المحاضرة السابعة

20

Inner classes

تنويه:

هذا البحث هام جداً وفيه الكثير من التفاصيل الدقيقة، وأتوقع شخصياً أن فيه مجالاً واسعاً لأسئلة الفحص، والمشكلة أنه طرحه في المرجع ليس منظماً كثيراً، لذا سأحاول جاهداً تنظيم معلوماته وترتيبها فأرجو أن أوفق لهذا، وأنصح زملائي بقراءة البحث من المرجع ***بعد إنهاء قراءته من المحاضرة*** وأنتظر أي تعليق أو تنبيه حتى أستدرك ما قد ينقص في محاضرة الأسبوع القادم..

من الممكن أن نعرف جسم الـ class ضمن جسم class آخر، وهذا ما يفتح علينا الباب للدخول في خضم بحث كبير وكثير التفاصيل وهو *Inner classes*.

سنتحدث الآن عن جميع تفاصيل هذا النوع الجديد من الصفوف بدون التطرق لأهميته واستخداماته والتي سنتركها للنهاية..

ما الجديد في الـ Inner classes ؟

- يعرف ضمن جسم class آخر يدعى (Outer class).
- إن وضعية الـ Inner class ضمن جسم الـ Outer class تخفي اسم الـ Inner class عن باقي الصفوف الخارجية وبالتالي لا يمكننا إنشاء غرض منه بمجرد ذكر اسمه مثل أي صف آخر، وإنما يخضع هذا الأمر لبعض التفاصيل..

مثال:

```
class Outer
{
      class Inner {}

      public static void main() {
            // Inner n = new Inner();     compile-time error!!
      }
}
```

- من أهم شروط إنشاء غرض من الـ Inner class هو أن يكون هذا الغرض محتوى في غرض من الـ Outer class، و لا يوجد أي إمكانية لإنشاء أي غرض من الـ Inner class بدون وجود غرض من الـ Outer class وهذا الغرض (من الـ Outer class) يدعى *Enclosing object*.

- الأصل في التعامل مع الـ Inner class أن ينشأ الغرض منه ضمن method في الـ Outer class و التي ترد مؤشر (reference) على هذا الغرض للتعامل معه من خارج الـ Outer class، ولكن هذا الـ reference ليس عادياً:
  - في بعض حالات الـ Inner class *التي سيأتي تفصيلها* يمكن تعريف مؤشر عليه كما يلي: *(Outer class).(Inner class)*
  - يمكن أن يرث الـ Inner class من أي class أو interface خارج جسم الـ Outer class، وبالتالي يمكن استخدام ميزات الـ Upcasting معه من خارج الـ Outer class.

مثال(I):

```
interface Interface1 {}

public class Outer
{
	class Inner implements Interface1 {}

	public Inner newInner() {
		return new Inner();
	}

	public static void main(String[] args) {
		Outer o = new Outer();
		Interface1 a = o.newInner();
		Outer.Inner b = o.newInner();
	}
}
```

نلاحظ أننا استطعنا التعامل مع الغرض الذي عرفناه بطريقتين:

١. إما أن نعرف مؤشراً عليه: وهذه الطريقة ليست محققة دوماً *كما سنرى بعد قليل*.

٢. أو أن نستخدم الـ Upcasting: وهذه هي الطريقة السليمة والفعالة في كل الحالات.

- يمكن تعريف Inner class ضمن method وحتى ضمن scope، وهذا يسمح لنا بتصنيف أنواع الـ Inner classes.

أنواع الـــ Inner classes:

حتى ندرس خصائصها يجب تصنيف الـــ Inner classes:

١. جسم الـــ Inner class ضمن جسم الـــ Outer class مباشرة:

وهو بدوره يقسم لنوعين:

a. (Non-Static): وهو نفس النوع الذي ضربنا عليه جميع الأمثلة السابقة، وله خصائص هامة:

- تستطيع توابعه أن تصل إلى جميع حقول وتوابع الـــ Outer class سواء منها الـــ (instance) أو الـــ (static)، وسواء منها الـــ public والـــ private وغيرها، أي أنه يمتلك نفس صلاحيات أي method ضمن الـــ Outer class.
- عند إنشاء غرض من هذا النوع، يكون لديه مؤشر على الـــ Enclosing object الـــذي يحويـــه وبالتالي يمكنه الوصول إلى جميع خصائصه كما أشرنا في النقطة السابقة، هذا المؤشر يكتب كمـــا يلي:

(Outer class name).this

(Outer class name): هو اسم الـــ class وليس اسم الـــ object.

لنأخذ المثال(I) ونطبق عليه:

```
Outer.this.
```

- يجب الانتباه إلى الفرق بين المؤشر السابق وبين المؤشر (this) فقط، والذي يشير إلـــى الغـــرض الحالي من الـــ Inner class كأي class آخر.
- **أنواعه:** تعودنا أن الـــ class له أحد النوعين: (public) أو (friendly) حصراً، لكن الوضـــع هنا مختلف إذ يمكن أن يكون نوع هذا الـــ class واحداً من الأنواع التالية:

❖ ***public***: إذا أضفنا كلمة public قبل الـــ Inner class يصبح لديه الخصائص التالية:

- يمكن تعريف مؤشر عليه من خارج الـــ Outer class بكتابة اسم الـــــ Outer class وإتباعه بنقطة ومن ثم اسم الـــ Inner class:

(Outer class).(Inner class)

- يمكن إنشاء غرض منه من خارج الـــ Outer class ولكن ليس بالطريقة المعتادة، وإنما بطريقة غير مألوفة كما في المثال التالي:

```
class Outer
{
      class Inner {}

      public static void main() {
            Outer o = new Outer();
            Outer.Inner n = o.new Inner();
      }
}
```

نلاحظ بوضوح أننا لا نستطيع إنشاء غرض من الـ Inner class بدون أن ننشئ غرضاً من الـ Outer class، كما نلاحظ كيف عرفنا المؤشر على الـ Inner class بـنفس الطريقة التي شرحناها في النقطة السابقة.

- عند الوراثة سيمتلك الصف الذي سيرث من الـ Outer class حق التعامل مع الـ Inner class وكأنه معرف ضمنه..

❖ ***private***: إذا أضفنا كلمة private قبل الـ Inner class يكتسب الخصائص التالية:

- لا يمكن مطلقاً إنشاء غرض من الـ Inner class خارج حدود الـ Outer class ولا حتى مجرد تعريف مؤشر عليه.
- عند الوراثة لا يمكن للصف الابن التعامل مع الـ Inner class الموجود عند أبيه مباشرة.
- الطريقة الوحيدة للاستفادة منه خارج حدود الـ Outer class هي تمرير غرض منه عن طريق أحد التوابع و الاستفادة من الـ Upcasting كما فعلنا في المثال(I).

❖ ***friendly***: عندما لا نضع أي كلمة قبل الـ class يتمتع بالخصائص التالية:

- في نفس الـ package هو تماماً كالـ public.
- في package أخرى هو تماماً كالـ private.

❖ ***protected***: إذا أضفنا كلمة protected قبل الـ Inner class يكتسب الخصائص التالية:

- في نفس الـ package هو تماماً كالـ public.
- في package أخرى هو تماماً كالـ private ولكن الفرق هو أنه عند الوراثة يستطيع التعامل مع الـ Inner class وكأنه معرف ضمنه.

- نستنتج مما سبق أننا نستطيع تطبيق الـ Upcasting على الـ Inner class دوماً ولكن لا نستطيع تطبيق الـ Downcasting إلا في الحالات التي نستطيع فيها تعريف مؤشر على الـ Inner class.
- لا يمكن أن تحوي هذه الصفوف أي متحولات أو توابع أو صفوف (static).

b. (Static): ويكون بوضع كلمة static قبل الـ Inner class:

- يدعى هذا النوع من الصفوف بـ (*Nested classes*).
- لا يملك هذا الـ class أي مؤشر على أي غرض من الـ Outer class، أي أنه لا يستطيع التحكم بالحقول والتوابع غير الـ static.
- يشكل الـ Outer class بالنسبة له مجرد غطاء.

- يمكن تعريف غرض منه بشكل طبيعي بدون أن نضطر لتعريف أي غرض من الـ Outer class لأنه غير مرتبط به.
- يمكن تعريف مؤشر عليه بوضع اسم الـ (Outer class) أولاً ثم إتباعه بنقطة ثم اسم الـ (Nested class).
- إذا أردنا تعريف غرض منه عن طريق أحد توابع الـ Outer class فيجب أن يكون هذا التابع static.
- يمكن أن نضع Nested class ضمن Nested class آخر وبالتالي يمكن الوصول إلى جميع المستويات.
- بما أن الـ Nested class لا يهمه التعامل مع أي object من الـ Outer class، فبإمكاننا أن نضعه ضمن الـ interface بدون أن يؤثر على بنيتها، وبدون أن يضطر الصف الذي يحققها من إعادة تعريفه.
- يمكن أن يعرف بأحد الأنواع (public, private, protected, friendly) وهي تعني نفس المعاني تقريباً للنوع السابق، والفرق أن عملية (new) هنا تستخدم بشكل طبيعي.
- تمعن جيداً في المثال القادم وحاول اكتشاف الخصائص السابقة كلها فيه:

مثال(II):

```
interface Interface1
{
	public static class InInterface {
		private int i = 11;
		public int value() {return i;}
	}
}

class Outer implements Interface1
{
	static int j;

	public static class Inner {
		Inner(int i) {j = i;}
		public static class InInner {}
	}

	public static Inner getInner() {
		return new Inner(5);
	}
}

public class Main
{
	public static void main(String[] args) {
		Outer.Inner n1 = Outer.getInner();
		Outer.Inner.InInner n2 = new Outer.Inner.InInner();
		Interface1.InInterface n3 = new Interface1.InInterface();
	}
}
```

٢. جسم الـ Inner class ضمن method أو scope ضمن الـ method:

يمكن أن نعرف جسم الـ Inner class ضمن method، وعندها يكون مرئياً فقط في هذه الـ method، ولا يمكن رؤيته في باقي توابع الـ Outer class، كما يمكن أن يعرف الـ Inner class ضمن scope ضمن method ما وعندها لا يكون مرئياً إلا ضمن هذا الـ scope، وحتى الـ method التي تحوي الـ scope لا تستطيع رؤيته.

يقسم هذا النوع إلى نوعين:

a. (Local Inner classes):

وهو Inner class طبيعي معرف ضمن تابع أو scope، وخصائصه:

- لا يستطيع هذا الـ class أن يرى أي شيء خارج الـ method أو الـ scope التي يقبع داخلها.
- كما أنه لا يُرى من خارج الـ method أو الـ scope التي يقبع بداخلها، وهذا يعني أننا إذا أحببنا الاستفادة منه خارجهما يجب استخدام الـ Upcasting.

مثال:

```
interface MyInterface
{
	public void ToOverride();
}

public class Outer
{
	public MyInterface myMethod() {
		class LocalClass implements MyInterface
		{
			public void ToOverride() {
				System.out.println("I'm Local Class");
			}
		}
		return new LocalClass();
	}
}
```

b. (Anonymous Inner classes):

وهو صف خاص لا يملك أي اسم!

الفكرة أنني قد أضطر لاستخدام غرض من صف ما مشتق من صف آخر أو محقق لـ interface ما لمرة واحدة فقط، وبالتالي ليس لدي أي أهمية لإعطائه اسماً، إذ أن إعطاء اسم لصف هدفه إتاحة إنشاء أغراض منه متى شئت، وبما أنني ضامن أنني سأستخدم هذا الغرض مرة واحدة فقط فلا داعي للاسم.

سنوضح طريقة التعامل عن طريق مثال، ولكي لا نذهب بعيداً سأعيد كتابة المثال السابق باستخدام الـ Anonymous Inner classes.

مثال:

```
interface MyInterface
{
	public void ToOverride();
}

public class Outer
{
	public MyInterface myMethod() {
		return new MyInterface()
		{
			public void ToOverride() {
				System.out.println("I'm Local Class");
			}
		};
	}
}
```

نلاحظ من المثال:

- ينطبق على هذا الصف جميع ما ذكرناه على الـ Local class.
- لا يمكن تعريف هذا الصف إلا عن طريق توريثه مباشرة من class أو interface.
- بما أن هذا الـ class لا يملك اسماً فإن التعامل معه سيتم حصراً عن طريق الـ Upcasting، وبالتالي لا معنى لإضافة أي توابع أو حقول (public) فيه لأن أحداً لن يصل إليها أبداً، إذ أن الوصول إلى هذه الخواص يقتضي وجود مؤشر من نوع هذا الصف، وطبعاً هذا غير موجود.
- بما أن تعريف هذا الصف جاء ضمن تعليمة return فيجب أن تنتهي التعليمة بفاصلة منقوطة (;).
- عندما ننشئ Anonymous Inner classes ويكون وارثاً من صف (وليس من Interface) فإن باني الصف الأب يستدعى قبل أي شيء في الـ Anonymous Inner classes *كما اعتدنا في الوراثة*، وبالطبع فإن الباني الذي سيستدعى هو الـ default constructor.

مثال:

```
abstract class Base
{
	public Base() {
		System.out.println("Base default constructor");
	}
	public Base(int i) {
		System.out.println("Base constructor, i = " + i);
	}

	public abstract void f();
}

public class AnonymousConstructor
{
	public static Base getBase(int i) {
		return new Base(i)
		{
			{
				System.out.println("Inside instance initializer");
			}
```

```
            public void f() {
                System.out.println("In anonymous f()");
            }
        };
    }

    public static void main(String[] args) {
        Base base = getBase(47);
        base.f();
    }
}
```

إن خرج البرنامج السابق هو:

```
Base constructor, i = 47
Inside instance initializer
In anonymous f()
```

لندرس هذا المثال معاً:

- إن استدعاء التابع getBase أدى إلى إنشاء غرض من الصف AnonymousConstructor، والذي يرث من الصف Base، وبالتالي يجب أن يستدعى الباني الافتراضي، ولكننا أحببنا أن نستدعي بانياً غير الباني الافتراضي لذا نلاحظ أننا استدعيناه عند إنشاء الصف AnonymousConstructor وذلك عن طريق التعليمة:

```
new Base(i)
```

- ألم تتساءل كيف سنعرف بانياً للـ Anonymous Inner classes؟

المشكلة أنه لا يملك اسماً حتى نعطيه نفسه للباني كما اعتدنا، لذا لا يوجد حل إلا استخدام الـ *instance initializer* وهو عبارة عن scope بدون أي ترويسة كما في المثال السابق.

لدينا خاصة مشتركة بين الـ Anonymous Inner classes و الـ Local Inner classes:
قلنا أن كلا النوعين لا يستطيعان رؤية أي شيء خارج الـ method أو الـ scope التي تحويهما، وفي الحقيقة هما لا يستطيعان رؤية أي شيء داخلها أيضاً إلا إذا كان من النوع *final* حصراً.

الوراثة من الـ Inner class:

لدينا مشكلة هنا وهي أن الـ Inner class يجب ألا يستخدم بدون وجود غرض من الـ Outer class، فكيف سنضبط هذا الكلام عند الوراثة؟

مثال:

```
class WithInner {
    class Inner {}
}

public class InheritInner extends WithInner.Inner {
    //! InheritInner() {} // Won't compile

    InheritInner(WithInner wi) {
        wi.super();
    }
```

```
	public static void main(String[] args) {
		WithInner wi = new WithInner();
		InheritInner ii = new InheritInner(wi);
	}
}
```

الفكرة أننا يجب أن نوجد طريقة لتعريف غرض من الصف WithInner قبل تعريف غرض من الصف InheritInner، لذا فإن الباني الافتراضي لن يعمل هنا والبديل عن ذلك هو الطريقة التي اتبعت في المثال السابق.

هل يمكن عمل Override للـ Inner class؟

إذا عرفنا في الصف الابن Inner class له نفس اسم Inner class موجود في الأب ما الذي يحدث؟

هل سيشكل الـ Inner class في الابن Override لمثيله عند الأب؟

للإجابة عن هذا السؤال لنر ما هو خرج هذا البرنامج:

مثال:

```
class BaseClass
{
	private Inner y;

	protected class Inner
	{
		public Inner() {System.out.println("BaseClass.Inner()");}
	}

	public BaseClass() {
		y = new Inner();
	}
}

public class SubClass extends BaseClass
{
	public class Inner
	{
		public Inner() {System.out.println("SubClass.Inner()");}
	}

	public static void main(String[] args) {
		new SubClass();
	}
}
```

للأسف جوابك خاطئ..

الخرج ليس كما توقعت:

```
SubClass.Inner()
```

ولكنه في الحقيقة:

```
BaseClass.Inner()
```

أعتقد أن الفكرة قد وصلت: يبقى كل Inner class على مستوى الـ Outer class الذي يحويه ولا تجري أي عملية Override.

لماذا Inner classes؟[1]

تعلمنا حتى الآن كيف نكتب Inner class، ولكن كيف ومتى نستخدمه؟

في الحقيقة إن الـــ Inner class يستخدم غالباً في الوراثة، حيث يتيح إمكانيات كبيرة في هذا المجال، فمثلاً:

- قد لا يكون الصف A محققاً تماماً للصف B وإنما قد يكون قسم منه فقط محققاً لهذا الصف، وقسم آخر يحقق الصف C، وثالث يحقق الـــ D (Interface)...

عندها ليس من المنطقي أن يرث الصف نفسه من جميع ما سبق، وخصوصاً أن الوراثة من أكثر من صف غير ممكنة أصلاً، لذا فالحل أن نجزئ مهمات الصف A على عدة Inner classes كل منها يرث من الـــ class أو الـــ Interface المناسبة له.

- الجملة التالية مذكورة في المرجع بحرفيتها فأحببت أن أدرجها هنا:

** Each inner class can independently inherit from an implementation. Thus, the inner class is not limited by whether the outer class is already inheriting from an implementation *.*

- يمكن أن يكون كل من الصفين B, C يحقق الـــ A (Interface) بطريقته الخاصة، ومطلوب من الصف D أن يمتلك كلتا الطريقتين، عندها لا حل إلا الـــ Inner class.

خلاصة:

بنهاية بحث الـــ Inner classes يمكن القول بأننا أنهينا أهم دعامات البرمجة غرضية التوجه..

وفي الحقيقة يطول الحديث في هذا البحث ولكنني حاولت ذكر المفيد وترتيب الأفكار قدر المستطاع، وأحب أن أنوه إلى أن على من يحب أن يكمل طريقه ويحترف البرمجة بلغة Java خاصة وفي البرمجة غرضية التوجه عامة ويصبح فعلاً (Thinking in Java)، فعليه أولاً أن يكون قد فهم الأبحاث التي كنا ندرسها في المحاضرات السابقة جيداً، ليستطيع إكمال المشوار وتعلم باقي الأبحاث التي تعتبر تقنيات مهمة للانتقال إلى مستوى جديد من الفكر البرمجي..

[1] سأتكلم هنا عن أفكار وأترك لكم أمثلة المرجع تحت العنوان (Why inner classes) لكي لا تطول المحاضرة بلا داع