لغات برمجة

الأحد 13/11/2005

السنة الثالثة

المحاضرة السادسة

11

مكتبة المستقبل Future Library هاتف 2119292

مقدمة:

إن المحاضرة التي أعطاها الدكتور يوم الأحد (13/11/2005) كانت طويلة جداً وتتضمن عدة أبحاث من المرجع، لذا لا مجال إلا لتجزئتها على عدة محاضرات.

:Abstract classes and methods

لندرس المثال التالي:

مثال:

```
class Shape
{
	void draw() {}
}

class Circle extends Shape
{
	void draw() {
		System.out.println("Circle");
	}
}

class Rectangle extends Shape
{
	void draw() {
		System.out.println("Rectangle");
	}
}

public class Main
{
	public static void main(String[] args) {
		Shape[] s = {new Circle(), new Rectangle()};
		s[0].draw();
		s[1].draw();
	}
}
```

لنتخيل أن لدينا برنامج رسوميات وأننا سنستخدم الصفوف السابقة فيه، وأن تابع draw للصف Circle يقوم برسم دائرة على الشاشة وكذلك تابع draw للصف Rectangle يقوم برسم مستطيل على الشاشة.

س. ما الذي سيرسمه تابع draw للصف Shape ؟

ج. لن يرسم شيئاً !!

نلاحظ أن التابع draw للصف Shape هو تابع غير واقعي، لأن الصف Shape بحد ذاته صف غير واقعي وإنما هو صف مجرد وظيفته تشكيل مساحة مشتركة بين أبنائه، ونلاحظ كيف استفدنا من هذا الدور عندما عرفنا مصفوفة من Shape ووضعنا فيها Circle و Rectangle.

نستنتج مما سبق أننا لن ننشئ object من الصف Shape أبداً، ولا معنى أصلاً لإنشاء object منه.

أتاحت لنا Java إمكانية تعريف تابع مجرد ضمن صف ما، وذلك بإضافة الكلمة المحجوزة (*abstract*) قبل تعريف التابع، وبما أن هذا التابع مجرد ولن يستخدم مطلقاً إلا لعمل overriding عليه، لذا لا داعي ليكون له جسم تابع أبداً، إذ يمنعنا الــ compiler من تعريف جسم (implementation) له ويسمح لنا بتعريف مــا تبقى منه (return type, method name, arguments)، وسنطبق هذا الكلام مباشــرة علــى التــابع draw للصف object:

```
abstract void draw();
```

وكما لا حظنا في المثال السابق بأن أي class يحوي abstract method منطقياً يجب أن يكون abstract class، وتعرف Java الــ abstract class بأنه صف مجرد لا يمكن أن نعمل object منه، وسنطبق هذا المفهوم على الصف Shape:

```
abstract class Shape
{
	abstract void draw();
}
```

<u>خصائص الــ abstract method:</u>

١. لا يجوز تعريف جسم تابع لأي abstract method.

٢. لا يجوز تعريف abstract method من النوع private لأنه لا يمكــن عمــل overriding للــ private method، وإن abstract method وجدت لنعمل لها override.

<u>خصائص الــ abstract class:</u>

١. لا يمكن إنشاء object منه، وبالتالي فإن فائدته الوحيدة تكمن في الوراثة والــ polymorphism.

٢. لا يشترط أن يحتوي الــ abstract class على abstract method، ولكن بمجرد احتواء أي صــف عادي على abstract method فإن الــ compiler يجبر المبرمج على جعل الصف abstract.

٣. يمكن أن يحتوي الــ abstract class على توابع غير abstract، وبالتالي يكون لها جسم تابع، ويمكن استخدامها في الأبناء مباشرة أو عمل overriding عليها.

٤. كما يمكن أن يحتوي الــ abstract class على حقول بمختلف أنواعها.

٥. يمكن أن يحتوي الــ abstract class على حقول وتوابع static تستدعى من اسم الــ class.

٦. هناك استخدام آخر مهم لمفهوم الـ abstract:

عندما أعرف صفاً جميع توابعه من النوع static يكون دور هذا الصف هو تجميع هذه التوابع لا أكثر، وبما أن التابع الـ static يستدعى عادة من اسم الـ class، فلا داعي لإنشاء object من هذا الـ class أبداً، لذا يعرف عادة (abstract).

:Abstract classes and inheritance

عندما نشتق من الـ abstract class فإن لدينا خيارين يجبرنا الـ compiler على التقيد بأحدهما، وإلا ستكون النتيجة compile-time error وهما:

١. أن نعيد تعريف جميع الـ abstract methods في الصف الابن (overriding).

٢. وإلا فسيجبرنا الـ compiler على جعل الصف الابن أيضاً abstract، وبالتالي عندما يورث منه سيكون للحفيد نفس الخيارين السابقين، وهكذا..

الصف (Object):

ذكرنا مسبقاً أن جميع الـ classes في Java مشتقة من الصف Object، وهذا ما يسمى اصطلاحاً بـ:

(The singly rooted hierarchy)

إن التوابع الموجودة في هذا الـ class لم توضع عبثاً، ولكنها مختارة بعناية بحيث يكون من المنطقي احتواء كل class على نسخته الخاصة منها (Overriding)، وسنتكلم هنا عن بعض هذه التوابع:

- toString(): مر معنا مسبقاً، ونعيد تعريفه في أي صف كما يلي:

```
public String toString() {...}
```

- finalize(): أيضاً مر معنا مسبقاً، ونعيد تعريفه في أي صف كما يلي:

```
public void finalize() {...}
```

- equals(): وهو التابع الذي نستطيع عن طريقه المقارنة بين غرضين من نفس النوع، إذ أن المقارنة عن طريق العملية (==) إنما هي مقارنة مؤشرات، ويعاد تعريفه كما يلي:

```
public boolean equals(Object obj) {...}
```

قد يتطرق إلى الذهن أن الصف الأساسي Object هو أكثر صف مجرد في Java فلماذا لم تعرفه Java على أنه abstract؟؟

لم تفعل Java هذا لكي لا تجبر المبرمج على إعادة تعريف جميع التوابع في هذا الصف لأن هذا قد يكون فيه مشقة..

مكتبة المستقبل Future Library هاتف 2119292

:Interfaces

هي مفهوم شبيه بالــ abstract class ولكنها أكثر تجريداً منه.

في abstract class كنا نستطيع تعريف توابع غير abstract، كما كنا نستطيع تعريف حقول من أي نـــوع لكن الأمر تغير هنا، وهذا ما سيتضح لنا من خلال:

خصائص الــ Interfaces:

١. يمكن أن نقول أن الــ Interface عبارة عن class ولكن بمواصفات خاصة.

٢. نعرف الــ Interface كما نعرف الــ class ولكن باستبدال الكلمة (*class*) بالكلمة (*interface*).

٣. كما في الــ class: يمكن أن تكون الــ Interface من أحد النوعين:

a. <u>public</u>: ترى في جميع الــ packages عندما نعمل لها import.

b. <u>package access</u>: لا ترى إلا في نفس الــ package.

٤. جميع التوابع في الــ Interface هي (abstract methods) أي لا يوجد لها Body، أي لا يوجد أي implementation لأي تابع في الــ Interface.

٥. أي تابع في الــ Interface هو حتماً public، لأن الــ compiler يمنعك من تعريفه private أو protected وحتى لو عرفته (package access) فسيُعامَل وكأنه public، والحكمة من ذلك أنـــه وضع في الــ Interface ليحققه جميع الأبناء، وبالتالي نحن نضمن أن جميع الصفوف التي تحقـــق الــ Interface حتماً تحقق جميعها كل توابع الــ Interface.

٦. جميع الحقول في الــ Interface حتماً تكون static و final و public، أي أنها تأخذ قيمتها مـــرة واحدة ولا تسمح بتغييرها كما أنها تتعلق باسم الــ Interface ولا يوجد نسخة منها في كل غـــرض من الصفوف التي تحقق الــ Interface.

٧. هذه الحقول لا تأخذ قيماً ابتدائية، وإنما أنت مضطر حتماً لعمل (initialization) لجميع الحقول.

٨. لا يمكن أن نعرف object من الــ Interface وإنما تستخدم في الوراثة والــ polymorphism، ونقول عن الصف الذي يرث من الــ Interface بأنه (يحقق الــ Interface).

٩. تتم الوراثة من الــ Interface بطريقة مشابهة للــ class ولكن باستبدال الكلمة (*extends*) بالكلمة (*implements*).

الوراثة المتعددة:

لنفرض أن لدينا class دكتور في الجامعة (Professor)، إن هذا الدكتور هو عالم لذا يجب أن يـــرث مـــن الصف (Scientist)، كما أنه موظف في الجامعة لذا يجب أن يرث من الصف (Employee).

- في ++C كنا نستطيع أن نرث من الصفين في نفس الوقت، لكن Java لا تسمح بهذا أبداً وتحصر الوراثة بأب واحد فقط.

- الحكمة من ذلك تتضح عند وجود حقول أو توابع مشتركة بين الآباء، عندها سيحدث تضارب عند الابــن، ولكن فكرة الوراثة المتعددة هامة ومفيدة فما الحل؟
- الحل كان عن طريق مفهوم الــ Interface الذي يحل تماماً مشاكل الوراثة المتعددة، إذ يسمح للــ class بأن يحقق عدداً غير منته من الــ Interfaces وذلك عن طريق وضــع أســمائها جميعــاً بعــد الكلمــة (*implements*) ووضع فواصل عادية بينها، أي أن بإمكاننا الوراثة من class واحــد وتحقيــق عــدة Interfaces.
- بالطبع لابد من إعادة تعريف جميع توابع الــ Interface في كل أبنائها.
- وبما أن الــ Interfaces لا تحوي implementation لأي تابع فلن نرى مشاكل الــ C++ في الــ Java وهذا ما سنوضحه جيداً بعد قليل.
- الفائدة الحقيقة من الــ Interface هي إمكانية عمل (Upcasting) و (Downcasting) إلــى عــدة أنماط جديدة وليس فقط إلى صف واحد، وستتضح هذه الفكرة من خلال الأمثلة.

<u>مثال:</u>

```
interface CanEat {
	void eat();
}

interface CanStudy {
	void study();
}

interface CanWalk {
	void walk();
}

class Person
{
	public void walk() { System.out.println("Person can walk"); }
}

class Student extends Person implements CanEat, CanStudy, CanWalk
{
	public void eat() { System.out.println("Student can eat"); }
	public void study() { System.out.println("Student can study"); }
}

public class Main
{
	public static void main(String[] args) {
		Student s = new Student();
		s.eat();
		s.study();
		s.walk();
	}
}
```

لندرس المثال السابق:

- نلاحظ أن الـ methods ضمن الـ Interfaces ليس لها (method body).
- كما أنها تعرف (public) بشكل طبيعي حتى ولو لم نعرفها كذلك *أي حتى ولو لم نضع كلمة public قبلها*.
- لو أن الصف Student لم يعد تعريف جميع التوابع التي ورثها من الـ Interfaces، فإن الـ compiler سيحتج!
- إن وراثة التابع (walk()) من الأب الصف الأب (Person) أغنت عن إعادة تعريفه ونالت رضا الـ compiler إذ أنه اعتبر التابع walk() وكأنه عُرِّف في الصف Student، ولكن هذا لا يمنع من عمل override للتابع walk() ضمن الصف Student.

:Name collisions

لنفرض أن عدة Interfaces تشاركت في نفس الحقول أو في نفس التوابع، ما الذي سيحصل؟

- في C++ كان ينتج لدينا تضارب ولن نستطيع التمييز بين هذه المتشابهات إلا عن طريق العملية (::).
- أما في Java فنميز بين حالتين:

١. تضارب في الحقول: بما أن هذه الحقول حتماً static فيمكن عند حدوث تضارب طلب هذه الحقول من اسم الـ Interface مباشرة.

مثال:

```
interface A { int i = 2; }
interface B { int i = 7; }
```

يمكننا الآن أن نكتب: (A.i) و (B.i).

٢. تضارب في التوابع: وهنا تتميز Java بفكرة استخدام الـ Interface بدل الوراثة المتعددة:

- لندرس المثال التالي ونوضح الأفكار من خلاله:

مثال:

```
interface I1 { int f(); }
interface I2 { int f(); }

class A
{
      public int f() { return 1; }
}

class B extends A implements I1, I2
{
      public int f() { return 2; }
}
```

مكتبة المستقبل Future Library هاتف 2119292

نلاحظ تكرار وجود التابع في كل من الـ Interface (I1, I2) وفي الصف A، ألن يحدث تضارب؟

في الحقيقة لن يحصل أي تضارب، لأن التضارب يحدث عادة في C++ عند عمل (override) للتابع الموجود في أكثر من أب وله (implementation) مختلفة في كل أب، وبالتالي لن يميز الـ

compiler بين التوابع، أي أنه لا يعلم إلى أي أب من الآباء ستوصله تعليمة (super) عندما يكــون لدى جميع الآباء نفس اسم التابع!!

أما عند الانتقال إلى مفهوم الــ Interface فإن التوابــع بــلا (implementation)، أي أن الـــ Interface لا تهتم إلا بإجبار الصف الذي سيحققها على تعريف تابع يحمل نفس اسم التابع المعــرف فيها، ولن نقوم أصلاً باستدعاء هذا التابع من الــ Interface.

نتيجة: إن وجود f() في الصف B مرة واحدة فقط أرضى طموح جميع الــ Interfaces التي يحققها، وإن أي استدعاء من الشكل (super.f()) إنما يعني التابع المعرف في الـــ (base class) الوحيــد للصف B حصراً *المقصود هو الصف A*.

وسنرى بعد قليل أن الهدف في النهاية من كل هذه العملية هو القيام بعملية الــ polymorphism.

- لندرس الآن حالة هذا المثال:

مثال:

```
interface I1 { int f(int i); }
interface I2 { int f(); }

class B implements I1, I2
{
      public int f() { return 0; }
      public int f(int i) { return 1; } // overloaded
}
```

في المثال السابق نلاحظ أن (signature) التابع اختلف بين الــ Interfaces وبالتالي نحن مضطرون لإعادة تعريف كل منهما وهذا يسمى (overload).

- لدينا حالة أخرى هامة:

مثال:

```
interface I1 { int f(); }
interface I2 { void f(); }

class B implements I1, I2 {}
```

في المثال السابق سيحتج الــ compiler، إذ لا توجد طريقة تميز التابعين عن بعضهما عند الصف B.
لتتضح الصورة أكثر: لنفترض أننا أعدنا تعريف كل من التابعين في الصف B، عندها سيصبح لــدينا تابعان لهما نفس الاسم ونفس الــ arguments ولا يختلفان إلا بالــ return type، وهــذه حالــة مطابقة لعمل (overload) لتابعين عن طريق الــ return type، وقد ذكرنـــا مـــسبقاً أن Java لا تسمح بذلك.

مكتبة المستقبل Future Library هاتف 2119292

:Interface and inheritance

يمكن لأي Interface أن ترث من أي عدد من الـــ Interfaces الأخر وذلك ببساطة شديدة عـــن طريـــق التعليمة (*extends*) وذلك كما في المثال التالي:

مثال:

```
interface A {
      int a();
}

interface B {
      int b();
}

interface C extends A, B {
      int c();
}

class ss implements C
{
      public int a() {}
      public int b() {}
      public int c() {}
}
```

نلاحظ أن علاقة الوراثة بين الـــ Interfaces جعلت توابع A, B موجودة في C، وبالتـــالي فـــإن أي class سيحقق الـــ Interface (C) يجب أن يعيد تعريفها جميعاً.

:Interface and polymorphism

في الحقيقة هنا تكمن الفائدة الحقيقية لمفهوم الوراثة المتعددة، إذ أن الـــ Interface تـــؤمن واجهـــة تخاطـــب مشتركة بين الصفوف التي تمثلها، وبالتالي يمكن استخدام مفهومي الـــ (Upcasting) و (Downcasting) مع هذه الصفوف.

سنطرح مثالاً عن Interface موجودة فعلاً في الـــ Java وتدعى (Comparable):

نحن نعلم أن (Operator overloading) ممنوع في Java، وبالتالي لا يمكن إيجاد طريقة لمقارنة غرضين عن طريق العمليات المعتادة (< ، >) إذ أن هذه المقارنة هي دوماً مقارنة مؤشرات (references)، لذا لابـــد من مقارنة الأغراض عن طريق كتابة تابع ما وهو التابع (compareTo()).

نستنتج من هذا الكلام أن علينا كتابة تابع compareTo() لكل class على حدة، ولكننا نخسر بهذه الطريقـــة جميع ميـــزات الـــــ polymorphism، والأفـــضل مـــن هـــذا هـــو إنـــشاء Interface تحـــوي التـــابع compareTo()، وأجعل جميع الصفوف القابلة للمقارنة *Compare Able* تحقق هذه الـــ Interface وبالتالي نستفيد من خصائص الـــ polymorphism.

لنأخذ مثالاً عملياً ونرى كيف سنستخدم الـــ Interface (Comparable) فيه:

لنفرض أننا نريد كتابة إجرائية فرز لمصفوفة ما.

في البرمجة الإجرائية نحن مضطرون لكتابة إجرائية فرز خاصة لكل نمط من أنماط المصفوفات، أي أنه سيكون لدينا عدد هائل من الإجرائيات كل منها مختص بفرز مصفوفة من نمط معين.

يفترض أننا أصبحنا ندرك أن مثل هذا الحل مرفوض رفضاً تاماً في البرمجة غرضية التوجه.

أي أننا يجب أن نستفيد من تقنيات الـ polymorphism لكتابة إجرائية واحدة تقوم بفرز أي نوع من أنواع المصفوفات.

لنأخذ عينة مكونة من ٣ صفوف ونطبق جميع الأمثلة عليها:

الصف (Integer) والصف (String) والصف (Person) المعرف كما يلي:

```
class Person
{
      private String name;
      .
      .
      .
}
```

لتستطيع إجرائية الفرز أن تتعامل مع مصفوفة من أحد الأنواع السابقة *وحسب الـ polymorphism* يجب أن يكون دخل هذه الإجرائية عبارة عن مصفوفة من نمط أب لجميع الأنماط السابقة حتى تتم عملية الـ (Upcasting)، وإن أول ما يتبادر إلى الذهن هو كون دخل الإجرائية مصفوفة من Object وهو كما نعلم النمط الأب لجميع الصفوف.

```
public void sort(Object[] x) {}
```

كيف ستعمل الإجرائية؟

إن على الإجرائية أن تستدعي التابع compareTo() لتقارن مثلاً العنصر (x[i]) مع العنصر (x[i+1])، ولكن الصف Object لا يحوي التابع compareTo() وبالتالي لا نستطيع استدعاءه كما يلي:

```
x[i].compareTo(x[i+1]);
```

فما الحل؟؟

الحل يكون بالبحث عن **نمط ما** يكون أباً لجميع تلك الصفوف ويحوي التابع compareTo() ليكون نمطاً لعناصر هذه المصفوفة، والنمط الوحيد الذي يمكن أن يحقق هذه الصفة هو الـ Interface التي تدعى (Comparable).

ولكن هل تتشكل هذه الـ Interface أباً لجميع الصفوف المطلوبة؟

الجواب: إن جميع صفوف Java التي تحوي التابع compareTo() إنما هي محققة للـ Interface (Comparable) بما فيها الصفان (Integer, String)، ولم يبق علينا إلا أن نجعل الصف (Person) يخضع لنفس القاعدة:

<u>**مثال:**</u>

مكتبة المستقبل Future Library هاتف 2119292

```
class Person implements Comparable
{
      private String name;

      public int compareTo(Object o) {
            String myName = this.name;
            String anotherName = ((Person)o).name;
            return myName.compareTo(anotherName);
      }
}
```

نلاحظ من المثال السابق ما يلي:

إن دخل الإجرائية compareTo() كان من الـنمط Object، ونحن مجبرون على ذلك لأن التابع compareTo() معرف بهذه الصيغة في الـ Interface (Comparable) وهذه العملية هي في النهاية عملية (Override)، وبالتالي عندما رغبنا بالتعامل مع المؤشر من النوع Object على أنه من النوع Person قمنا بعمل (Downcasting) له إلى النمط Person *إذا أدخل المستخدم إلى الإجرائية أي صف غير Person فسيظهر Exception وهو موضوع المحاضرة بعد القادمة*.

كما نلاحظ أننا قارنا بين الاسمين عن طريق تابع compareTo() المعرف مسبقاً للنمط String، ولم نعد اختراع العجلة *كما يقولون*!!

يمكننا الآن التعامل مع إجرائية الـ (sort) بكل أريحية، إذ أن بإمكاننا أن نكتب:

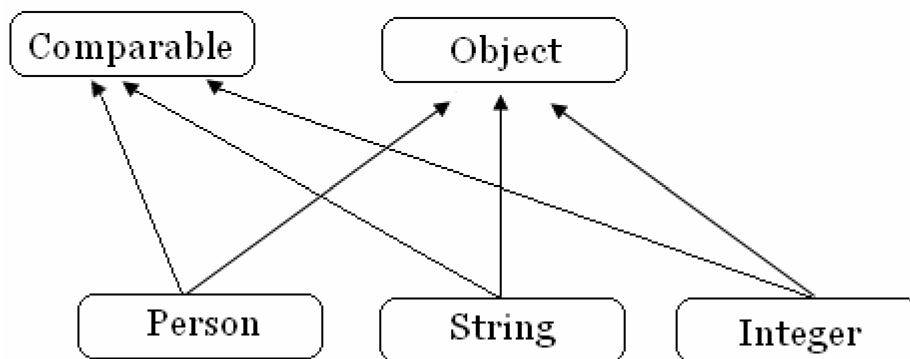
```
public void sort(Comparable[] x) {
      x[i].compareTo(x[i+1]);       // now I can compare between the array elements
}
```

وعلى فرض أن دخل التابع كان مصفوفة Object فيمكننا أن نعمل (Casting) من النمط Object إلى النمط Comparable كما يلي:

```
public void sort(Object[] x) {
      ((Comparable)x[i]).compareTo(x[i+1]);
}
```

ملاحظة:

من الواضح أنني لم أكتب نص خوارزمية الفرز وإنما وضحت كيف نقارن بين عناصر المصفوفة فقط، وإن الشكل التالي يوضح العلاقة السابقة تماماً:



:Grouping constants

يمكن أن تستخدم الــ Interface كنمط مجمع للمتحولات الثابتة *يشبه مفهــوم الــــ enum فــي ++C*، والمثال المطروح في المرجع واضح ولا داعي لإعادة كتابته في المحاضرة.

ملاحظة:

أعود وأنبه زملائي الأعزاء إلى ضرورة الرجوع إلى أمثلة المرجع، ربما تغطي المحاضرة جميع المعلومــات في المرجع ولكنها لن تغطي جميع الأمثلة حتماً *لأنها ستصل إلى ما يقرب من ٣٠ صفحة!!*، لذا لا غنى عن أمثلة المرجع لمن يريد أن يحيط بالأفكار إحاطة تامة.

انتهت المحاضرة ..

lectures_team@hotmail.com