المحاضرة الرابعة

12

مقدمة:

إن طريقة طرح الدكتور للمادة من حيث ترتيب الأبحاث تعتبر طريقة غير مرتبة، لذا أستميح زملائي عذراً في عدم التقيد بترتيب المحاضرات كما طرحها الدكتور، وإنما سأتقيد بترتيب الأبحاث في المرجع ما استطعت مع بعض الاجتهاد الشخصي، فإذا كان لديكم أي تعليق أو انتقاد أرجو منكم إفادتي به عن طريق الـ Email الخاص بالـ Team والموضح في نهاية المحاضرة.

Reusing Classes:

ذكرنا سابقاً أن المهارة في استخدام الـ OOP تكمن في إمكانية إعادة استخدام الـ classes التي عرفناها مسبقاً.

ولكن إعادة الاستخدام لا تكون عن طريق (copy - paste)!! ولكن تتم عن طريق تقنيات برمجية هي:

١. التجميع (*composition*): إنشاء objects ذات أنواع متعددة ضمن class ما.
٢. الوراثة (*inheritance*): إنشاء class جديد يعتبر نوعاً من أنواع الـ class الأصلي، أو حالة خاصة له.
٣. إن ميزات الوراثة تظهر عن طريق الخاصة الهامة جداً: تعددية الأشكال (*Polymorphism*).

التجميع (composition):

استخدمنا هذه العملية كثيراً خلال المحاضرات السابقة، وهي خاصة مهمة وبديهية يوضحها المثال التالي:

```
class Door {...}
class Window {...}

class Car
{
      Door d1, d2;
      Window w1, w2, w3, w4;
}
```

طبعاً لا بد من عمل (init) لهذه المتحولات ضمن الباني.

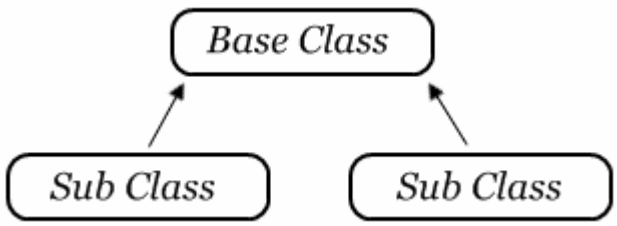
نستنتج أن خاصة التجميع تكون باستخدام صفوف متعددة ضمن صف ما عن طريق تعريف objects منها ضمن ذلك الصف، وهذه العلاقة توصف بالعبارة: **has a**.

بالنسبة للمثال السابق نستطيع القول:

Car **has a** Doors , Car **has a** Windows

الوراثة (inheritance):

وهي وسيلة هامة ودقيقة لإعادة استخدام class، وتقوم على فكرة كون الـ class الوارث هو عبارة عن نوع من أنواع الـ class الموروث منه، أي أنه يشترك معه في **جميع** صفاته ووظائفه، وقد يزيد صفات جديدة أو يقتصر على إعادة تعريف الخصائص الموجودة مسبقاً في الـ class الأب.



أساسيات في الوراثة:

- إن الصف الأب يدعى (*base class*)، والصف الابن يدعى (*sub class*).
- تتم الوراثة بوضع كلمة (**extends**) بعد اسم الـ (*sub class*) ثم إتباعها باسم (*base class*).
- لا يمكن الوراثة في Java إلا من أب واحد فقط، أي أن مفهوم الوراثة المتعددة غير موجود.

مثال:

```
class Food {...}

class Fruit extends Food
{...}
```

سماحيات الوصول:

إن الـ (*sub class*) يرث جميع حقول وتوابع الـ (*base class*) ولكنه لا يستطيع الوصول إليها جميعاً، وإنما تختلف سماحيات وصوله بحسب أنواع هذه الحقول والتوابع، ونميز حالتين:

a. (*base class*) و (*sub class*) في package واحدة:

في هذه الحالة يستطيع الـ (*sub class*) الوصول إلى جميع الحقول والتوابع الخاصة بـ (*base class*) ما عدا الـ **private**.

b. (*base class*) و (*sub class*) ليسا في package واحدة:

في هذه الحالة لا يستطيع الـ (*sub class*) الوصول إلا الحقول والتوابع ذات الأنواع (**public**) و (**protected**).

ماذا تعني كلمة (protected)؟

يختلف مفهوم الـ protected بحسب الحالتين السابقتين:

a. في نفس الـ package ينطبق مفهوم الـ protected على مفهوم friendly تماماً.
b. أما عندما تختلف الـ packages فإن الحقول والتوابع الـ protected تكون مشابهة تماماً للـ private، والفرق هو أن الحقول والتوابع الـ protected مسموح الوصول إليها بالنسبة للأبناء، أما الـ private فإن الأبناء تملكها ولكن لا تستطيع الوصول إليها.

ما الحكمة من عدم الوصول إلى الحقول الـ private؟

الحكمة أنها حقول خاصة بالصف الأب وهي خاصة لدرجة أنه لا يحق حتى للابن الذي يملكها الوصول إليها إلا عن طريق التوابع التي يتيحها الأب لهذا الغرض.

إذا كنا نبرمج الصف الأب، الأصل أن تكون الحقول private والتعامل معها عن طريق توابع public أو protected، إلا إذا كان ضرورة كبيرة أن يصل الابن إلى هذه الحقول عندها تكون من النوع protected.

الوراثة والبناء:

بما أن الـ (*sub class*) سيرث جميع حقول الـ (*base class*)، وبما أن الباني الخاص بالـ (*sub class*) مسؤول عن عمل (initialization) لهذه الحقول، فمن المنطقي بل الضروري جداً استدعاء constructor الأب، لذا فإن Java تقوم باستدعاء constructor الأب بمجرد إنشاء object من الـ class الابن، وحتى قبل استدعاء الـ constructor الخاص بالـ class الابن، كما أن الأب نفسه سيستدعي الـ constructor الخاص بأبيه، والتالي سيصبح لدينا تسلسل تنفيذ للبواني من أعلى الهرم الوراثي إلى أسفله.

س. ولكن أي constructor للأب سيستدعى؟؟

ج. سيستدعى default constructor للأب.

س. ولكن ماذا لو أردنا استدعاء constructor آخر غير الـ default constructor؟؟

ج. الحل يكون عن طريق استخدام الكلمة المحجوزة (super) والتي يشبه عملها عمل المؤشر this إلا أنها مختصة بالوراثة، ولها عدة استخدامات:

١. استدعاء بواني الأب: هذا الاستخدام يكون حصراً ضمن الباني، ويكون أول تعليمة في الباني وشكله:

```
super(...);
```

٢. الوصول إلى حقول وتوابع الأب: أي أنها تصبح مؤشراً على الصف الأب، ولكننا اتفقنا على أن جميع حقول وتوابع الأب أصبحت متوفرة بالنسبة للابن، فما أهمية هذا الاستخدام؟؟

تظهر أهمية هذا الاستخدام في موضعين:

أولاهما: عندما نريد الوصول إلى الحقول والتوابع الـ static عند الأب (إذ أنها لا تخضع لنفس قوانين الـ Overriding التي تخضع لها التوابع العادية).

وثانيهما: عند عمل Overriding وسيأتي شرحها بعد قليل..

٣. للوصول إلى الأجداد يمكن استخدام super أكثر من مرة (super.super...) ولكن هذه الطريقة في العمل غير محبذة، والأصل أن تكون علاقة كل صف مع أبيه المباشر فقط.

للوراثة نوعان:

١. إذا لم يضف الـ (*sub class*) أي member أو method إلى ما ورثه من الـ (*base class*)، وإنما اكتفى بإعادة تعريف بعض الـ methods، ويوصف هذا النوع بالعبارة: **is a**.

مثال:

ليكن لدينا صف اسمه Car ونريد أن نرث منه الصفين (RallyCar, TownCar)، نلاحظ أننا لن نضيف أي خصائص جديدة إلى أي من الصفين، وإنما الاختلاف قد يكون بقوة المحرك أو بعدد مقاعد السيارة أو ببعض الوظائف الأخرى، كما أننا نستطيع أن نقول:

RallyCar **is a** Car , TownCar **is a** Car

٢. إذا أضاف الـ (*sub class*) members أو methods إلى ما ورثه من الـ (*base class*)، ويوصف هذا النوع بالعبارة: **is like a**.

مثال:

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 عندها نقول:

Car **is like a** Vehicle , Bus **is like a** Vehicle

Overriding:

قد تختلف بنية بعض التوابع عند الانتقال إلى الابن عن ما كانت عليه في الأب، لذا سنضطر إلى إعادة تعريفها مرة أخرى وهذا ما يعرف بـ (Overriding)، فالتابع الذي نعمل له Override يكون له نفس Signature التابع الأصلي الموجود في الأب تماماً أي بدون زيادة أو تغيير أي arguments، وإذا حدث أي تغيير في الـ Signature، فستعتبر العملية Overload ويصبح للصف الابن تابعان بنفس الاسم، بينما تحجب عملية الـ Override التابع الموجود في الأب تماماً.

قد لا تختلف بنية تابع الابن عن تابع الأب كلياً وإنما تزيد فيها بعض العمليات، لذا من الضروري الوصول إلى تابع الأب ضمن تابع الابن ، وهذه هي المهمة الثانية للكلمة (super) *انظر المثال(I)*.

سنضع مثالاً عاماً نشرح فيه ما نستطيع عليه من مفاهيم الوراثة ولكنني أعيد نفس النصيحة لزملائي بالعودة إلى المرجع وفهم أمثلته لأنها غنية جداً بالأفكار التي فرغنا للتو من شرحها نظرياً:

المثال(I):

```
public class Student
{
	private String name;
	protected int birthDate;

	public Student(String name, int birthDate){
		this.name = name;
		this.birthDate = birthDate;
		System.out.println("Student Name : " + name);
	}

	public void doExam(){
		System.out.println("Student " + name + " do exam..");
	}

	public String getName(){
		return this.name;
	}
}

public class ItStudent extends Student
{
	private String exam;

	public ItStudent(String name, int birthDate, String exam){
		super(name, birthDate);
		this.exam = exam;
	}

	public void doExam(){
		super.doExam();
		System.out.println("ItStudent name: " + getName());
		System.out.println("ItStudent birthDate: " + birthDate);
		System.out.println("exam name: " + exam);
	}

	public static void main (String[] args){
		ItStudent stu = new ItStudent("Salem", 1985, "programming");
		stu.doExam();
	}
}
```


خرج البرنامج السابق هو:

```
Student Name : Salem
Student Salem do exam..
ItStudent name: Salem
ItStudent birthDate: 1985
exam name: programming
```

سندرس المثال بالتفصيل:

- نلاحظ استدعاء باني الصف الابن لباني الصف الأب عن طريق التعليمة super والذي تولى بدوره عمل initialization لجميع الحقول الموروثة من الأب.
- تابع doExam() في الصف الابن أعاد تعريف (Override) تابع doExam() في الصف الأب، ونلاحظ استدعاء تابع الأب ضمن تابع الابن عن طريق المؤشر super.

- نلاحظ أن الصف الابن لم يستطع الوصول مباشرة إلى خاصية name لأنها private في الصف الأب، وإنما اضطر لاستدعاء التابع ()getName، بينما استطاع الوصول مباشرة إلى الخاصة birthDate ذات النوع protected.

الهدم:

كنا قد تحدثنا عن أن الـ class في Java لا يحوي Destructor وإنما علينا أن نبنيه بيدنا، وكما أن الباني في الوراثة يستدعى من الأب الأعلى وحتى الابن الأسفل، فإن الهادم يجب أن يستدعى ولكن بدءاً من الابن الأسفل إلى الأب الأعلى، وذلك لأن عمل حقول الابن قد يكون مبنياً على عمل حقول الأب، لذا ستبنى حقول الأب أولاً وتهدم حقول الابن أولاً.

Name hiding:

لنفرض أن لدينا تابع في الـ (*base class*) له عدة نسخ (Overload)، وقمنا بعمل نسخة جديدة (Overload) من هذا التابع في الـ (*sub class*):

في لغة C++ التابع المعرف في الصف الابن يغطي توابع الأب، ولكن Java لا تخفي هذه التوابع وإنما تصبح جميع هذه التوابع وكأنها نسخ متوفرة جميعاً (Overloading) في الصف الابن.

مثال:

```
class A
{
	void func(int i){
		System.out.println("int func " + i);
	}
	void func(double d){
		System.out.println("double func " + d);
	}
}

class B extends A
{
	void func(char c){
		System.out.println("char func " + c);
	}
}
```


نستطيع في Java استدعاء أي تابع من هذه التوابع كما يلي:

```
public static void main (String[] args){
	B b = new B();
	b.func(1);
	b.func(3.7);
	b.func('d');
}
```

الخرج سيكون:

```
int func 1
double func 3.7
char func d
```

Polymorphism:

علمنا أن الوراثة تساعد المبرمج على الاستفادة من الكود الذي قد كتبه مسبقاً وتوفر عليه إعادة كتابته مرة أخرى، ولكن في الحقيقة ليست هذه هي الميزة الكبرى للوراثة، ولكن روعة الوراثة تتجلى في العلاقات التي تربط بين أعضاء الهرم الوراثي للـ classes.

هناك علاقتان أساسيتان تضبطان علاقة الـ (*base class*) بالـ (*sub classes*) هما:

١. Upcasting:

ذكرنا مسبقاً أن الـ class الابن إنما هو نوع من أنواع الـ class الأب، لذا فإننا نستطيع التعامل معه من منظور أنه حالة خاصة من هذا الأب.

مثال:

```
class Fruit
{
	void eat(){
		System.out.println("Eat Fruit");
	}
}

class Apple extends Fruit
{
	void eat (){
		System.out.println("Eat Apple");
	}
}

class Banana extends Fruit
{
	void eat (){
		System.out.println("Eat Banana");
	}
}
```


في المثال السابق تتضح فكرة كون الصف الابن حالة خاصة من الصف الأب، ونلاحظ أن كلى الصفين Apple, Banana يعيدان تعريف التابع ()eat، فإذا تكلمنا بوجهة نظر منطقية سنلاحظ أن أي نوع من أنواع الفاكهة يؤكل، وإن كانت طريقة الأكل تختلف من نوع لآخر لذلك يأتي مفهوم إعادة تعريف طريقة الأكل في كل من الأبناء (Override)، ولكن الأب والأبناء جميعاً يشتركون في صفة الأكل.

بما أن جميع الأبناء (Apple, Banana) هي في النهاية فاكهة (Fruit) تؤكل، فلماذا لا نتعامل معها من هذا المنظور العام؟

لنفرض أن لدينا صف (Man) نريد أن نكتب فيه تابع لأكل الفواكه، فهل يعقل أن نكتب تابع يكون دخله Apple وآخر دخله Banana وثالث..

هنا يأتي مفهوم الـــ Upcasting ليحل لنا هذه المشكلة عن طريق التعامل مع الصفة المشتركة لجميـــع هـــذه الصفوف وهي أنها (فاكهة)، أي أنه سيتعامل مع الصف (Fruit) الذي يمثل الصفة المشتركة بين جميع أبنائـــه عن طريق تعريف تابع واحد لكل هذه الصفوف، والمثال التالي يوضح الفكرة:

مثال:

```
class Man
{
	void eatFruit (Fruit f){
		f.eat();
	}

	public static void main (String[] args){
		Man m = new Man();
		m.eatFruit(new Apple());
		m.eatFruit(new Banana());
	}
}
```


ماذا نتوقع أن يكون خرج البرنامج السابق؟

```
Eat Apple
Eat Banana
```

لنعدد الظواهر الجديدة في هذا المثال مع التعليل:

أولاً: كيف مررنا object من النوع Apple إلى method تأخذ object من النوع Fruit؟ والسؤال يتكرر من أجل الـــ object من النوع Banana..

التعليل: إن المؤشر (f) أصبح يؤشر على object من النوع Apple ولكنه لا يزال ينظر إليه على أنه Fruit أي أنه لا يستطيع التعامل إلا مع الحقول والتوابع المعرفة في Fruit، وبما أن جميع حقـــول وتوابـــع Fruit موجودة حتماً في Apple، فإننا سنضمن أن أي استخدام للـــ object من النوع Apple من المنظور Fruit لن يعطي أي مشاكل.

ثانياً: اتفقنا أنه لا مشكلة في النقطة الأولى، ولكن عندما استدعينا تابع الـــ ()eat من مؤشر من النوع Fruit كيف ظهر لنا خرج التابع ()eat الخاص بـــ Apple؟

الفكرة تشبه مفهوم الـــ virtual في C++، حيث كان للـــ method في C++ خيارين:

- إما أن تكون عادية (ليست virtual): عندها تستدعى الـــ method الخاصة بالأب.
- أو أن تكون virtual: عندها تستدعى الـــ method الخاصة بالابن.

في Java جميع الـــ methods من النوع virtual، وهذا يسبب بعض البطء في العمل.

تكمن روعة الـــ Upcasting في أنه استطاع توفير الكثير من التعب على المبرمج عندما استطاع التعامل مع جميع الأبناء (في المساحة المشتركة بينهم) عن طريق الأب، فتخيل أنني لولا هذه الميزة سأضطر إلى تعريـــف method جديدة في الصف Man كلما أردت أن أضيف نوع فواكه جديد إلى مجموعتي، ولكن مع وجود الـــ Upcasting لن أقترب مطلقاً من الصف Man ولن أعدل عليه مهما زدت من أبناء للصف Fruit.

ملاحظات:

١. نلاحظ أننا لا نستطيع معرفة أي تابع eat() سيستدعى أثناء الـــ (compile time) وإنما يحدد هذا أثناء تنفيذ البرنامج (Run-time) وهذا ما يجعل استدعاء التوابع في Java يصنف على أنه استدعاء ديناميكي، ويدعى: (**Late Binding** أو **Dynamic Binding** أو **Run-time Binding**).

٢. جميع التوابع في Java تستدعى بالطريقة السابقة عدا التوابع الـــ (static) و الـــ (final)، إذ أنها لا تقبل الـــ Overriding في الأساس، وسنوضح هذه الفكرة في المحاضرة القادمة.

٣. Overriding private methods:

هناك فخ برمجي خطير يقع فيه المبرمجون وهو إعادة تعريف الـــ private methods، إذ أن مفهـــوم إعادة التعريف (Overriding) إنما يكون على القسم المتاح للمبرمج من الصف الابن، أي أنه لا معنـــى لإعادة تعريف تابع لا أستطيع الوصول إليه أصلاً، لكن المشكلة أن الـــ compiler لا يعترض على إعادة تعريف الـــ private method، ويعتبرها method جديدة في الابن، لذا عند استدعائها قد تظهر نتائج غير متوقعة.

مثال:

```
public class PrivateOverride
{
	private void f() {
		System.out.println("private f()");
	}

	public static void main(String[] args) {
		PrivateOverride po = new Derived();
		po.f();
	}
}

class Derived extends PrivateOverride
{
	public void f() {
		System.out.println("public f()");
	}
}
```


حسب ما رأينا قبل قليل سنتوقع أن يكون الخرج:

```
public f()
```

ولكن الخرج الحقيقي لهذا البرنامج هو:

```
private f()
```

٤. ذكرنا سابقاً أن للوراثة نوعين، وإن طريقة الـــ Upcasting تؤتي ثمارها تماماً عندما تكون الوراثة مـــن النوع الأول (is-a)، إذ لا يضيف الصف الابن أي حقل أو تابع إلى ما ورثه عن أبيه، وإنما يعيد تعريـــف التوابع فقط، في هذه الحالة يكون الـــ Upcasting فعالاً للغاية، ويمكن استخدامه على جميع الأبناء بغض

النظر عن نوع الـــ object لأن جميع الأبناء تشترك في جميع الصفات ولسنا بحاجة لمعرفة أي معلومات إضافية عن الأبناء أكثر مما يتيحه لك المؤشر على الأب.

٥. أما إذا كانت الوراثة من النوع الثاني (is-like) فإن الـــ Upcasting يقيد المبـــرمج، إذ أنـــه لا يـــسمح برؤية جميع توابع وحقول الابن، وإنما يتيح ما يشترك منها مع الأب فقط، فإذا كانت هذه المتاحـــة كافيـــة لعمل البرنامج كان بها، وإلا فسنضطر إلى التقنية الثانية: Downcasting.

٦. سلوك التوابع المعاد تعريفها في الأبناء، ضمن بواني الآباء:

لندرس معاً المثال الآتي:

```
class Fruit
{
	void eat() {}
	Fruit () {
		System.out.println("Fruit() before eat()");
		eat();
		System.out.println("Fruit() after eat()");
	}
}

class Apple extends Fruit
{
	private int color = 1;
	Apple (int c) {
		color = c;
		System.out.println("Apple.Apple(), color = " + color);
	}
	void eat() {
		System.out.println("Apple.eat(), color = " + color);
	}

	public static void main(String[] args) {
		Apple c = new Apple(5);
	}
}
```


لنتابع التنفيذ خطوة خطوة:

عندما عرفنا object من الصف Apple وقبل استدعاء الـــ constructor الخاص به، كان لابد أن يستدعى constructor الأب كما اتفقنا في المحاضرة ٤، لذا انتقل التنفيذ إلى constructor الأب (Fruit) وطبـــع العبارة الأولى: (Fruit () before eat()).

بعدها استدعي التابع eat()، وبما أنه قد عمل له (override) في الصف Apple سيـــستدعى تـــابع eat() للصف Apple والذي يطبع قيمة اللون (color)، **فماذا نتوقع أن تكون قيمة الـــ color؟**

سيتابع بعدها تنفيذ باني الأب ويطبع (Fruit () after eat()).

ومن ثم ينتقل إلى الباني الخاص به ويعطي القيمة ٥ للـــ color، وبالتالي يطبع: (Apple.Apple(), color = 5).

إن خرج البرنامج السابق هو:

```
Fruit() before eat()
Apple.eat(), color = 0
Fruit() after eat()
Apple.Apple(), color = 5
```

نلاحظ أن قيمة color عندما طبع من باني الأب كانت تساوي الصفر على غير التوقع، ولكن لماذا؟؟
لماذا لم تكن قيمة color تساوي الـــ ١ علماً أننا أسندنا له هذه القيمة مباشرة بعد تعريفه (initialization)؟
الفكرة الهامة هنا أن constructor الأب يعمل قبل أن ينفذ الـــ (initialization) لحقول الصف الابن، وبما أن الحقول في الصفوف تعطى قيماً ابتدائية صفرية بمجرد تعريفها، لاحظنا أن قيمة color كانت ٠ وليس ١.
مثل هذا الـــ bug يعتبر صعب الكشف لأننا نعتقد أن كل شيء يسير على ما يرام، ولن نتوقع مثل هذا التصرف للبواني، لذا ينصح بعدم استدعاء أي تابع في الـــ constructor يتوقع أن يعمل له (override) في الأبناء.

٢. Downcasting:

في التقنية السابقة (Upcasting) كنا نصعد من الأبناء إلى الآباء، أما في هذه التقنية فـــالأمر معكـــوس، إذ أن علينا معرفة نوع الابن الذي يشير إليه المؤشر من نوع الأب.
سنوضح الفكرة من خلال مثال:
لنفرض أن لدينا class اسمه Shape، ونريد أن نشتق منه الأشكال التالية Rectangle, Circle ..
تشترك هذه الصفوف مع أبيها في تابع الرسم draw() إلا أنها تختلف في بعـــض خصائـــصها إذ أن الـــدائرة تحتوي نصف قطر، بينما يحوي المستطيل طولاً وعرضاً..
ليكن لدي برنامج رسومي يستخدم الصفوف السابقة ونريد أن ننشئ Array نضع فيها جميع الأشكال المرسومة على الشاشة.
حسب ما تعلمنا قبل قليل (Upcasting) نستنتج أن الـــ Array يجب أن تكون من النمط Shape ومـــن ثـــم نملؤها حسب الحاجة إما بـــ Rectangle أو بـــ Circle، ويمكن استخدام تابع واحد نرسم فيه جميـــع هـــذه الصفوف إذ أنها جميعاً تعيد تعريف تابع الرسم.

```
Shape[] sa = {new Circle(), new Rectangle()};
```

ولكن ماذا لو أردنا استرجاع object بعينه (وليكن الدائرة الثالثة مثلاً) ونجري عليه بعض العمليات الخاصـــة بالصف Circle؟
جميع الـــ objects في المصفوفة يشير إليها مؤشرات من النوع Shape، فكيف سأســـترجع الـــ object Circle من المؤشر Shape؟
سنقوم هنا بعملية casting من النمط Shape إلى النمط Circle وتتم ببساطة كما في المثال التالي:

```
Shape sh = new Circle();
Circle c = (Circle)sh;
```

نلاحظ أننا استرجعنا الـــ object من نوع Circle ووضعناه في reference من نفس نوعه.

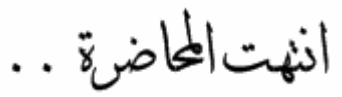
لكن العملية ليست دوماً بنفس البساطة..

لنعد إلى الـــ Array التي تحدثنا عنها قبل قليل، والتي تحوي عدة أنواع من الأشكال، ما الذي سيدلني على أن العنصر الرابع مثلاً من المصفوفة يشير إلى object من النوع Circle؟؟
تصور أنه كان يحوي object من النوع Rectangle، وقمنا بعمل casting إلى النمط Circle، سينتج لدينا ما يعرف بـــ (Exception) وهو خطأ يحدث أثناء Run-time.

نستنتج أن عملية الـــ Downcasting ليست آمنة دوماً كما كان الحال في الـــ Upcasting، وعلينا أن نوجد طرقاً نستطيع من خلالها معرفة نوع الـــ object قبل عمل casting.
هناك كلمة محجوزة هي (instanceof) يمكن استخدامها لمعرفة نوع object ما كالتالي:

```
if (i instanceof Circle)
```

إن عملية فحص نوع الـــ object تدعى run-time type identification (RTTI) وهي مفصلة في المرجع في البحث العاشر كاملاً، فأرجو أن يقوم الدكتور بشرحه وإن لم يفعل فأرجو أن يسعفني الوقت للقيام بهذه المهمة..



lectures_team@hotmail.com