15

المحاضرة التاسعة

Object's containers

زملائي الكرام:

أعتذر بشدة عن تأخر نزول هذه المحاضرة ولكن ما باليد حيلة، وأرجو أن تكون هذه المحاضرة في المستوى المطلوب ليكون (* خِتَامُهُ مِسْك *).

سنتكلم في هذه المحاضرة عن بنى تخزين المعطيات في Java، حيث لا يخفى على أحد أهمية هذه البنى وخصوصاً بعد أن تسلينا بمادتي (الخوارزميات وبنى المعطيات 1،2)!!

لم نعد في Java مضطرين لبناء هذه البنى يدوياً كما كنا نفعل سابقاً وإنما لدينا الآن مجموعة كبيرة من بنى تخزين المعطيات المتميزة و التي نستطيع استخدامها مباشرة.

Arrays:

تعتبر المصفوفات أهم بنية تخزين معطيات وقد تكلمنا عنها سابقاً ووضحنا طريقة تعريفها والتعامل معها، وسنبحث اليوم في ميزات جديدة للمصفوفات..

- *ميزات المصفوفة:*
  1. فعاليتها: حيث تتميز بالسرعة، وسهولة الوصول إلى عناصرها عن طريق العملية [].
  2. النوع: حيث يمكن أن نعرف مصفوفة من النمط الذي نريده[1].
  3. سهولة التعامل مع الأنماط البسيطة (primitives).
- *سلبياتها:*

  مشكلة المصفوفات الكبرى هي عدم قابليتها للتوسع.

[1] وهذا ما لن نراه بعد قليل في بقية البنى

الصف Arrays:

يقدم لنا هذا الصف الموجود ضمن المكتبة (java.util)[1] مجموعة من التوابع الـ static والتي تقدم ميـزات للتعامل مع المصفوفات ومنها:

- Arrays.fill(): مهمة هذا التابع هي ملء المصفوفة كلها بنفس القيمة، إذ أنه يأخذ المصفوفة والقيمة التي نريد أن نضعها في جميع العناصر، وقد قدمت لنا Java عدة نسخ من هذا التابع كل نسخة مختصة بنـوع من أنواع المصفوفات الـ primitives، أما بالنسبة لمصفوفات الـ objects فيكفيها تابع واحـد يأخـذ Object (علل)، وسيتكرر هذا نفسه بالنسبة لجميع التوابع.
- Arrays.equals(): يقوم هذا التابع بمقارنة مصفوفتين من نفس النوع.
بالنسبة لمصفوفات الـ primitives فإن المقارنة طبيعية، أما مصفوفات الـ objects فإن التابع السابق يستدعي تابع equals() الخاص بكل object.
- System.arraycopy(): قدمت لنا Java تابعاً لنسخ المصفوفات ولكن هذا التابع موجود ضمن الصف (System) وليس ضمن الصف (Arrays)، وهو يقوم بنسخ أي مصفوفة -من أي نوع- إلى أي مصفوفة أخرى من نفس النوع، ولكن بشرط أن تتسع المصفوفة الأخرى للعناصر المنسوخة.

ملاحظات:

1. إن نسخ المصفوفات عن طريق هذا التابع أسرع من نسخها عبر حلقة ما، لأن هذا التابع يقوم بنسخ المصفوفة على أنها مجموعة من البايتات بغض النظر عن نوعها.
2. إن نسخ مصفوفة من objects هو نسخ مؤشرات فقط وليس نسخ objects.

- Arrays.sort(): قدمت لنا Java مجموعة من توابع فرز المصفوفات متمثلة في عدة نسخ للتـابع sort حيث يختص كل تابع بنوع من أنواع المصفوفات وسنتكلم عن هذه التوابع:
  - توابع فرز المصفوفات من النوع primitives: تفـرز هـذه التوابـع المـصفوفات الـصغيرة[2] بخوارزمية (Insertion sort) أما المصفوفات الكبيرة فتفرزها بخوارزمية (Quick sort).
  - توابع فرز المصفوفات من النوع object: تستخدم هذه التوابع خوارزمية (Merge sort).

ولكن يجب أن نتوقف هنا قليلاً لنتساءل: على أي أساس يتم المقارنة بين الـ objects عند الفرز؟
وهنا تبرز طريقتان للفرز:

---

[1] أي أننا حتى نستخدمه يجب أن نكتب: (import java.util.*;) أو (import java.util.Arrays;)

[2] التي لا يتجاوز طولها 7

1. عن طريق استخدام التابع compareTo() الموجود في الـ interface (comparable) وقد توسعنا في شرح هذه الفكرة في المحاضرة السادسة.
2. عن طريق التابع compare(): قد أرغب بأن أقدم بنفسي طريقة المقارنة التي أريدها لخوارزمية الفرز، لذا أستطيع أن أعرف صفاً ما يحقق الـ interface (comparator) وبالتالي سأضطر إلى إعادة تعريف التابع compare ضمنه وأن أمرره لتابع الفرز حيث سيقوم باستخدام التابع compare الموجود ضمنه للمقارنة بين الأغراض[1]، وبما أن هذه الفكرة مرت في الوظيفة فلن أعيد شرحها[2].

- Arrays.binarySearch(): يطبق هذا التابع خوارزمية البحث الثنائي على **مصفوفة مرتبة**[3] للبحث عن عنصر ما ويرد دليل العنصر ضمن المصفوفة.

لاحظنا في ما سبق أننا مضطرون لعمل نسخ overload من كل تابع من أجل جميع أنواع مصفوفات الـ primitives، بينما يكفي تابع واحد من أجل جميع الـ objects وهذا يعطينا انطباعين:

1. المرونة العالية التي حصلنا عليها كنتيجة لتطبيق الـ polymorphism (في حالة الـ objects).
2. لم تقدم Java طريقة مرنة للتعامل مع الـ primitives بدون أن نضطر لإعادة كل تابع أكثر من مرة، وفي الحقيقة نجد هنا أن C++ تتفوق على Java بتقنية الـ Template.

مثال:

```
import java.util.*;

class MyInt
{
    int value;

    MyInt(int value) {
        this.value = value;
    }

    public String toString(){
        return String.valueOf(value);
    }
}

public class CopyingArrays
{
    public static void main(String[] args) {
        MyInt[] x = new MyInt[10];
        MyInt[] y = new MyInt[5];
```


---

1 من المنطقي أن يكون هذا الصف (Inner class)

2 وهي مشروحة في المرجع لمن يحب أن يقرأها مع مثال واضح.

3 إذا لم تكن مرتبة فسنحصل على نتائج خاطئة

```
        Arrays.fill(x, new MyInt(10));
        Arrays.fill(y, new MyInt(5));
        System.out.println("x = " + Arrays.asList(x));
        System.out.println("y = " + Arrays.asList(y));

        System.arraycopy(y, 0, x, x.length / 2, y.length);
        System.out.println("x = " + Arrays.asList(x));

        y[2].value = 2;
        System.out.println("x = " + Arrays.asList(x));
        System.out.println("y = " + Arrays.asList(y));
    }
}
```

الخرج:

```
x = [10, 10, 10, 10, 10, 10, 10, 10, 10, 10]
y = [5, 5, 5, 5, 5]
x = [10, 10, 10, 10, 10, 5, 5, 5, 5, 5]
x = [10, 10, 10, 10, 10, 2, 2, 2, 2, 2]
y = [2, 2, 2, 2, 2]
```

لندرس المثال السابق:

قمنا بإنشاء مصفوفتين من الصف MyInt وملأنا كل منهما وقمنا بطباعتهما..

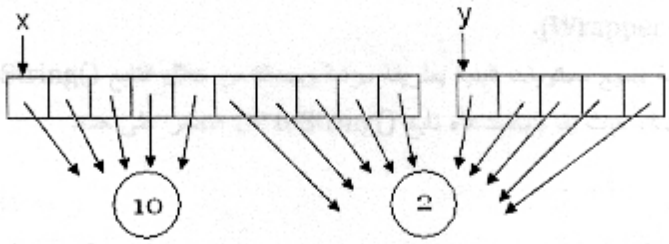
إن التابع asList() يرد سلسلة خطية[1] تحوي جميع عناصر مصفوفتنا، وعندها يمكن الاستفادة من التابع toString() لهذه السلسلة لطباعة محتوياتها.

ومن ثم قمنا بنسخ المصفوفة y إلى النصف الثاني للمصفوفة x، وطبعنا المصفوفة x بالقيم الجديدة.

بعدها غيرنا قيمة العنصر الثالث في المصفوفة y وطبعنا المصفوفة.. ولكن ما الذي حدث؟؟

في الحقيقة إن المثال السابق يوضح أن ملئ المصفوفة ونسخها إنما يكون للمؤشرات فقط!!

لذا فإن العمليات السابقة تكافئ الشكل التالي تماماً:



أحب أن أنوه في نهاية حديثي عن المصفوفات إلى أن مؤلف كتاب Thinking in Java السيد (Bruce Eckel) أدرج مع الكتاب صفاً هاماً يحتوي مجموعة من التوابع الجيدة والتي تساعد في التعامل مع المصفوفات وقد شرح توابع هذا الصف في المرجع، واسم الصف هو:

```
com.bruceeckel.util.Arrays2
```

---

[1] سنتعرف عليها بعد قليل

Containers:

قدمت لنا Java –تقريباً– كل ما نحتاج إليه من بنى معطيات مما قد مر معنا مسبقاً، ولكنني لن أقوم بشرح كل بنية على حدة، وإنما سأقدم رؤية عامة عنها وعن علاقاتها مع بعضها، وسأترك لكم مهمة تجربة هذه البنى..

قبل كل شيء يجب أن ننوه إلى أن جميع الصفوف التي سنتحدث عنها موجودة في المكتبة (java.util).

ما هي صفات هذه البنى؟

1. جميع هذه البنى قابلة للتوسع وليست محدودة مثل المصفوفات.
2. تمت الاستفادة من خاصية الـ polymorphism، حيث أن هذه البنى جميعاً ذات نوع واحد ولا يمكن تحديدها بأي نوع آخر، إذ أنها جميعاً تخزن references من نوع الصف Object[1]، ولهذا الكلام ميزات ومساوئ:
   - ✓ إن جميع هذه البنى يمكن أن تخزن أي نوع من الـ objects.
   - ☒ ولكن هذا يعني أننا عندما نريد استخدام هذه الأغراض يجب أن نعيدها إلى نوعها الأصلي عن طريق الـ Down-casting، وبالتالي يجب أن نتذكر جيداً نوع الأغراض التي وضعناها في بنية المعطيات.
   - ✓ كما أنه بالإمكان أن تحوي البنية نفسها أكثر من نوع من الـ objects.
   - ☒ ولكن هذا سيعطي كوارث عند استعادة هذه الأغراض، إذ أننا قد نقوم بعملية Down-casting خاطئة وبالتالي سنحصل على Exception، لذا يجب الانتباه جيداً إلى نوع الغرض قبل محاولة عمل casting.
3. نستنتج مما سبق أن هذه البنى غير قادرة على تخزين الـ primitives، إلا من خلال الـ (Wrapper classes).
4. يمكن طباعة جميع محتويات البنية بطريقة مرتبة وجميلة من خلال التابع toString() والذي يطبع كل عناصر البنية، حيث يقوم باستدعاء تابع toString() لكل عنصر على حدة.

مثال:

```
import java.util.*;

public class PrintingContainers
{
    static Collection fill(Collection c) {
        c.add("Apple");
        c.add("Apple");
        c.add("Orange");
        c.add(new Integer(10));
        return c;
    }
```


[1] الصف الأب لجميع صفوف Java

```
    public static void main(String[] args) {
        System.out.println(fill(new ArrayList()));
        System.out.println(fill(new HashSet()));
    }
}
```

الخرج:

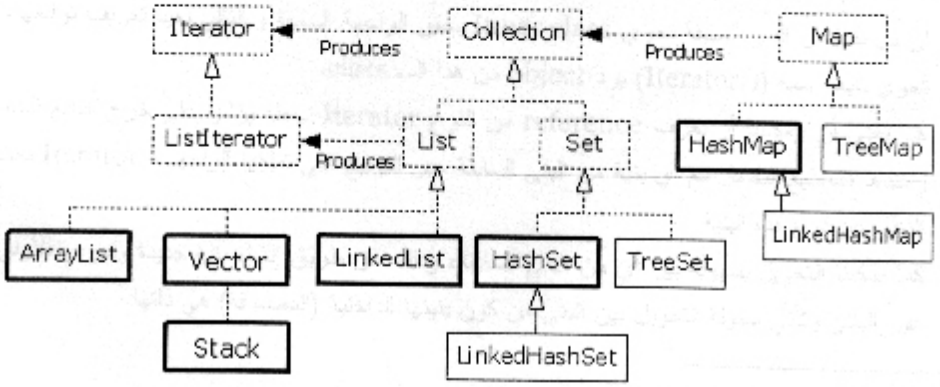
```
[Apple, Apple, Orange, 10]
[Apple, Orange, 10]
```

أنواع الـ Containers:

لهذه البنى نوعان:

1. *Collection*: وهي مجموعة من البنى التي تشترك مع بعضها بأنها عبارة عن سلسلة وحيدة تخـــزن فيها العناصر بشكل مستقل، وجميع هذه البنى تحقق الـــ interface (*Collection*)، وهي بـــدورها تقسم إلى قسمين:
   a. *List*: مجموعة من البنى كل منها مكون من سلسلة يمكن التخزين فيها كما نشاء وجميعها تحقق الواجهة List.
   b. *Set*: مجموعة من البنى كل منها مكون من سلسلة لا يمكن تكرار العنصر فيها أكثر من مرة[1] وجميعها تحقق الواجهة Set.
2. *Map*: وهي مجموعة من البنى التي تشترك مع بعضها بأن العناصر تخزن فيها بشكل مزدوج يعرف بـــ (key-value)، وهي تشبه Database مصغرة، وجميع هذه البنـــى تحقـــق الـــ interface (*Map*) وبما أنها أقل أهمية من الـــ Collections وبما أن الدكتور لم يتطرق لها مطلقاً، سأركز في المحاضرة على الـــ Collections فقط وسأترك لكم مهمة التعرف على الـــ Map.

سنضع مخططاً يوضح أهم البنى من كل نوع:



[1] وهذا ما يفسر خرج المثال السابق إذ أن HashSet نوع من أنواع الـــ Set

المربع المنقط: interface

المربع العادي: class

المربع الغامق: أهم البنى وأكثرها استخداماً.

Collections:

إن القوة التي قدمتها Java في هذه الصفوف تكمن في إمكانية التعامل معها جميعاً بنفس الطريقة مما يعطي البرنامج ثباتاً في التعامل مع أي بنية معطيات تقدم له، كما يمكن التحويل بينها بسهولة كما سنرى.

إن سبب هذه المرونة هي البنية الداخلية المشتركة لجميع البنى التي تحقق الـ interface (Collection) وهي مصفوفة. ولكن هذه البنى تختلف فيما بينها في التوابع التي تقدمها للتعامل مع هذه المصفوفة.

سنتعرف على المميزات المشتركة لهذه البنى:

- الصف *Collections*: يجب الانتباه إلى الفرق بين (interface *Collection*) و (class *Collections*)[1].

هذا الصف شبيه بالصف Arrays الذي تعرفنا عليه قبل قليل حيث يقدم مجموعة من التوابع الـ static الهامة جداً والتي تتعامل مع جميع البنى السابقة أنصح بالاطلاع عليها.

الواجهة[2] *Iterator*: قد أضطر في برنامجي للتعامل مع بنى معطيات من أنواع مختلفة (stack, list, queue) والقيام بمسح جميع عناصرها ولكن بغض النظر عن نوع هذه البنية، عندها أنا بحاجة لطريقة تعامل خاصة بكل بنية على حدة وهذا شيء متعب، لذا قدمت لنا Java فكرة جديدة تقوم على إيجاد طريقة تعامل مشتركة تسمح بمسح أي بنية مما سبق وكان هذا عن طريق الواجهة *Iterator* والتي تحوي التوابع التالية:

- next(): يرد هذا التابع العنصر التالي من السلسلة.
- hasNext(): يرد true إذا لم تنته السلسلة.

إن كل بنية من البنى السابقة تحوي Inner class يحقق الواجهة السابقة وبالتالي يعيد تعريف توابعها، كما تحوي تابعاً اسمه (Iterator()) يرد object من هذا الـ class.

هذا يعني أن بإمكاننا أن نعرف reference من النوع Iterator نجعله يؤشر على خرج التابع السابق وعندها نستطيع التعامل مع أي بنية من البنى السابقة عبر التوابع التي تملكها الواجهة Iterator بغض النظر عن نوع هذه البنية.

- كما يمكننا التحويل بسهولة بين أي من البنى السابقة، وذلك عن طريق إنشاء بنية جديدة وتمرير الأولى لها عبر الباني وتأتي سهولة التحويل بين البنى من كون بنيتها الداخلية (المصفوفة) هي ذاتها.

---

[1] انتبه إلى حرف الـ s في نهاية اسم الـ class

[2] أي الـ interface

إن كل بنية تتعامل مع المصفوفة التي بداخلها[1] بطريقة مختلفة عن الأخرى، مما يعطينا فارقاً في سـرعة الأداء بين هذه البنى وقد وضعت مقارنات في المرجع بين هذه البنى لذا يمكنكم العودة للمرجع للاطلاع على التوابـع المشتركة بين هذه البنى، ولتتعرفوا على المكان الأفضل لاستخدام كل منها..

مثال:

```
import java.util.*;

public class PrintingContainers
{
    static void print(Iterator it) {
        while (it.hasNext())
            System.out.print(it.next() + " ");
        System.out.println();
    }

    public static void main(String[] args) {
        LinkedList list = new LinkedList();
        for (int i = 0; i < 10; i++) {
            list.addFirst(new Integer(i));
            list.addLast(new Integer(i));
        }
        HashSet set = new HashSet(list);
        set.add(new Integer(10));

        print(list.iterator());
        print(set.iterator());
    }
}
```


الخرج:

```
9 8 7 6 5 4 3 2 1 0 0 1 2 3 4 5 6 7 8 9
4 8 3 7 2 9 6 1 10 5 0
```

- استطعنا من خلال الـ Iterator التعامل مع البنيتين وكأنهما بنية واحدة على الرغم من الفارق الكبير بينهما.
- نلاحظ كيف حولنا الـ LinkedList إلى HashSet عن طريق تمرير الأولى إلى باني الثانية، حيث قامت الثانية بمعالجة السلسلة التي استلمتها بما يتناسب مع بنيتها[2] و من ثم أصبحت جاهزة للعمل.

ملاحظة:

يوجد واجهة أخرى اسمها (Enumeration) تقوم بنفس عمل الواجهة Iterator ولكنها ليست موجودة في جميع الصفوف السابقة، ويمكن التعرف عليها من خلال الصف Vector الذي يحوي تابعاً يدعى (elements()) ويرد غرضاً من صف يحقق تلك الواجهة.

في نهاية حديثنا عن الـ Containers لابد أن أنوه إلى أنني التزمت بما أعطاه الدكتور علماً أن البحث أوسع من هذا ويحوي الكثير من التفاصيل الموجودة في المرجع والتي أنصح بقراءتها ولو بعد الامتحان..

1: أي وضع العناصر بداخلها وإعادة حجزها بهدف الترميز..

2: التي لا تسمح بتكرار العناصر