15

المحاضرة الثالثة

مقدمة:

كان هناك عدة تعقيبات من الزملاء *مشكورين* على المحاضرة السابقة:

١. كنا قد أشرنا إلى أن الصف الذي يحوي main method يجب أن يكون من النوع public، وهذا الكلام لم يكن دقيقاً إذ أنه يجوز أن يكون (default).

٢. في آخر المحاضرة السابقة عندما تحدثنا عن الـ JDK كان هناك أخطاء إملائية في اسم الملفات وذلك بالنسبة لحالة الأحرف، وصوابها جميعا هو: MyClass.

في Java لدينا نوعان من الأخطاء:

١. Compile Error: وهي الأخطاء التي يكتشفها الـ (Compiler) قبل البدء بتنفيذ البرنامج، ومنها الأخطاء القواعدية..

٢. Runtime Error: وهي الأخطاء التي لا تكتشف إلا خلال تنفيذ البرنامج، كالتقسيم على صفر مثلاً، وتكون ردة فعل الـ JVM تجاهها هي توليد اعتراض (Exception) وهو ما سنتكلم عليه لاحقاً إن لم يدركنا الوقت في نهاية الفصل..

الحلقات:

هناك بعض النقاط المهمة في الحلقات في Java، سنعرج عليها سريعاً:

ما الخطأ في المثال التالي؟

```
int i = 10;
while (i>0);
      i--;
```

الخطأ يكمن في وضع فاصلة منقوطة بعد الشرط وهذا ما جعل التعليمة التالية واقعة خارج الحلقة وبالتالي فهي لا تنفذ في كل دورة من دورات الحلقة، وبالتالي ستبقى قيمة (i) أكبر من الصفر ولن يكسر شرط الحلقة أبداً، وسنحصل على حلقة غير منتهية..

ما هي الحلقة غير المنتهية؟ هي حلقة ليس لها شرط توقف:

```
for (;;)
```

أو حلقة لها شرط توقف ولكنه لا يتحقق أبداً:

```
while (true)
```

يمكن تغيير مسار الحلقة عبر تعليمتين:

continue: تقوم بإنهاء اللفة الحالية للحلقة والبدء باللفة التالية مباشرة.

break: تخرج من الحلقة نهائياً.

لنمعن النظر في المثال التالي:

```
label1:
for (int i=0; i<10; i++)
{
	for (int j=0; j<10; j++)
	{
		continue label1;

		break label1;
	}
}
```


نلاحظ أننا استخدمنا التعليمتين السابقتين ولكن بإضافة label، وهنا تقوم كل من التعليمتين بإعادة التنفيذ إلى الـ label المناسب، مع الانتباه إلى الفرق بين عمل كل منهما، حيث تقوم continue بإعادة التنفيذ إلى الـ label والدخول في الحلقة مرة أخرى ولكن في اللفة الجديدة، أما break فتنهي الحلقة تماماً.

Initialization:

تحدثنا في المحاضرة السابقة عن إعطاء القيم الابتدائية للمتحولات، وفرقنا بين حالتين:

١. المتحولات ضمن الـ methods: لا تأخذ قيماً ابتدائية من تلقاء نفسها وإنما يجب إعطائها قيماً ابتدائية يدوياً قبل استخدامها، وإن استخدامها دون (Initialization) سيؤدي إلى Compile Error.

٢. حقول الـ class: تأخذ قيم ابتدائية من تلقاء نفسها، فإذا كانت primitive أخذت قيمها من الجدول الذي وضحناه في المحاضرة السابقة وإذا كانت reference فإن قيمتها تكون (null).

ولكن ماذا لو أردنا أن نعطيها القيم بأنفسنا؟

يمكننا إعطاء قيم ابتدائية للحقول بعد تعريفها مباشرة *حتى ولو كانت هذه الحقول references* كما يلي:

```
class Test {
	int i = 5;
	Object obj = new Object();
}
```

أو أن نستخدم الـ constructor لهذا الغرض كما سنرى بعد قليل.

ملاحظة:

عندما نريد أن نعطي قيمة ذات فاصلة عشرية لمتحول أو حقل من النوع float سيعطينا المترجم خطأ ترجمة كما في المثال التالي:

```
float f = 5.8;                  // compiler error
float f = 5.8f;                 // no problem
```

نلاحظ أن حل المشكلة كان عن طريق إضافة الحرف (f) إلى نهاية العدد وذلك ليفهم الـ compiler أن هذه القيمة من النوع float وليست من النوع double.

لدينا حالة هامة وهي حالة المتحولات من النوع static:

نعلم أن هذه المتحولات تتعلق بالـ class وليس بالـ objects، وهذا يعني أنها تعرف مرة واحدة وبالتالي تعطى قيماً ابتدائية مرة واحدة فقط، ولكن هذا لا يتم إلا عندما يصبح ضرورياً، أي أن إعطاء القيم الابتدائية للحقول الـ static لا يحدث في أول البرنامج مرة واحدة، ولكن يحدث عندما يتعامل الـ interpreter مع الـ class لأول مرة، أي عند القيام بأحد الأعمال التالية:

١. تعريف object من الصف لأول مرة في البرنامج.
٢. محاولة استخدام حقل أو تابع static من الصف لأول مرة.

هناك طريقة خاصة تستخدم لـ (Initialization) المتحولات الـ static وتعريف متحولات جديدة بدون كتابة كلمة static كل مرة، وتكون بكتابة كلمة static وفتح scope بعدها مباشرة، كما في المثال التالي:

```
class Test {
      static int i;
      static {
            i = 1;
            double d = 9.5;
            String s = "Ammar";
      }
}
```


ماذا حدث في المثال السابق؟

ما حدث هو أننا عرفنا متحول static وهو (i) وأعطيناه القيمة الابتدائية (١)، كما عرفنا المتحولات (d, s) أيضاً static وأعطيناها القيم الابتدائية ("Ammar" , 9.5) على الترتيب.

وبالتالي فإن خطوات ترجمة أي class *بما فيه الـ class الذي يحوي main method* تكون كالتالي:

١. عمل (Initialization) لكل المتحولات الـ static *مهما كان موقعها ضمن الـ class*.
٢. ثم عمل (Initialization) لكل المتحولات الـ instance *مهما كان موقعها ضمن الـ class*.
٣. ثم تنفيذ الـ constructor.

وسنوضح هذا الكلام بعد أن نتكلم عن الـ constructor.

Method Overloading:

لنفرض أن لدينا صف ما فيه تابع طباعة print على سبيل المثال، وصادف أن الطباعة ليست لنوع واحد من المتحولات، وإنما يجب أن يكون لدينا تابع طباعة يختلف بحسب نوع المتحول المدخل إلى هذا التابع، إذاً علينا أن نكتب عدة توابع ونسميها كالتالي: (intPrint, floatPrint, charPrint ...).

ولكن هذا مزعج قليلاً، لذا سنفكر مباشرة بأنه من الأفضل لو كان اسم جميع التوابع print فقط، ولكن هذا سيؤدي إلى تضارب، فما الحل؟

الحل يكمن في تقنية Method Overloading حيث تسمح لنا Java بتسمية عدة methods بنفس الاسم.

السؤال الذي يطرح نفسه: كيف سيفرق المبرمج والـ compiler بين هذه التوابع؟

يكون التفريق عبر اختلاف الـ arguments حيث لا يجوز أن نكتب تابعين لهما نفس الاسم ونفس الـ arguments وإنما يجب أن يكون هناك اختلاف في الـ arguments، حتى ولو كان الخلاف فقط في ترتيب هذه الـ arguments.

مثال:

```
class Test {
	static void print(byte b) {
		System.out.println("I am byte printer");
	}

	static void print(int i) {
		System.out.println("I am int printer");
	}

	static void print(double d) {
		System.out.println("I am double printer");
	}

	public static void main (String[] args) {	// main method
		Test.print(5);			// int value
		Test.print(5.0);		// double value
	}
}
```

سيكون خرج البرنامج السابق:

```
I am int printer
I am double printer
```

نلاحظ من الخرج السابق أن الـ compiler عرف التابع المقصود من نوع الدخل المدخل، ولكن لماذا اختار التابع الذي يأخذ int ولم يختر التابع الذي يأخذ byte عندما أدخلنا 5 ؟

الجواب هو لأن جميع الأعداد الصحيحة وجميع العمليات عليها إنما تكون من النوع int، وكذلك الأعداد الحقيقية والعمليات عليها تكون من النوع double، ولو أردنا استدعاء التابع الذي يأخذ byte لكان علينا أن نحول العدد ٥ إلى النوع byte يدوياً عن طريق Down casting كما يلي:

```
Test.print((byte)5);
```

لتتضح هذه الفكرة جيداً لابد من مثال، ولكن المثال الذي طرحه الدكتور كبير، وهو موضح بحذافيره في المرجع في البحث ٤ تحت العنوان (Overloading with primitives)، لذا أنصح الجميع بالعودة للمرجع ومطالعة المثال.

هناك نقطة أخيرة قد يُتَوقف عندها:

ذكرنا أن التفريق بين التوابع إنما يكون عبر الـ arguments، ألا نستطيع أن نفرق بين التوابع عن طريق الـ return type؟

```
int func () {}
void func() {}
```

للوهلة الأولى قد لا نجد أي مشكلة فإن استدعاء التابع الأول وإسناده إلى متحول من النوع int سيفرقه عن التابع الثاني الذي لا يسند لأي متحول، ولكن إذا انتبهنا أكثر سنجد أن هذا مستحيل، فما الذي يمنعني من استدعاء التابع الأول وعدم إسناده إلى أي متحول؟؟

نستنتج أن الفرق بين التوابع التي عملنا عليها Overloading إنما يكون بالـ arguments حصراً.

الباني (Constructor):

غالباً ما تكون قيم الحقول غير معروفة للمبرمج عند كتابة البرنامج وإنما تعرف عند التنفيذ، وهذا يعني أننا بحاجة لاستدعاء التوابع المناسبة لضبط قيم هذه الحقول (setName, setAge...) بعد إنشاء object جديد، ولكن ألا نستطيع أن نضبط هذه القيم عند إنشاء الـ object مباشرة؟

هنا تظهر أهمية وجود method موجود في كل class ويستعدى عند إنشاء object بشكل تلقائي، هذا الـ method يدعى: الباني (Constructor).

ما هي خصائص الباني؟

١. اسمه نفس اسم الـ class الذي يحويه.
٢. يستدعى مباشرة عند إنشاء object.
٣. لا يرد أي قيمة وبالتالي ليس له return type.
٤. يمكن أن يحوي الصف الواحد أكثر من باني واحد (Overloading).

مثال(I):

```
class Student
{
	int age;
	String name;

	Student () {                                    // first constructor
		age  = 0;
		name = "";
	}
```

```
	Student (int age, String name) {		// second constructor
		this.age  = age;
		this.name = name;
	}

	public static void main (String[] args) {	// main method
		Student s1 = new Student();
		Student s2 = new Student(20, "Ammar");
	}
}
```

نلاحظ وجود بانيين للـــ class السابق أحدهما لا يأخذ أي arguments والآخر يأخذ اسم الطالب وعمره.
كما نلاحظ طريقة استدعاء كل منهما *عبر الـــ main method* وذلك بعد تعليمة new مباشـــرة، أي أن العملية أصبحت تشبه استدعاء تابع اسمه ()Student، ولدينا عدة نسخ منه (Overloading).

Default constructor:

ماذا لو لم نضع أي بانٍ في الـــ class؟ ربما تقول لي أين المشكلة؟ ولكن لننتبه إلى أن تعريف object جديد من الصف يتضمن استدعاء للباني بشكل أساسي وذلك عن طريق القوسين:

```
Student s = new Student();
```

حلت Java هذه المشكلة عن طريق استدعاء بانٍ افتراضي وهو بانٍ لا يأخذ أي arguments ولا ينفـــذ أي تعليمة (تابع فارغ)، أي أن بإمكاننا إنشاء object من الصف كالتالي وبدون أي مشاكل:

```
Student s = new Student();
```

هذا يعني أننا عندما نعرف بانياً لا يأخذ أي argument، سنكون قد عملنا Overload للبـــاني الافتراضـــي وبالتالي سيستدعى الباني الذي كتبناه، **وسيصبح هو الباني الافتراضي**.

ملاحظة: عندما نعرف أي بانٍ يأخذ arguments بدون أن يكون لدينا بانٍ لا يأخذ أي arguments، نكون قد عطلنا عمل الباني الافتراضي، وبالتالي لن نستطيع إنشاء object من الـــ class بدون تمرير متحولات.

مثال:

- لنعد إلى المثال(I) ولنفرض أن كلاً من الباني الأول والثاني لم يكونا موجـــودين، هـــذا يعنـــي أن إنـــشاء object من الصف Student سيؤدي إلى استدعاء الباني الافتراضي.

```
Student s = new Student();		// call default constructor
```

- لنفرض أن الباني الأول موجود والثاني غير موجود، هذا يعني أن إنشاء object من الـــصف Student سيؤدي إلى استدعاء الباني الأول وليس الباني الافتراضي الذي تنشئه Java.

```
Student s = new Student();		// call my default constructor
```

- لنفرض أن الباني الثاني موجود والأول غير موجود، هذا يعني أن إنشاء object من الـــصف Student بدون تمرير arguments سيؤدي إلى خطأ في المترجم، وذلك لأن الـــ class لـــم يعـــد يحـــوي بانيـــاً افتراضياً.

```
Student s = new Student();		// Compile Error!
```

وستصبح الطريقة الوحيدة لتعريف object جديد هي :

```
Student s = new Student(20, "Ammar");
```

مراحل الترجمة:

ذكرنا أن مراحل ترجمة أي class تبدأ بإعطاء القيم الابتدائية (Initialization) للحقول الـ static، وهـذا يتم مرة واحدة فقط لأنه لا يوجد إلا نسخة واحدة من هذه الحقول تتعلق بالـ class، ولكن ما لا يتم مرة واحدة وإنما كلما عرفنا object جديد هو ما يلي:

١. Initialization the instance members.

٢. Call constructor: وهو يتم مباشرة بعد الخطوة السابقة، ومن المفروض أن يكون إعطـاء القـيم الابتدائية للمتحولات الـ instance هو مهمة الباني لأن هذا يعطي مرونة أكبر للمبرمج إذ أنه يستطيع إدخال هذه القيم من خارج الـ class وبالتالي تكون قيم متغيرة.

لنفرض أنني أريد أن أعرف object من الصف Student فليس من المنطقي أن أعرف اسمه وعمره وقت البرمجة وإنما تعرف هذه التفاصيل أثناء تنفيذ البرنامج، لذا يجب أن تعطى القيم الابتدائية لها أثناء التنفيذ عن طريق الباني.

كنت أريد أن أضع مثالاً يوضح تسلسل ترجمة وتنفيذ الـ class، ولكنني لاحظت أن الأمثلة المطروحة في المرجع في البحث (٤) واضحة ومفهومة، لذا أرجو من الجميع العودة للمرجع وفهم الفقرة وأمثلتها جيداً لأن هذه الفقرة أهم فقرة بالنسبة لأسئلة الفحص، وأغلب العلامات تضيع في أسئلة الـ (Initialization).

الكلمة المحجوزة (this):

تستخدم هذه الكلمة بطريقتين مختلفتين وهامتين جداً، وهما:

١. **reference:** تستخدم this كمؤشر على الـ object الذي نحن فيه حالياً، ويكـون هـذا الاسـتخدام حصراً ضمن توابع الـ class.

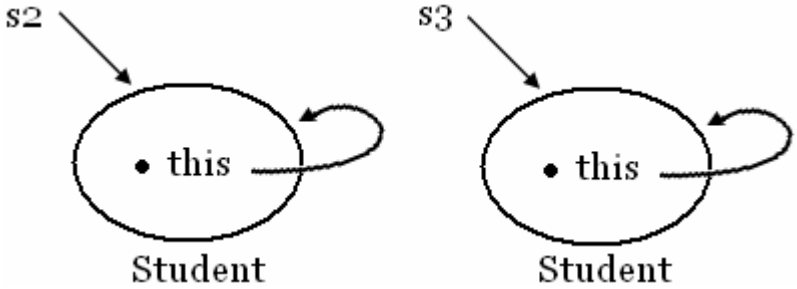
مثال:

لنعد إلى المثال(I):

في الباني الأول نلاحظ أننا استخدمنا أسماء الحقول (name, age) مباشرة دون أي زيادات.

أما في الباني الثاني فنلاحظ أننا وصلنا لنفس الحقول (name, age) ولكن عن طريـق المؤشـر this، والطريقتان متكافئان بشكل عام، إلا أن المثال السابق فيه حالة خاصة سنتكلم عنها.

نلاحظ في الـ (main method) أننا أنشأنا object يؤشر عليه المؤشر s2 واستدعينا الباني الثاني، وهذا يعني أن this في هذه الحالة تشير إلى نفس الـ object الذي يشير إليه s2، ولو أنشأنا object يؤشر عليه المؤشر s3 لكانت this تشير إلى نفس الـ object الذي يشير إليه s2.



٢. **اسم الباني:** ليكن لدينا class ما وفيه بانيان، ولكن هذين البانيين ليسا مستقلين عن بعضهما، وإنما يقوم الباني الثاني بنفس أعمال الباني الأول بالإضافة إلى أعمال أخرى، من الطبيعي أننا لن نفكر بإعادة كتابة نفس تعليمات الباني الأول في الباني الثاني، وإنما سنستفيد من الباني الأول ونستدعيه، ولكن كيف؟

هنا يبرز الدور الجديد لـ this، حيث تعمل كاسم للبواني ولكن بشروط:

أ. أن يكون هذا الاستدعاء ضمن البواني فقط، ولا يسمح ضمن أي method أخرى أبداً.

ب. أن يكون الاستدعاء وحيداً.

ت. أن يكون أول تعليمة في الباني حصراً.

مثال(II):

```
class Student
{
	int age;
	String name;

	Student (String name) {                          // first constructor
		this.name = name;
		.
		.
		.
	}

	Student (String name, int age) {                 // second constructor
		this(name);                                  // call first constructor
		this.age = age;
	}
}
```

نلاحظ أن الباني الأول يضبط قيمة حقل الـ name، والباني الثاني يضبط قيمة كل من الحقلين name , age فهما يشتركان في جزء من مهماتهما وهو ضبط قيمة الحقل name، فلم لا نقوم باستدعاء الباني الأول من الباني الثاني ونوفر إعادة كتابة نفس التعليمات؟

يتم ذلك كما اتفقنا عن طريق التعليمة this والتي أصبحت تمثل اسم الباني، ونميز الباني المطلوب استدعاؤه عن طريق الـ arguments، مع الانتباه إلى الشروط السابقة التي ذكرناها.

لدينا حالات يكون استخدام الكلمة this فيها إجبارياً:

١. عندما أمرر متحول للـ method ويكون اسم هذا المتحول يطابق اسم أحد حقول الـ class، عندها سينتج لدينا تضارب في الأسماء، حلت Java المشكلة بجعل اسم المتحول يدل على المتحول المدخل، ويكون الوصول للحقل الذي يحمل نفس الاسم عن طريق المؤشر this حصراً. *عد للمثال(I) وانتبه للباني الثاني*.

٢. عند استدعاء method تأخذ مؤشراً على الـ object الذي أنا فيه:

سنوضح الفكرة السابقة عن طريق **المثال(III):**

```
class College
{
    void addStudent (Student c)
    {
        // Add to my list..
    }
}
```

```
class Student
{
    void goToCollege(College c)
    {
        c.addStudent(this);
    }
}
```

نلاحظ أن التابع (addStudent) يأخذ reference من النوع (Student)، وقد أحتاج ضمن أحد الـ objects من النوع (Student) *كما في التابع (goToCollege)* إلى تمرير مؤشر على نفسي، عندها يكون الحل الوحيد هو استخدام الكلمة المحجوزة this.

٣. عندما أريد استدعاء بانٍ آخر: وقد فصلنا في شرحها قبل قليل.

ملاحظات:

١. عند استدعاء أي method، مثلاً :(c.addStudent)، فإن المؤشر this يمر مع الـ arguments ليعلم الـ compiler أي object هو الذي استدعى هذا الـ method، ويسمى هذا الـ object بـ **(current object)**.

٢. لا نستطيع التعامل مع المؤشر this ضمن التوابع الـ static: وهنا يتضح مفهوم الـ static أكثر فأكثر إذ أن الـ static methods تتعلق باسم الـ class وليس لها علاقة أو وصول إلى أي object

٣. جميع الأمثلة التي ذكرناها عن المؤشر this كانت تتعلق بحقول الـ class ولكن الكلام نفسه ينطبق على توابع الـ class.

هدم الـ objects:

تكلمنا مسبقاً عن أن Java حلت مشكلة الهدم اليدوي للـ objects عن طريق ما يسمى بالـ garbage collector (gc)، ولكن لدينا عدة نقاط يجب أن نفصل فيها:

- في C++ كان لدينا method تدعى (Destructor) تستدعى تلقائياً عند هدم الـ object، وبما أن هدم الـ object في Java يتم عن طريق الـ gc ونحن لا نعلم متى يعمل الـ gc، نستنتج أن وجود الـ destructor لم يعد ذا معنى وبالتالي نجد أن الـ class في Java لا يحوي destructor.
- فعلياً لم نعد بحاجة لوجود الـ destructor وذلك لأن الـ gc لا يهدم الـ object فحسب، وإنما يهدم جميع ما يحويه من حجز ديناميكي أيضاً وبالتالي لم نعد نقلق لهذا الأمر.
- ومع ذلك فقد أمنت Java تابعاً يعمل عندما يقوم الـ gc بهدم الـ object ويدعى finalize()، هذا التابع موجود ضمن الصف Object الذي يشكل أباً لجميع صفوف Java وبالتالي هذا التابع موجود في جميع الصوف التي نكتبها وما علينا إلا أن نعمل له Overriding *مفاهيم الورثة والـ Overriding سيأتي شرحها في محاضرات قادمة*..
- قد يكون من مهمات الـ class الذي نكتبه أن يفتح ملفات مثلاً، و بالتالي هذه الملفات بحاجة لإغلاق، لذا قد نستخدم تابع finalize() لهذه المهمة، ولكن هذا الخيار خاطئ لأن تابع finalize() يستدعى كما أسلفنا عندما يعمل الـ gc وقد ينتهي البرنامج ولا يعمل الـ gc وبالتالي يبقى الملف مفتوحاً وتحدث مشاكل، لذا يجب أن نوجد حلولاً أخرى لهذه المهمة:
  - ❖ إما أن نبقي العمل ضمن تابع finalize() ونستدعي الـ gc قسرياً عن طريق التعليمة System.gc()، وهذا يبطئ عمل البرنامج، لأن الـ gc يستدعى بشكل مدروس بحيث لا يؤثر على سرعة عمل البرنامج، وبالتالي فإن استدعاءه يدوياً غير محبذ.
  - ❖ أو أن نقوم بهذه المهمة يدوياً عن طريق استدعاء method خاصة نكتبها يدوياً.

Hiding the Implementation:

وضحنا في المحاضرة السابقة بنية برنامج الـ Java وسنذكر بها سريعاً:

البرنامج مؤلف من مجموعة ملفات كل منها يحوي class أو أكثر، تتجمع في مجموعة packages.

في الحقيقة إن أي class له نوعان:

١. **package access**: أي أنه حكر على الـ package الموجود ضمنها، وبالتالي لا يمكن إنشاء object منه إلا ضمن الصفوف التي تتشارك معه في نفس الـ package، (ويكون الـ class من هذا النوع عندما لا نضع قبل كلمة class أي شيء).

٢. **public**: أي أننا نستطيع إنشاء object منه في أي class في البرنامج (ويكون الـ class من هذا النوع عندما نضع كلمة public قبل كلمة class).

بما أننا تعرفنا على أنواع الـ class يمكننا أن نوضح فكرة الملفات جيداً، إذ أن الملف الواحد يمكن أن يحوي عدداً غير محدود من الصفوف ولكن يجب أن يكون أحدها فقط من النوع public عندها يجب أن يكون اسم الملف مطابقاً لاسم هذا الصف مع مراعاة حالة الأحرف (including the capitalization)، بالطبع ستكون لاحقة الملف (.java).

كيف نجعل الـ packages تتخاطب مع بعضها؟ أي كيف نستطيع استخدام صفوف package ما ضمن package أخرى؟

يتم هذا عن طريق التعليمة (import) والتي تسمح بتضمين class معين من package ما أو بتضمين جميع الصفوف الـ public ضمن الـ package.

مثال:

```
import java.awt.ActiveEvent;
import java.awt.*;
```

في المثال الأول ضمّنّا الصف (ActiveEvent) فقط من المكتبة (java.awt).

أي أنني أستطيع أن أكتب:

```
ActiveEvent ac = new ActiveEvent();
```

أما في المثال الثاني ضمّنّا جميع الصفوف الـ public من المكتبة (java.awt)، وعادة نلجأ إلى هذه الطريقة عندما نحتاج إلى صفين فما فوق من المكتبة.

ولكن هل من المعقول أن نضع جميع المكتبات التي نحتاجها في مجلد واحد؟

المنطقي أن تترك كل مكتبة في مكانها ويتمكن البرنامج من رؤيتها، وبالتالي يجب أن يكون عندنا متحول معين يحوي مسارات المكتبات التي سنستخدمها، وهذا المتحول يدعى (CLASSPATH) وهو من متحولات الـ (Environment) الخاصة بالنظام.

يحوي هذا المتحول مسارات المجلدات التي تحوي الـ packages التي سنستخدمها في برنامجنا، ويكفي في البرنامج وضع أسماء الـ packages مباشرة، وإذا كان عندنا ملفات jar فيجب وضع مسار الملف نفسه.

ملاحظة: ملف الـ jar عبارة عن تجميع لعدة مكتبات مع بعضها.

Collisions:

لنفرض أنني ضمنت مكتبتين:

```
import ams.*;
import java.awt.*;
```

وكانت كل منهما تحوي الصف (ActiveEvent)، كيف سيفرق المترجم بينهما؟؟

في الحقيقة سيحصل تضارب هنا، والحل يكمن في إدراج اسم المكتبة كاملاً قبل اسم الـ class كما يلي:

```
java.awt.ActiveEvent ac1 = new java.awt.ActiveEvent();
ams.ActiveEvent ac2 = new ams.ActiveEvent();
```

في هذه الحالة لم نعد بحاجة لعمل import، إلا إذا كنا بحاجة لصفوف أخرى من نفس المكتبة.

كل ما سبق من كلام كان يخص الـ class نفسه، ولكن ماذا عن أنواع الحقول والـ methods ضمن الـ class؟

عادة يكون هناك مبرمج للـ class، وهذا الـ class سيستفيد منه مبرمج آخر يدعى (client programmer)، ولكن ليس من المفروض أن يتمكن الـ (client programmer) من الوصول إلى جميع حقول وتوابع الـ class ، وإنما يستطيع الوصول إلى ما يتيحه له مبرمج الـ class فقط..

تضبط هذه العملية عن طريق تحديد نوع الحقول والتوابع ضمن الصف نفسه، وهي ٤ أنواع، ولكن قبل شرحها يجب أن نفرق دوماً بين حالتين:

I. حالة استخدام الـ class (أي إنشاء object منه) عن طريق class آخر في نفس الـ package.

II. حالة استخدام الـ class (أي إنشاء object منه) عن طريق class في package أخرى.

الأنواع الأربعة هي:

١. **<u>public:</u>**

أي أنني إذا عرفت object من الـ class فإنني أستطيع كـ (client programmer) أن أصل إلى جميع الحقول والتوابع من هذا النوع بدون أي قيود وذلك عن طريق كتابة اسم الـ object وإتباعه بنقطة (.) ثم اسم الحقل أو التابع، وذلك في كلى الحالتين **(I)** و **(II)**.

٢. **<u>private:</u>**

أي أنني إذا عرفت object من الـ class فلن أستطيع كـ (client programmer) أن أصل إلى الحقول والتوابع من هذا النوع بأي طريقة وذلك في كلى الحالتين **(I)** و **(II)**، وهذا يعني أن هذه الحقول والتوابع أصبحت خاصة بمبرمج الـ class فقط ولا يحق لغيره الوصول إليها، ولهذه الخاصة ميزات كثيرة منها:

- الاحتفاظ بخصوصية الحقول وعدم السماح للـ (client programmer) بالتعديل على قيمها بشكل مباشر وإنما يتم ذلك عن طريق methods من النوع public وبالتالي سيتمكن مبرمج الـ class من وضع القيود التي يريدها على عملية التعديل ولن تبقى العملية عشوائية، وعادة يتيح مبرمج الـ class توابع ذات أسماء متعارف عليها، فتابع تعديل قيمة حقل ما يكون اسمه (set) وتابع الحصول على قيمة حقل ما يكون اسمه (get) إلا إذا كان (boolean) عندها يكون تابع الحصول على قيمته (is) *سنوضح ذلك عن طريق مثال*.

- يمكن أن يجعل مبرمج الـ class أحد الحقول read only وذلك عن طريق ضبط قيمته مرة واحدة عن طريق الباني مثلاً ثم عدم تعريف تابع (set) له والاكتفاء بتابع (get).
- قد يحتاج مبرمج الـ class إلى حقول خاصة لا يريد أن يقترب منها (client programmer) لا بتعديل ولا بمعرفة قيمتها.
- يمكن أن نحتاج إلى methods خاصة بأعمال داخلية ضمن الـ class ولا يرغب مبرمج الـ class بإتاحتها للـ (client programmer) وبالتالي تكون من النوع private *مثل تابع حساب الراتب في الصف employee، ولكن التابع الذي يجب أن يتاح في هذه الحالة هو تابع (get) للراتب*.
- أهم فائدة للخاصتين (private, public) هي أن يكون تعامل (client programmer) مع الصف محصوراً بالتوابع الـ public بينما يكون عمل الـ class داخلياً كله private، وبالتالي إذا اضطر مبرمج الـ class إلى إجراء تعديل على صفه داخلياً لن يتأثر الكود الذي كتبه الـ (client programmer) إذ أن أسماء التوابع الـ public ستبقى ثابتة ويكون التغيير فقط في ما هو private *وهو أصلاً ممنوع من وصول (client programmer) إليه*.

٣. **<u>package access (friendly):</u>**

في النوعين السابقين لم نفرق بين الحالتين **(I)** و **(II)**، ولكن الوضع هنا مختلف:

<u>في الحالة **(I)**:</u> إذا عرفنا object من الـ class في class آخر من نفس الـ package، عندها نستطيع الوصول إلى هذه الحقول والتوابع مباشرة وكأنها من النوع public.

<u>في الحالة **(II)**:</u> إذا عرفنا object من الـ class في package أخرى، فلن نستطيع الوصول إلى هذه الحقول والتوابع أبداً وكأنها من النوع private.

٤. **<u>protected:</u>**

هذه الخاصة تعتبر أعم من سابقتها، أي أنها تشمل كل خصائص سابقتها بالإضافة إلى عدة خصائص أخرى تتعلق بموضوع الوراثة (inheritance) سنتعرض لها بالتفصيل عندما نتكلم عن الوراثة.

<u>ملاحظة:</u>

لنجعل حقل أو تابع من الأنواع (public, private, protected) يكفي أن نضيف أحد هذه الكلمات قبل نوع الحقل أو قبل (return type) بالنسبة للتوابع، أما الخاصة (friendly) فتكون بعدم وضع أي كلمة قبل تعريف التابع أو الحقل أي أن جميع الأمثلة في المحاضرات السابقة تعتبر من النوع friendly، والمثال القادم سيوضح هذه النقطة تماماً.

مثال:

```
public class Employee
{
    private String name;
    private int salary;
    private boolean married;
    int age;

    public Employee(String name, boolean married, int age)
    {
        this.name = name;
        this.married = married;
        this.age = age;
    }

    public void setName(String name)
    {
        this.name = name;
    }

    public String getName()
    {
        return name;
    }

    public void setMarried(boolean married)
    {
        this.married = married;
    }

    public boolean isMarried()
    {
        return married;
    }

    public int getSalary()
    {
        salary = calcSalary();
        return salary;
    }

    public int getAge()
    {
        return age;
    }

    private int calcSalary()
    {
      ...
    }
}
```

نتائج من المثال:

- الحقل (age) من النوع Read only بالنسبة للصفوف التي تقع خارج الـــ package ،أما الصفوف التي تقع في نفس الـــ package فيمكن أن تعدل قيمة هذا الحقل.
- لا يمكن لأي class الولوج إلى تابع حساب الراتب، وإنما بالإمكان فقط معرفة قيمة الراتب.

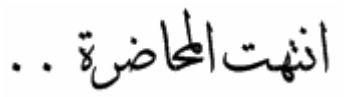
أعتقد أن المثال واضح، ولكن أتمنى من أي زميل يحب أن يستفسر عن أي نقطة في المثال أن يراسلنا على الـ Email الموضح في آخر المحاضرة.

نتيجة:

هذه الخصائص هي التي تجسد مفهوم الكبسلة (encapsulation) في الـ class، فالـ class عبارة عن نمط يحوي مجموعة من الحقول –يفترض أن تكون– من الأنواع (friendly, private, protected) تضبط علاقتها مع الـ (client programmer) مجموعة من التوابع من النوع (public) فيما يشبه عملية إرسال رسائل واستقبال ردود بين الـ classes.

ملاحظة: إذا وضعنا ملفين في نفس المسار دون أن ينتميا إلى أي package، فستكون العلاقة بين الـ classes في هذين الملفين من النوع ((package access (friendly)، وهذا ما يسمى اصطلاحاً بـ (default package).

كانت المحاضرة طويلة وغزيرة المعلومات فأتمنى أن تكون خالية من الأخطاء ومفيدة المعلومات، وجل من لا يسهو...



lectures_team@hotmail.com