هي إرجاع خرج مناسب لذلك الدخل.



مثالها : لغة الاستعلامات SQL التي سنتعلمها في الفصل القادم..

- **اللغات الإجرائية:** وتقسم إلى:
  - **لغات غير بنيوية:** لا تحوي بنية برمجية منظمة، و تستخدم فيها تعليمات القفز بشكل أساسي.
    مثالها: Fortran - Basic
  - **لغات بنيوية:** التعليمات البرمجية منظمة ضمن (Scope)، ولا يفترض أن نستخدم تعليمات القفز فيها، وهذا النوع بدوره يقسم إلى قسمين:
    - برمجة إجرائية: ومثالها C++ - Pascal.
    - برمجة غرضية التوجه: ومثالها C# - Java.

ضمن التصنيف السابق نلاحظ أن لغة Java صنفت ضمن اللغات التي تدعم الـ OOP وفي الحقيقة هي لغة لا تدعم إلا الـ OOP أي أن جميع عناصر اللغة هي objects ولا يمكن تعريف توابع أو متحولات إلا ضمن object.

<u>ميزات Java:</u>

١- جميع مكونات البرنامج هي objects.

٢- لغة محمولة: أي أنها تعمل على أي جهاز وبالتالي على أي نظام تشغيل!

ما السبب ؟؟

السر في هذه الميزة الرائعة لـ Java يكمن في وجود (Java Virtual Machine) أو اختصاراً JVM.

<u>ما هو JVM ؟</u>

هو عبارة عن معالج افتراضي **يحول** مجموعة تعليمات مكتوبة بـ (Bite Code) إلى لغة الآلة التي يفهمها معالج الآلة التي تشغل برنامج Java، وهذه التعليمات ناتجة عن ترجمة الملفات ذات اللاحقة (java.) والتي تحوي تعليمات Java وينتج عن هذه الترجمة ملفات ذات اللاحقة (class.) والتي تحوي

الـ (Bite Code) ، وتكون نتيجة تنفيذ هذه التعليمات عبر JVM هي ناتج تنفيذ برنامج Java، وبالتالي لا يمكن تشغيل أي برنامج مكتوب بلغة Java ما لم يتوافر JVM على الجهاز.

٣- برمجيات Java تعمل شكل تطبيقات مكتبية (server - client)، كما تعمل على صفحات الـ web، وبما أن برنامج Java مكون من عدة classes فيمكن استدعاء مكتبات جاهزة وموجودة على مواقع أخرى..

٤- إن تعدد استخدامات Java جعل لدينا عدة إصدارات من اللغة وذلك تبعاً للآلة التي تشغل برنامج Java ، فمن المنطقي أن تتنوع التطبيقات التي أحتاجها تبعاً لتنوع الآلات * mobile , server ... * (الفرق في المكتبات المستخدمة وبالتالي تناسب الـ JVM) لذا نسمع عن المصطلحات التالية:
J2ee , J2me ...

في نهاية هذه المقدمة ننوه إلى أن المرجع المعتمد للمادة هو كتاب:
Thinking in Java, 3rd Edition.

والذي يمكن تحميله من الإنترنت (بتنسيق HTML) من أحد الروابط التالية:
http://www.pythoncriticalmass.com/downloads/TIJ-3rd-edition4.0.zip
http://carti.ss.pub.ro/eckel/TIJ-3rd-edition4.0.zip

ولمن يحب ملفات الـ pdf الرابط التالي:
http://www.nmr.unisi.it/html/java/TIJ-3rd-edition-beta-PDF.zip

هناك نقطة مهمة أيضاً وهي أن الدكتور أعطى في هذه المحاضرة ملخصاً عن المنهاج بالكامل فلا تقلق إن لم تتوضح لديك بعض النقاط لأننا سندرسها بالتفصيل ضمن السنة..

ملاحظة هامة:

قبل البدء بأي مشروع برمجي يجب القيام بدراسة مفصلة لـ Model (نموذج) المشروع وبالتالي نحن بحاجة لما يدعى بـ UML ، وهو عبارة عن توصيف للـ objects والعلاقات بينها.
وبما أن جميع عناصر Java هي objects فنحن بحاجة لأداة نمذجة سنتطرق لدراستها لاحقاً.
نصيحة: أي مشكلة تظهر خلال البرنامج يكون حلها من خلال تعديل النموذج وليس عبر ترقيع البرنامج!!

الصفوف (Classes):

قبل البدء بالحديث عن البرمجة غرضية التوجه لابد من التذكير بالبنية الأساسية فيها ألا وهي الـ class.
الصف عبارة عن بنية برمجية تقوم بتجميع مجموعة من المتحولات (data members) من أنماط متعددة (قد تكون أيضاً صفوفاً) وإلى جانبها مجموعة من التوابع (methods) التي تقوم بوظائف معينة على هذه المتحولات **وتؤمن وسيلة تخاطب الصف مع العالم الخارجي** (بقية صفوف البرنامج)، أي أن مفهوم الصف قريب جداً من مفهوم النمط (type) ولكن الفرق أنه يحوي جميع ما يحتاجه ضمن بنيته الداخلية، وهذا ما يدعى اصطلاحاً بالكبسلة (Encapsulation).

مثال: عندما نعرف الصف *مكدس*، فإننا نضع ضمنه البنية البرمجية للمكدس (مصفوفة أو مؤشرات) كما نعرف ضمنه توابع الإضافة والحذف..
الفوائد من هذا التجميع كثيرة ومتعددة وستتضح تدريجياً من خلال المحاضرة، ولكن يجب أن نشرح هنا إحدى أهم الفوائد وهي حماية المعلومات:
عندما نعرف متحولات أو توابع ضمن الصف فإننا مضطرون لاختيار نوع هذا التعريف وبالتالي علينا الاختيار من الأنواع التالية:

١. **public**: وتعني أن المتحول أو التابع عام، أي أنه يمكن رؤيته والتحكم به من خلال توابع أخرى خارج الصف.
٢. **private**: وتعني أن المتحول أو التابع خاص، أي لا يمكن رؤيته والتحكم به إلا من خلال التوابع المعرفة ضمن الصف ذاته.
٣. **protected**: لها نفس معنى private والفرق أن هذه المتحولات والتوابع تصبح مسموحة للابن عند إجراء وراثة –سنشرحها بعد قليل–، بينما لا يسمح له بالوصول إلى المتحولات والتوابع المعرفة private.
٤. **default**: أو (friendly) كما يحلو للدكتور تسميتها، وتتعلق بطريقة بناء برنامج Java:
إن برنامج Java يتكون من عدة packages يحتوي كل منها على مجموعة من الصفوف، وعندما أعرف خاصة أو تابع على أنه friendly فإنني أكون قد أعطيته نفس خصائص الـ private ولكن على مستوى الـ package وليس على مستوى الـ class، أي أنه يمكن أن يرى ضمن أي تابع في نفس الـ package بغض النظر عن كونه في نفس الـ class أو لا.

المتعارف عليه في البرمجة غرضية التوجه هو أن المتحولات ضمن الـ class يجب أن تكون private أو protected وبالتالي لا يسمح للمبرمج الذي يستخدم الـ class بتغيير قيمتها إلا عن طريق التوابع التي يتيحها

لها مبرمج الـ class والتي تكون من النوع public والتي يستطيع من خلالها ضبط القيم التي تسند للمتحولات، وبالتالي يحمي هذه المتحولات من الاستخدام الخاطئ أو من وضع معلومات خاطئة فيها.

ملاحظة(مهمة من أجل الفحص): من الواضح أن مفهوم الـ class يقرب التفكير البرمجي كثيراً من التفكير الواقعي إذ أننا نستطيع اعتبار جميع الأشياء من حولنا صفوفاً تحوي خصائص وطرائق تتحكم بهذه الخصائص، ولكن عندما نريد تمثيل أي شيء واقعي برمجياً عن طريق class فإننا ننتقي من خصائصه الكثيرة ما يهمنا في مسألتنا فقط.

الفرق بين Class و Object:

إن فهم الفرق بين المصطلحين السابقين أساسي جداً لاستيعاب فكرة الـ OOP..
الـ class هو مفهوم مجرد (نمط) ولا يمثل بشكل فيزيائي على الحاسب، بينما الـ object فهو عبارة عن التمثيل الفيزيائي للـ class وهو الذي يحجز مساحة في الذاكرة.

مثال: لنعد إلى مثال الرسوميات: عندما أعرف نمطاً يدعى دائرة أكون قد عرفت class وليكن:

```
class circle{
      int diameter;
      void draw (...){
            ...
      }
}
```

ولكنني عندما أحتاج لرسم دائرة على الشاشة فإنني بحاجة لتعريف نسخة (instance) من النمط *circle* لها وجود فيزيائي في الذاكرة وتحتوي نسخة من المتحول diameter كما تستطيع استخدام التابع draw.

- اسم الـ class يجب أن يكون معبراً عن شيء مجرد فلا يفترض أن نسمي class ما باسم (MyPen) والصحيح أن يسمى (Pen) ويمكن أن نسمي الـ object التي نعرفها من هذا الـ class باسم (MyPen).

- الـ Object: يشبه كثيراً مفهوم المتحول العادي (... int, float) والذي تتغير قيمته ضمن البرنامج، وبالتالي فإن الـ Object قد يحمل عدة قيم ضمن البرنامج.
مثال: إذا كان لدينا class يدعى student وعرفنا object منه فإن هذا الـ object قد يحمل معلومات الطالب **A** في وقت ما ومعلومات الطالب **B** في وقت آخر.

الـ Object بين c++ و Java:

لنجري مقارنة بين اللغتين لنعرف كيف تتعامل كل منهما مع الـ object:

في c++: يمكن أن تعرف object بشكل طبيعي كما تعرف أي متحول وبالتالي فإن دورة حياته تنتهي مع نهاية الكتلة البرمجية (scope) كأي متحول آخر.
كما يمكننا تعريف object بشكل ديناميكي وذلك عن طريق تعليمة new، وبالتالي فإن هدمه يجب أن يتم يدوياً عن طريق تعليمة delete.

مثال: لنأخذ المثال السابق والذي عرفنا فيه الصف circle:

```
circle c1();                          (1)
circle* c2 = new circle();            (2)
```

المتحول c1 عبارة عن object من الصف circle، بينما المتحول c2 عبارة عن مؤشر على object من الصف circle، لذلك فإن التعامل معه يكون عن طريق تعليمتي new و delete.

في Java: لا يمكن تعريف object في Java إلا ديناميكياً وبالتالي نحن بحاجة لتعليمة new دوماً عند التعريف، ولكن ماذا عن الهدم؟؟
في الحقيقة ليس عليك هنا أن تهتم بهدم الـ objects إذ أن JVM تحمل عنك هذا العبء عن طريق الأداة (garbage collector) والتي تتحسس أن الـ object لم يعد مستخدماً فتقوم بهدمه تلقائياً.

مثال:

```
circle c = new circle();
```

المتحول c عبارة عن مؤشر على object من الصف circle.

ولكن ألا تلاحظ أن هناك شيئاً مختلفاً؟؟ لماذا لم نضع * قبل اسم المتحول حتى ندل على أنه مؤشر؟
هنا يبرز فرق هام بين c++ و Java، وهو أن Java تجعلك تتعامل مع المؤشرات من غير أن تشعر بوجودها، إذ أن اللغة مكونة من objects وجميع هذه الـ objects معرفة ديناميكياً وبالتالي فإننا نتعامل معها عن طريق المؤشرات ولكن بدون أن نضطر لتعريفها صراحة، وبالتالي فإن عملية الإسناد بين متحولين من نوع circle على سبيل المثال إنما هو إسناد مؤشرات، وتمرير object لتابع يكون دائماً (by reference)
سنفصل في هذا الكلام في المحاضرات القادمة.

أساسيات الـــ OOP :

تم شرح هذه الأساسيات في مقرر (البرمجة٣) في السنة الماضية وسنعرج عليها سريعاً:

١. الكبسلة (Encapsulation) : تم شرحها ضمن تعريف الـــ class..

٢. الوراثة (Inheritance) : بفرض أنني عرفت class ما، ومن ثم احتجت لتعريف class آخـــر يـــشابه الأول في جميع صفاته ولكنه يزيد عليه في بعض الخصائص..

أو أنني أريد تعريف مجموعة classes تتفق في مجموعة من الخصائص وتتفرد عن بعضها في بعـــض الخصائص ..

لتتضح الصورة سأضرب مثالاً: لدي برنامج رسومي أحتاج فيه لدائرة ومربع ومثلث..

جميع الصفوف السابقة تشترك مع بعضها في أنها أشكال ويجب أن يكون لها تابع رسم وإن كان مـــضمونه يختلف من شكل لآخر.

ليس من العملي أن أعيد كتابة جميع الأشياء المشتركة بين الصفوف في الحالات السابقة، والبديل هو اشتقاق الصف الذي أريد أن أزيد عليه من الصف الأصلي، وبالتالي سيملك الصف الابن جميع خصائص الـــصف الأب، ويستطيع الوصول إليها مباشرة، ماعدا الخصائص ذات النوع private.

بالعودة لبرنامج الرسوميات: الحل هو تعريف class رئيسي يدعى (shape) يحوي تابع الرســـم draw نشتق منه جميع الأشكال السابقة ونعيد تعريف التابع draw فيها.

٣. تعددية الأشكال (Polymorphism) : وهي أهم ميزة تنتج عن الوراثة، وليفهم مغزاها جيداً سنتكلم عنها من خلال الأمثلة في بقية المحاضرة..

إعادة الاستخدام Reusing:

لا خلاف على أن التفكير بالبرمجة بالطريقة غرضية التوجه أصعب من التفكير بالطريقة العادية، ولكنه أفضل بكثير، وذلك لعدة أسباب منها أن الكود يصبح أسهل للقراءة وبالتالي لتصحيح الأخطاء، والأهم من ذلك هو جعل الـــ class قابلاً لإعادة الاستخدام وذلك بطريقتين:

١. الوراثة (Inheritance): وقد سبق شرحها..

٢. التجميع (Composition): وهو تعريف instances من عدة صفوف ضمن class ما..

سنوضح الفرق بين الطريقتين السابقتين عن طريق الأمثلة:

* لنأخذ المثال الشهير *السيارة* والتي تحوي أبواباً ونوافذ ...

فإذا كان لدينا class اسمه (door) وآخر اسمه (window)، وأردنا أن ننشئ class اسمه (car) عندها يجب أن نعرف نسخاً instances من الأبواب والنوافذ ضمن الصف *سيارة*.

ونستطيع القول:

car **has a** door , car **has a** window

نلاحظ أن العلاقة السابقة هي علاقة تجميع (Composition) وهي توصف بالعبارة : **has a**.

* لنفترض أن لدينا شركة تضم مجموعة من الموظفين، ولكن الموظفين في هذه الشركة يقسمون إلى نوعين:

النوع الأول: نظام دوامه نظام طبيعي محدد بساعة دخول وساعة خروج.

النوع الثاني: ليس محدداً بساعات معينة للدخول والخروج وإنما يطلب منه تغطية ساعات عمله متى شاء.

نلاحظ أن كلا النوعين يطلق عليه اسم *موظف*، كما أنهما يشتركان في جميع الخصائص (كالاسم والعنوان و..)، والفرق الوحيد يكمن في تابع حساب الراتب فقط.

الحل هنا هو إنشاء class مجرد –سنشرح معنى كلمة مجرد في محاضرة قادمة– يدعى (Employee) يحوي جميع الخصائص المشتركة، ومن ثم نشتق منه صفين جديدين وهما (Employee1) و (Employee2) ونعيد تعريف تابع حساب الراتب في كل منهما بحسب نوع الدوام.

وبما أن كلى الصفين المشتقين هو عبارة عن نسخة طبق الأصل عن الصف Employee –أي أنهما لا يملكان أي خصائص جديدة– فنستطيع القول:

Employee1 **is a** Employee , Employee2 **is a** Employee

نلاحظ أن العلاقة السابقة هي علاقة وراثة (inheritance) وهي توصف بالعبارة : **is a**.

وكمثال آخر على هذه العلاقة:

circle **is a** shape , rectangle **is a** shape

* هناك نوع آخر من الوراثة توصفه العبارة : **is like a** وسنبينه من خلال المثال التالي:

ليكن لدينا class يدعى آلية (vehicle) ونريد أن نشتق منه الصفين (car) و (bus) ، أي أن الصفوف الجديدة تتمتع بجميع خصائص الصف vehicle ولكنها تملك خصائص جديدة غير موجودة في الأب vehicle عندها نقول:

car **is like a** vehicle , bus **is like a** vehicle

نستنتج أن للوراثة نوعين:

١. التوابع هي نفسها عند الأب والابن ولكن الفرق في بنية بعض التوابع، وتمثله العلاقة: **is a** .

٢. الابن يمتلك خصائص وتوابع جديدة ليست موجودة في الأب، وتمثله العلاقة: **is like a** .

تعددية الأشكال (Polymorphism) :

لنعد إلى مثال الشركة والموظفين، ولنفرض أننا احتجنا إلى تعريف مصفوفة أو سلسلة خطية نضع فيها مؤشرات على جميع الـ object التي تحوي معلومات موظفي الشركة..

عادة لا نستطيع وضع نوعين مختلفين من المؤشرات في نفس السلسلة وبالتالي نحن مضطرون لإفراد سلسلة لكل نوع..

وماذا عن مثال الرسوميات؟ هل سنفرد أيضاً سلسلة لكل شكل من الأشكال؟ ماذا لو أردت أن أضع جميع الأشكال في سلسلة واحدة بحسب ترتيب رسمها على الشاشة؟؟

نلاحظ أننا أمام مشكلة هامة ومنطقية، فما الحل؟؟

الحل يكمن في مفهوم الـ Polymorphism و الذي يتلخص كما يلي:

إذا كان لدي class اسمه **A** و class آخر اسمه **B** مشتق من **A** فإنني أستطيع أن أجعل مؤشراً من النوع **A** يشير إلى Object من النوع **B**..

ففي مثال الشركة : يمكننا تعريف سلسلة عناصرها مؤشرات على الصف Employee وإسناد object من نوع Employee1 أو من نوع Employee2.

بنفس الطريقة يمكننا تعريف سلسلة عناصرها مؤشرات على الصف shape وإسناد object من نوع circle أو من نوع rectangle أو...

إن المؤشر من النوع **A** لن يتيح لك الوصول إلى جميع خصائص الـ **B** object ، ولن تستطيع الوصول إلا إلى الخصائص المشتركة بين **A** و **B**.

لذا تختلف طريقة التعامل مع مفهوم الـ Polymorphism بحسب نوع الوراثة:

- إذا كانت الوراثة من النوع الأول فليس لدينا أي مشكلة إذ أن جميع الخصائص والتوابع مشتركة بين الأب والابن، وبالتالي يكفي تعريف مؤشر على الأب وتمرير أحد الأبناء، وهذا ما يدعى *Up casting*.

مثال: في مثال الرسوميات وبما أن جميع الصفوف تحوي تابع draw فيمكننا كتابة تابع يرسم أي شكل على الشاشة ثم نستدعيه من أجل الشكل المطلوب:

```
void drawShape (shape s){
    .
    .
    s.draw;
    .
    .
}
```

نلاحظ أن التابع يأخذ مؤشراً على shape وهو الـ class الأب، الآن يمكنك تمرير Object من أي نوع من أبناء الصف shape وسيقوم البرنامج باستدعاء تابع draw الخاص بالـ object المدخل *وذلك بسبب الخاصة virtual والتي سنتكلم عنها في المحاضرات القادمة*.

- أما إذا كانت الوراثة من النوع الثاني فإن الابن يحوي خصائص وتوابع إضافية غير التي ورثها من أبيه، فكيف نتمكن من استدعائها؟

الحل يكون باستخدام ما يسمى بـ *Down casting* ، وهو يتم يدوياً بطريقة مشابهة لما اعتدنا عليه في الأنماط الأخرى، والمثال التالي يوضح ذلك:

لنفرض أن الصف **B** يحوي التابع sendMsg والذي لم يرثه من أبيه **A** وإنما عرفه عنده.

فإذا كان لدي مؤشر p من النوع **A** وكان يشير إلى object من النوع **B** فكيف أستدعي التابع sendMsg؟

يجب أن أحول المؤشر إلى النوع **B** عن طريق عملية *Down casting* كالتالي:

```
((B)p).sendMsg();
```

نلاحظ أن عملية *Down casting* عملية يدوية بينما عملية *Up casting* عملية تتم تلقائياً، وسنوضح هذه الأفكار والمفاهيم في المحاضرات القادمة.

<u>هيكلية Java:</u>

تتكون Java كما أسلفنا من مجموعة كبيرة من الـ classes والتي تجمع بينها علاقات وراثة، ولكن جميع هذه الصفوف مشتقة من class أساسي اسمه (Object) وبالتالي يمكن تطبيق مفهوم الـ Polymorphism على جميع صفوف اللغة.

على سبيل المثال : ليكن لدي class ما وأريد أن أترك فيه مكاناً فارغاً سيتم استخدامه لوضع object لن يتضح نوعه إلا خلال عمل البرنامج فما الحل؟؟

يكفي أن أعرف مؤشراً من النوع Object وأضع فيه أي object أشاء فيما بعد، ويمكنني استخدامه عن طريق *Down casting*.

في النهاية أحب أن أذكر بأن هذه المحاضرة مقدمة، وليس ذنبي أن الدكتور تكلم عن المنهاج كله فيها!! لذلك تجاهلت بعض الأبحاث التي ذكر أسماءها دون شرح، ولم يكن لدي مجال للتفصيل في ما ذكرته أكثر من هذا، على أن يتم شرح وتفصيل جميع المعلومات في المحاضرات القادمة..

انتهت المحاضرة . .

lectures_team@hotmail.com