المحاضرة الخامسة 13

مقدمة:

هذه المحاضرة عبارة عن عدة مفاهيم متفرقة ولكنها مهمة جداً، كما أنها تحوي الكثير من الدقائق (معلومات صغيرة وهامة)، لذا أرجو التركيز عليها جيداً..

Array:

على الرغم من أن الدكتور لم يتكلم عن الـ Array في محاضراته إلا أنني لمست ضرورة الحديث عنها، وقد أكد رأيي هذا أن وظيفة العملي تحتاج إلى الـ Array، لذا أستأذنكم بالحديث عنها، ولمن يريد التوثق من هذا البحث أنصحه بالعودة للمرجع في البحث الرابع تحت عنوان: (**Array initialization**).

كنا قد تعلمنا في PASCAL كيف نعرف Array بشكل مباشر، ولكننا سعدنا كثيراً عندما انتقلنا إلى C++ وتعلمنا كيف نحجز مصفوفة ديناميكياً، إلا أننا شعرنا بالارتباك عندما لمسنا صعوبة التعامل مع الـ Array في C++ وخصوصاً مشاكل المؤشرات..

المفاجأة أن Java قدمت الـ Array بطريقة مريحة جداً وعملية جداً بحيث تجمع كل الميزات وتتجنب كل المشاكل السابقة.

جرياً على عادة الـ Java في أن جميع عناصر اللغة هي objects، فإن الـ Array لا تشذ عن القاعدة، وبالتالي يمكن اعتبار المصفوفات في Java عبارة عن objects ولكن لها طريقة معاملة خاصة، وبالتالي يمكن استنتاج الأمور التالية:

- حجز Array يكون عن طريق تعليمة new.
- لسنا مسؤولين عن حذف الـ Array إذ أن هذه المهمة سيقوم بها الـ gc.
- جميع المصفوفات في Java ديناميكية، وبالتالي يكمن تحديد طول المصفوفة في وقت الـ Compile أو في وقت الـ Runtime، بدون أي مشاكل.

كيف نعرف Array؟

يمكن تعريف Array من أي نمط سواءً كان (primitive type) أو (reference) كما يلي:

(Array type)[] *(Array name)*;

كما يمكن أن نضع الأقواس المربعة بعد اسم المصفوفة كما يلي:

(Array type) (*Array name)*[];

مثال:

```
int[] x;          // you can do this
int x[];          // and you can do this
```

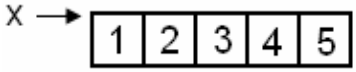
في المثال السابق قمنا بتعريف reference على Array ولكننا لم نقم بحجز الـ Array فعلياً في الـذاكرة، ولا يجوز أن نضع أي رقم بين القوسين السابقين كما كنا معتادين في الـ C++ وهذا يؤكد أن حجز المصفوفات ديناميكي دوماً، وإنما يتم حجز المصفوفة بطريقتين:

١. عن طريق ملء الـ Array مباشرة دون تحديد طولها، والذي يحدد طولها هو عدد العناصر التي وضعناها بداخلها:

مثال:

```
int[] x = { 1, 2, 3, 4, 5 };
```

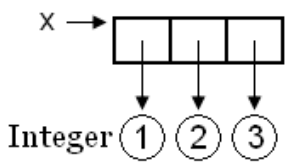
في هذا المثال عرفنا مصفوفة من int ووضعنا بداخلها ٥ عناصر وبالتالي سيحجز طول هذه المصفوفة في الذاكرة ٥ بشكل تلقائي، كما في الشكل:



مثال:

```
Integer[] x = {new Integer(1), new Integer(2), new Integer(3)};
```

أما في هذا المثال فقد عرفنا مصفوفة من references، وقمنا بإنشاء ٣ objects مباشرة وبالتالي سيحجز طول هذه المصفوفة في الـذاكرة ٣ بشكل تلقائي، كما في الشكل:



٢. ماذا لو لم نكن نعلم عدد عناصر المصفوفة أثناء كتابة البرنامج؟

يمكننا أن نحجز المصفوفة عن طريق تعليمة new وبالتالي يمكننا تحديد طولها أثناء تنفيذ البرنامج أي في وقت الـ Runtime.

```
int[] x = new int [10];
```

كما يمكن أن نضع أي متحول مكان الـ ١٠ في المثال السابق بشرط أن يكون من نوع صحيح، ويمكن أن يكون هذا المتحول خرج تابع ما:

```
int i = 10;
int[] x = new int [i];
```

عند تعريف مصفوفة بالطريقة السابقة فإن Java تقوم بإعطاء قيم ابتدائية لعناصر هذه المصفوفة بحسب نوع هذه العناصر، فإن كانت references ستكون قيمها null، وإن كانت primitives ستكون قيمها بحسب نوعها *كنا قد وضعنا جدولاً يوضح هذه القيم في الصفحة ٧ من المحاضرة ٢*.

كيف نتعامل مع الـ Array؟

- يمكن الوصول لعناصر المصفوفة بالطريقة المعتادة (x[i]).
- يبدأ ترقيم عناصر المصفوفة من الصفر.
- بما أن المصفوفات في Java عبارة عن objects، فلكل مصفوفة عدة خصائص يمكن الوصول إليها بكتابة اسم المصفوفة وإتباعه بنقطة، ومن أهم هذه الخصائص خاصة length:

```
int[] x = int new[10];
int i = x.length;        // i == 10
```

- إذا اضطررنا لتوسيع المصفوفة أثناء تنفيذ البرنامج فعلينا أن ننشئ مصفوفة جديدة ومن ثم ننقل عناصر القديمة إلى الجديدة عن طريق حلقة for، ولا ننسى هدم المصفوفة القديمة عن طريق إسناد null إليها وترك الـ gc يقوم بمهمته في هدمها.

```
x = null;
```

- ماذا يحدث عند تنفيذ الكود التالي:

```
int[] x = {1, 2, 3};
int[] y = {4, 5};
x = y;
```

قد نظن للوهلة الأولى أن قيم المصفوفة y قد نسخت إلى المصفوفة x، ولكن هذا الكلام خاطئ تماماً إذ أن ما حدث فعلاً هو أن المؤشر x أصبح يشير إلى نفس المصفوفة التي يشير إليها y، وبقيت المصفوفة الأولى بدون أي مؤشر وبالتالي سيقوم الـ gc بهدمها.

المصفوفات ذات الأكثر من بعد (Multidimensional Arrays):

يمكن تعريف مصفوفات بعدد الأبعاد الذي نشاء كما يلي:

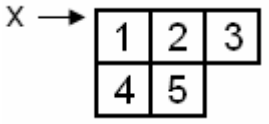
```
int[][] x;
int[][][] y;
```

ويكون الحجز بإحدى الطرق التالية:

```
int[][] x = {{1,2,3},{4,5,6}};
```

المصفوفة السابقة تشبه جدولاً بـ ٣ أسطر و ٣ أعمدة.

```
int[][] x = {{1,2,3},{4,5}};
```

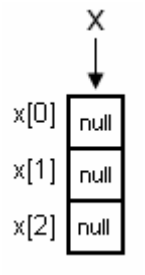


نلاحظ أن السطرين لم يعودا متساويين وهذا جائز في Java.

```
int[][] x = new int[4][5];
```

المصفوفة السابقة تشبه جدولاً بـــ ٤ أسطر و ٥ أعمدة.

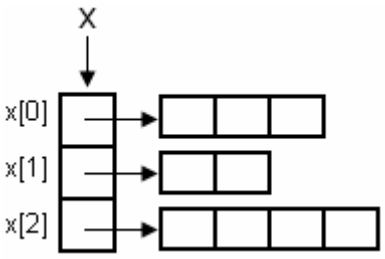
```
int[][] x = new int[3][];
x[0] = new int[3];
x[1] = new int[2];
x[2] = new int[4];
```



المصفوفة السابقة تعرف باسم (Jagged Array) وتوضح لنا عدة مفاهيم:

نستطيع تخيل هذه المصفوفة على أنها مصفوفة ذات بعد واحد، عناصرها أيضاً مـــصفوفات ذات بعد واحد، لذا فإن التعليمة الأولى أدت إلى حجز بعد واحد من أبعاد هذه المصفوفة، كل عنصر من عناصره عبارة عن مؤشر إلى مصفوفة ولكن قيمته حالياً تساوي (null).

حتى نستطيع بناء البعد الثاني للمصفوفة يجب أن نمر على كل عنصر من عناصـــر البعـــد الأول وننشئ له مصفوفته الأحادية الخاصة به، وبما أننا تعاملنا مع كل سطر علـــى أنه مصفوفة أحادية، نستنتج أننا نستطيع الحصول على طول كل سطر عـــن طريق التعليمة (x[i].length)، وهـــذا يعنـــي أن التعليمـــة (x.length) ستعطينا عدد أسطر المصفوفة (أي بعد الشعاع الذي أنشأناه في البداية)، وهذا يعني أننا نستطيع التجول في هذه المصفوفة عن طريق الحلقتين التاليتين:




```
for (int i = 0; i < x.length; i++) {
      for (int j = 0; j < x[i].length; j++) {
            System.out.println(x[i][j]);
      }
}
```


String:

تكلمنا كثيراً عن الصف String ولكن لابد لنا من مراجعة ما قد ذكرناه والتوسع في هذا الموضوع:

- الصف String موجود ضمن مكتبة تدعى (java.lang) هذه المكتبة تضمن تلقائياً في أي ملف (.java).
- يمكن معاملة الـــ String كصف وبالتالي إنشاء أغراض منه عن طريق تعليمة new، كما يمكن معاملته بطريقة مشابهة للـــ primitives حيث ننشئ أغراضاً منه بشكل مباشر:

```
String s1 = new String("Ammar");
String s2 = "Ammar";
```

- الطريقة الأولى للتعريف تتيح للمبرمج الاستفادة من تعدد البواني المعرفة للصف String، أمـــا الطريقـــة الثانية فتجبر المبرمج على طريقة واحدة في الإسناد.

- يوجد للصف String توابع عديدة سنذكر بعضها:
  - ❖ length(): يعيد طول الـ String.
  - ❖ charAt(): يتيح التعامل مع الـ String وكأنه مصفوفة char حيث يأخذ متحول (int i) كدخل ويعيد المحرف ذي الدليل i من الشريط المحرفي.
  - ❖ equals(): بما أن الـ String صف، فإن مقارنة two Strings عن طريق العملية (==) إنما هي مقارنة مؤشرات وهذا الكلام ينطبق على جميع الصفوف، ولمقارنة two Strings مقارنة حقيقية يجب استخدام التابع equals().
  - ❖ equalsIgnoreCase (): يقوم بنفس عمل التابع السابق مع تجاهل حالة الأحرف (كبيرة أو صغيرة).
- هناك توابع عديدة وكثيرة لا مجال لذكرها هنا، لذا أنصح زملائي بالعودة إلى دليل لغة Java (Help) للاطلاع على البواني والتوابع المتعددة التي يملكها الصف String.
- يمكن عمل concat بين عدة سلاسل محرفية عن طريق العملية (+):

```
String firstName = "Ammar", lastName = "Sarsar";
String name = firstName + " " + lastName;
```

- يمكن عمل concat بين String وأنماط أخرى (primitives)، وستكون النتيجة كلها على شكل String:

```
int i = 5, j = 9;
System.out.println("Text " + i + j);
System.out.println(i + j + " Text");
```

ماذا تتوقع أن ناتج تنفيذ التعليمات السابقة؟

الناتج سيكون كالتالي:

```
Text 59
14 Text
```

التفسير هو أن عملية الـ concat في تعليمة الطباعة الأولى جاءت بعد الـ String، لذا تم تحويل (i,j) أولاً إلى String ومن ثم تم الدمج.

أما في التعليمة الثانية فقد عوملت (i,j) على أنها متحولات int فتم حساب قيمتها أولاً بعملية جمع طبيعية ثم تم تحويلها إلى String..

- يمكن التحويل من نمط String إلى أنماط primitives كالتالي:

```
int i = Integer.parseInt("100");
float f = Float.parseFloat("100.5");
```

نستنتج أنه للتحويل من String إلى أي primitive type يجب أن نستخدم تابع (parse) وهو تابع static موجود في كل صف (Wrapper type)، وبما أن هذه التوابع static فبإمكاننا استدعاؤها دون الحاجة لعمل object من هذه الصفوف.

- وبالمقابل، للتحويل من primitives إلى String نستخدم التابع valueOf():

```
String s = String.valueOf(345);
```

- هناك نقطة أساسية جداً في الصف String وهي أن السلسلة المحرفية غير قابلة للتعديل مطلقاً.

معك حق في أن تستغرب من هذا الكلام، وقد تعطيني مثالاً لتثبت لي خطأ ما أقول:

```
String s = "ABC";
s = s + "DE";
System.out.println(s);
```

إن ناتج تنفيذ التعليمات السابقة هو:

```
ABCDE
```

ما المقصود إذاً بأن الشريط المحرفي غير قابل للتعديل؟

الحقيقة أن عملية الجمع السابقة لم تضف المحرفين "DE" إلى الشريط المحرفي الخاص بالـ object الذي يشير إليه المؤشر s، ولكنها أنشأت object جديد وضعت فيه الشريط (ABCDE) كاملاً وهدمت الـ object القديم.

ستقول لي: أين المشكلة؟؟ فعلاً لا مشكلة حتى الآن، ولكن المشاكل العظيمة تظهر عندما يكون لدينا حلقة تدور ١٠٠٠ مرة مثلاً وفي كل لفة تجري عملية concat على الـ String، تخيل أننا أنشأنا ١٠٠٠ object جديد خلال هذه العملية، وتركنا ١٠٠٠ object قديم ليتسلى الـ gc بهدمها!!

وبالتالي استهلكنا وقتاً كبيراً بلا أي مبرر.

الحل يكمن من خلال الصف (StringBuffer).

- الصف (StringBuffer) عبارة عن صف شبيه بالصف String ولكنه يقبل عمليات التعديل من خلال التابع (append) والإضافة من خلال التابع (insert) وغيرها ..

```
String s = "0";
StringBuffer buf = new StringBuffer(s);
for(int i=1; i<10; i++){
      buf.append(i);
}
s = buf.toString();
System.out.println(s);
```


إن خرج البرنامج السابق هو:

```
0123456789
```

toString():

ذكرنا مسبقاً أن جميع الصفوف في Java مشتقة من الصف Object، وذكرنا أن هذا الصف يحوي مجموعة من التوابع التي يفضل أن يعيد المبرمج تعريفها في جميع صفوفه.

من هذه التوابع: toString()

إذا فكرنا ببنية الـ classes، سنلاحظ أنه قلما يخلو class من تابع print، وبالتالي لماذا لا يكون لدينا تابع موجود في جميع الصفوف ويقوم بمهمة print؟

مثال:

في الصف (Student) غالباً سأحتاج إلى طبعة اسم هذا الطالب..

أمنت Java التابع ()toString كي نعيد تعريفه (override) ونضع فيه عبارة ما (String) يهمني أن تعبر عن تابع print لصفي.

مثال:

```
public class Student
{
	String firstName, lastName;

	Student(String firstName, String lastName){
		this.firstName = firstName;
		this.lastName  = lastName;
	}

	public String toString(){
		return firstName + " " + lastName;
	}

	public static void main(String[] args) {
		Student stu = new Student("Ammar", "Sarsar");
		System.out.println(stu);
		String s = "Student Name: " + stu;
		System.out.println(s);
	}
}
```


نلاحظ أن هناك موضعين غريبين في المثال السابق:

١. نلاحظ أننا استدعينا إجرائية الطباعة والتي تأخذ String ومررنا لها object من نوع Student..

ولكن كيف حدث هذا؟ وما الخرج المتوقع من هذا البرنامج؟

الخرج هو:

```
Ammar Sarsar
```

٢. نلاحظ أننا أجرينا عملية concat بين شريط محرفي وobject من نوع Student، ما الخرج المتوقع من هذا البرنامج؟

الخرج هو:

```
Student Name: Ammar Sarsar
```

نستنتج أن الـ compiler يستدعي تابع ()toString لأي class في حالتين:

a. عندما نطلب طباعة الـ object عن طريق التعليمة (System.out.println()) أو التعليمة (System.out.print()).

b. عندما نعمل concat للـ object مع متحول String أو مع شريط محرفي (" .. ").

ملاحظات:

١. لا يستدعي الـ compiler تابع ()toString إلا في الحالتين السابقتين، وليس كلما استخدمنا الـ object على أنه String، وكمثال على ذلك: إذا مررنا object من نوع Student إلى تابع يأخذ

argument من نوع String فلن يطلب الـ compiler تابع ()toString، وإنما سيعطي خطأ compile.

٢. لنفترض أننا استخدمنا الـ object في الحالتين السابقتين دون أن نعرف تابع ()toString له، ماذا يحدث؟

في الحقيقة سيبحث الـ compiler عن تابع ()toString في أب هذا الـ object إذا كان له أب مباشر، فإن لم يجد سيبحث في أبيه وهكذا.... فإن وصل إلى class ليس له أب ولا يحوي تابع ()toString سيقوم باستدعاء تابع ()toString المعرف في الصف object، وفي هذه الحالة سيقوم بطباعة عدة معلومات ليست ذات أهمية مثل اسم الـ class ورقم Hash...

وهذه هي التعليمة التي ينفذها:

```
return getClass().getName() + "@" + Integer.toHexString(hashCode());
```

Main Method:

استخدمنا كثيراً في المحاضرات السابقة الـ main method، وكنا دائماً نتعامل معها دون أن نحيط تماماً بالـ argument التي تأخذها، لذا سنتذكر شكلها ونوضح عمل هذا الـ argument:

```
public static void main (String[] args){

}
```

ذكرنا سابقاً الحكمة من كون هذه الـ method (public و static)، وسنتكلم الآن عن نقطتين هامتين:

١. في الفحص يجب الانتباه: إذا وجدنا صفين كل منهما يحوي main method فكيف نعرف أين يبدأ البرنامج؟

في الحقيقة سيبدأ البرنامج من الصف الـ public.

٢. نلاحظ أن main method تأخذ مصفوفة String كدخل، ولكن ما استخدام هذه المصفوفة؟

لنشرح الفكرة عن طريق مثال:

ألم تسأل نفسك يوماً كيف يعمل برنامج javac مثلاً؟

عندما نفتح محرر Dos ونكتب javac، فإننا في الحقيقة نشغل برنامجاً، والسؤال هو:

كيف سنمرر لهذا البرنامج مسار ملف الـ java الذي سيقوم بترجمته؟

اعتدنا على أن نكتب هذا المسار بعد اسم البرنامج وهذا ما يعرف بـ (commend line).

مهمة المصفوفة args أن تخزن جميع الـ Strings التي ندخلها للبرنامج بهذه الطريقة ومن ثم يتعامل معها البرنامج بطريقة طبيعية جداً على أنها مصفوفة String.

نستطيع إدخال عدد المتحولات الذي نشاء، وذلك بوضع (فراغ space) بينها.

مثال:

ليكن لدينا البرنامج التالي:

```
public class Test
{
      public static void main (String[] args){
            for(int i=0; i<args.length(); i++)
                  System.out.println(args[i]);
      }
}
```

سنشغله عن طريق محرر dos كما يلي:

```
C:\>Test Hello Shabab
```

سيكون خرج البرنامج:

```
Hello
Shabab
```

The final keyword:

تعودنا في لغات البرمجة التي تعلمناها على وجود مفهوم الثوابت (const)، وكنا نستخدم الكلمة المحجوزة (const) في كل من PASCAL و C++.
في Java الكلمة (const) محجوزة أيضاً وكنها لا تستخدم أبداً، ويستعاض عنها بالكلمة (final) مع بعض الاختلاف في المعنى والاستخدام.

لدينا ٣ استخدامات للكلمة final:

١. final data:

الحقل أو المتحول إن كان final فهذا يعني أحد أمرين:

- إذا كان primitive: فهذا يعني أن قيمته ثابتة لا يجوز تغييرها أبداً خلال تنفيذ البرنامج.
- أما إذا كان reference: فهذا يعني أن المؤشر هو الثابت وليس الـ object الذي يشير عليه، وهذا يعني أنني لا أستطيع إسناد أي object آخر إلى هذا المؤشر ولكنني أستطيع أن أعدل على حقول الـ object وأن أستخدم توابعه بشكل طبيعي، كذلك المصفوفة لا أستطيع إسناد مصفوفة أخرى لنفس المؤشر ولكنني أستطيع أن أغير في حقولها لأنها في النهاية object، ولا توجد طريقة في Java لجعل الـ object ثابتاً.

في كل من Pascal و C++ كانت قيمة الـ const تعطى له بعد تعريفه مباشرة، أي في وقت الـ compile ولم نكن نستطيع أن نعطيه قيمته الابتدائية أثناء الـ Runtime، لكن الأمر اختلف في Java حيث صار للمبرمج مطلق الحرية في عمل (Initialization) متى شاء للمتحول أو الحقل الـ final وذلك سواء كانت القيمة معلومة (٥ مثلاً) أو كانت ستعلم أثناء الـ Runtime (قيمة random مثلاً)، وبمجرد حصوله على قيمته الابتدائية لن نستطيع التعديل عليه أبداً.

هناك عرف سائد بين مبرمجي Java يقتضي أن تكون تسمية الحقول الـ final سواء كانـت static أو غير static على النحو التالي:

- إذا كنا سنسند لها قيمة ابتدائية معروفة أثناء الـ compile time فتكون التسمية بالأحرف الكبيرة ويكون الفاصل بين الكلمات في التسمية الواحدة هو (_).
- أما إذا كانت القيمة الابتدائية مجهولة في وقت الـ compile فتكون التسمية طبيعية وتدعى عندها بـ (Blank finals).

مثال(I):

```
import java.util.*;

class Temp
{
      int i;
}

public class FinalData
{
      private Random rand = new Random();

      final int VAL_ONE = 4;
      static final int VAL_TWO = 8;
      final int i1 = rand.nextInt(20);
      static final int i2 = rand.nextInt(20);
      final int i3;
      final String STR_VAL = new String("A");
      final String s1;
      final int[] a = { 1, 2, 3, 4, 5, 6 };
      final Temp t = new Temp;

      public FinalData(final String s, int i) {
            i3 = i;
            s1 = s;
            a[3] = 7;
            t.i = 100;
      }
}
```


ملاحظات على المثال(I):

- نلاحظ أننا طبقنا قاعدة التسمية التي ذكرناها آنفاً في هذا المثال فجميع المتحولات الـ final التي أسندنا لها قيمة مباشرة كانت أسماؤها ذات أحرف كبيرة.
- يمكن التعامل مع final array كما لاحظنا بشكل طبيعي ولكن لا يجوز إعادة حجزها مرة أخرى ولا إسناد المؤشر a إلى مصفوفة أخرى.
- نلاحظ أننا تعاملنا مع الغرض t بشكل طبيعي وعدلنا قيمة أحد حقوله، ولكـن لا يمكننـا أن نـسند أي object جديد إلى المؤشر t ولا أن نستخدم معه تعليمة new مرة أخرى.
- بما أن الـ String غير قابل للتعديل كما وضحنا في هذه المحاضرة، وإنما يقوم كل تعـديل بإنـشاء object جديد، هذا يعني أن الـ final String لا يمكن تعديله أبداً، على الرغم من أنه object.

- يمكن حماية الـــ arguments المدخلة إلى method ما من التغيير وجعلها Read only عن طريق وضع كلمة final قبلها، كما في الباني في المثال السابق.

٢. ***final methods:***

إن تعريف method على أنها final له سببان:

١. إن هذا التعريف يحمي الـــ method من أن تغير أبداً، أي أنه يمنع الـــ classes المشتقة من هذا الـــ class من عمل (Overriding) لهذه الـــ method، وبالتالي فإن أي استدعاء لهذه الـــ method هو فعلاً استدعاء لها بالذات وليس لأي method أخرى.

٢. بما أن هذه الـــ method ثابتة فهذا يتيح لـــ Java Compiler أن يستفيد من هذه الميزة ويحول استدعاء هذه الـــ method من استدعاء ديناميكي إلى (inline).

ما هو التابع الـــ *inline* ؟

هو تابع لا يستدعى استدعاءً وإنما تستبدل تعليمة الاستدعاء بلصق code التابع في مكان الاستدعاء أي كأنه أصبح جزءاً من التابع المستدعي، مما يوفر وقت الاستدعاء وحفظ المتحولات في الـــ stack..

هناك نقطة أخيرة وهي أن أي private method هي ضمنياً final method لأنه لا معنى لعمل (Overriding) لـــ private method كما وضحنا في المحاضرة السابقة، كما أن أي static method هي أيضاً final، وإن إضافة كلمة final قبل اسمي هذين النوعين لا يعطي أي معنى جديد.

٣. ***final classes:***

إن جعل الـــ class من النوع final يعني أننا نمنع أي class آخر من الوراثة من هذا الـــ class، وبالتالي ستصبح جميع الـــ methods في هذا الـــ class من النوع final، ولكن الحقول لن تتأثر وستبقى بشكلها المعتاد (أي أنها ليست final).

معلومات إضافية:

الفقرات القادمة ليست مهمة جداً للفحص، ولكنها مهمة لمعلومات مهندس المعلوماتية:

garbage collector:

هناك نقطة هامة في طريقة عمل الـــ gc اتضحت لنا من خلال التجربة، ومن خلال ملاحظات الزملاء:

لندرس المثال التالي بدقة:

```
public class A
{
	int id;
	public A (int id){
		this.id = id;
	}
	public void finalize(){
		System.out.println("End " + id);
	}
}

public class B
{
	public void test(){
		A a1 = new A(1);
		{
			A a2 = new A(2);
			new A(3);
		}
		// System.gc();                (I)
	}

	public static void main(String[] args){
		B b = new B();
		b.test();
		// System.gc();                (II)
	}
}
```


نحن نعلم أن استدعاء الـ gc سيؤدي إلى هدم جميع الـ objects غير المستخدمة ولكنه يستدعي تابع finalize() لها قبل هدمها.

لو أننا سمحنا للبرنامج باستدعاء الـ gc في الموقع (I) فقط لكان خرج البرنامج:

```
End 3
```

ولو أننا سمحنا للبرنامج باستدعاء الـ gc في الموقع (II) فقط لكان خرج البرنامج:

```
End 1
End 2
End 3
```

التفسير:

ظننا في البداية أن الخروج من الـ scope الداخلية التي استخدمناها في التابع test() لا يكفي لهدم المؤشر a2، ولكن توضح لنا أن مؤشر سيهدم فعلاً عند الخروج من الـ scope ولكن هذه العملية تستغرق بعض الوقت، وبالتالي فإن التعليمة التالية (System.gc();) ستنفذ قبل أن يتم حذف المؤشر، لذا لن يشعر الـ gc بأن الـ object المؤشر عليه بالمؤشر a2 أصبح غير مستخدم، ولكن الزمن المستغرق للعودة من استدعاء التابع إلى الـ main كاف لهدم المؤشر، لذا فإن استدعاء الـ gc في الموضع الثاني أدى إلى هدم الـ object.

ولكن الوضع مختلف بالنسبة للـ object الذي أنشأناه بدون مؤشر إذ أنه بمجرد إنشائه أصبح غير مستخدم، فتحسس الـ gc ذلك وقام بهدمه مباشرة.

ماذا عن توقيت عمل الـــ gc؟

ذكرنا أن الـــ gc لا يعمل إلا عند الحاجة، فلا شيء يضمن لنا أبداً أن يعمل قبل نهاية تنفيذ البرنـــامج، والـــذي يحدث عند نهاية البرنامج هو أن الـــ JVM يحرر جميع الذاكرة المحجوزة للبرنامج دفعة واحدة بدون أن يعذب نفسه ويضيع الوقت باستدعاء الـــ gc، لذا سنصل إلى نتيجة مهمة جداً وهي:

**** لا يوجد أي شيء يضمن لنا عمل الـــ gc ****

كيف يعمل برنامج Java؟

عند بدء إقلاع برنامج Java يعمل ما يسمى بـــ (class loader) ويقوم بتحميل المكتبات الضرورية لعمـــل البرنامج، ومن ثم يتم عمل check للكود عن طريق ما يعرف بـــ (JVM verifier)، ومن ثم ينفذ الـــ byte code حسب تقنية الـــ JIT (Just in time).

كما نعلم، لاشيء يمنع برنامج الـــ Java من استخدام مكتبات بعيدة، وكلما أراد أن يستعمل صفوف من مكتبـــة ما، فمن واجبه عمل check مرة أخرى للكود مما يسبب بطء شديد في التنفيذ، ولكن هذا الكلام كان صحيحاً في النسخ القديمة من الـــ JVM أما النسخ الحديثة فأصبحت تعتمد تقنية الـــ JVM والتي تقوم على أساس عمـــل check مرة واحدة فقط للكود خلال فترة حياة البرنامج، وتكون هذه المرة عند طلب استخدام صف من مكتبة ما لأول مرة، أما المرات القادمة فيصبح التنفيذ مباشر.

ملاحظة:

إن الـــ JVM يعمل عند تشغيل تطبيق ما لـــ Java، وينتهي تشغيله عند انتهاء آخر تطبيق Java يعمل على الجهاز.

ملاحظة أخيرة:

الفرق بين JRE و JDK أن الأداة الأولى تقوم فقط بتشغيل تطبيقات Java، أما الأداة الثانية فتسمح بتطـــوير (برمجة) وتشغيل تطبيقات Java، وبالطبع كل منهما تحوي JVM.

انتهت المحاضرة . .