شرح مفصل لأكواد هياكل البيانات في Java

كل سطر مفسر مع الأمثلة والتطبيقات

المصفوفات | القوائم المرتبطة | الأكوام | الطوابير
الأشجار | الفرز

1. المصفوفات (Arrays)

المفهوم الأساسي:

المصفوفة هي هيكل بيانات يخزن مجموعة من العناصر من نفس النوع في مواقع ذاكرة متجاورة. تسمح بالوصول المباشر إلى أي عنصر باستخدام الفهرس.

الكود الكامل لبرنامج المصفوفات:

مثال عملي للمصفوفات — **ArrayApp.java**

```java
// array.java - برنامج يوضح المصفوفات في Java
// لتشغيل البرنامج: C>java ArrayApp

class ArrayApp {
    public static void main(String[] args) {
        // تعريف مرجع للمصفوفة
        long[] arr;

        // إنشاء مصفوفة من 100 عنصر
        arr = new long[100];

        // متغيرات المساعدة
```

```
13      int nElems = 0;    // عدد العناصر الفعلية
14      int j;             // عداد للحلقات
15      long searchKey;    // المفتاح للبحث
16
17      // ------ إدراج 10 عناصر يدوياً ------
18      arr[0] = 77;
19      arr[1] = 99;
20      arr[2] = 44;
21      arr[3] = 55;
22      arr[4] = 22;
23      arr[5] = 88;
24      arr[6] = 11;
25      arr[7] = 00;
26      arr[8] = 66;
27      arr[9] = 33;
28
29      // تحديث عدد العناصر الفعلية
30      nElems = 10;
31
32      // ------ عرض جميع العناصر ------
33      for (j=0; j<nElems; j++)
34          System.out.print(arr[j] + " ");
35      System.out.println("");
36
37      // ------ البحث عن عنصر ------
38      searchKey = 66;    // البحث عن القيمة 66
39
40      for (j=0; j<nElems; j++)
41          if (arr[j] == searchKey)
42              break;
43
44      if (j == nElems)
45          System.out.println("Can't find " + searchKey);
46      else
47          System.out.println("Found " + searchKey);
48
      // ------ حذف عنصر ------
      searchKey = 55;    // حذف القيمة 55
```

```
            for (j=0; j<nElems; j++)
                if (arr[j] == searchKey)
                    break;

            // نقل العناصر لإغلاق الفجوة
            for (int k=j; k<nElems-1; k++)
                arr[k] = arr[k+1];

            nElems--;    // تقليل عدد العناصر الفعلية

            // ------ عرض العناصر بعد الحذف ------
            for (j=0; j<nElems; j++)
                System.out.print(arr[j] + " ");
            System.out.println("");
        }
    }
```

شرح سطراً بسطر:

1. **الأسطر 1-2:** تعليقات توضيحية عن البرنامج وكيفية تشغيله
2. **الأسطر 4-5:** تعريف فئة البرنامج الرئيسية والتابع main
3. **الأسطر 7-8:** تعريف مرجع للمصفوفة ثم إنشائها فعلياً بحجم 100 عنصر
4. **الأسطر 10-12:** تعريف المتغيرات المساعدة:
   - nElems: يخزن عدد العناصر الفعلية في المصفوفة
   - j: عداد للحلقات
   - searchKey: مفتاح البحث
5. **الأسطر 14-23:** إدراج 10 قيم يدوياً في المصفوفة باستخدام الفهرس
6. **السطر 25:** تحديث nElems ليعكس عدد العناصر الحقيقي (10)
7. **الأسطر 28-31:** عرض جميع العناصر باستخدام حلقة for

8 **الأسطر 34-42:** عملية البحث عن القيمة 66:

- الحلقة تمر على جميع العناصر
- إذا تم العثور على العنصر، يتم الخروج من الحلقة
- يتم التحقق إذا وصلنا لنهاية المصفوفة (لم نجد العنصر)

9 **الأسطر 45-53:** عملية حذف القيمة 55:

- البحث عن موقع العنصر المطلوب حذفه
- نقل جميع العناصر التي تليه خطوة للخلف
- تقليل عدد العناصر الفعلية بمقدار 1

10 **الأسطر 56-59:** عرض المصفوفة بعد عملية الحذف

النتائج المتوقعة:

عند تشغيل البرنامج سيكون المخرجات:

```
77 99 44 55 22 88 11 0 66 33    // قبل الحذف
Found 66                        // نتيجة البحث
77 99 44 22 88 11 0 66 33       // بعد حذف 55
```

2. القوائم المرتبطة (Linked Lists)

المفهوم الأساسي:

القائمة المرتبطة هي هيكل بيانات يتكون من عقد (Nodes) مرتبطة معاً بواسطة مؤشرات. كل عقدة تحتوي على بيانات ومؤشر للعقدة التالية.

هيكل العقدة (Node):

تمثل عنصراً واحداً في القائمة المرتبطة

فئة Link (العقدة)

```java
class Link {
    // بيانات العقدة
    public int iData;           // المفتاح
    public double dData;        // بيانات أخرى
    public Link next;           // مؤشر للعقدة التالية

    // ------ المنشئ (Constructor) ------
    public Link(int id, double dd) {
        iData = id;         // تهيئة المفتاح
        dData = dd;         // تهيئة البيانات

        // next يتم تعيينه تلقائياً إلى null
        // (لا توجد عقدة تالية عند الإنشاء)
    }

    // ------ طريقة لعرض العقدة ------
    public void displayLink() {
        System.out.print("{" + iData + "," + dData + "}");
    }
}
```

فئة القائمة المرتبطة (LinkList):

تدير القائمة المرتبطة كاملة

فئة LinkList (القائمة)

```java
class LinkList {
    private Link first;    // مؤشر لأول عنصر في القائمة

    // ------ المنشئ ------
    public LinkList() {
        first = null;    // القائمة فارغة عند الإنشاء
    }

    // ------ التحقق إذا كانت القائمة فارغة ------
    public boolean isEmpty() {
```

```
12          return (first == null);
13      }
14
15      // ------ الإدراج في بداية القائمة ------
16      public void insertFirst(int id, double dd) {
17          // 1. إنشاء عقدة جديدة
18          Link newLink = new Link(id, dd);
19
20          // 2. جعل العقدة الجديدة تشير إلى العقدة الأولى الحالية
21          newLink.next = first;
22
23          // 3. جعل first يشير إلى العقدة الجديدة
24          first = newLink;
25      }
26
27      // ------ حذف أول عنصر ------
28      public Link deleteFirst() {
29          // 1. حفظ المرجع للعقدة الأولى
30          Link temp = first;
31
32          // 2. جعل first يشير إلى العقدة الثانية
33          first = first.next;
34
35          // 3. إرجاع العقدة المحذوفة
36          return temp;
37      }
38
39      // ------ عرض القائمة كاملة ------
40      public void displayList() {
41          System.out.print("List (first-->last): ");
42
43          // بدء من أول عقدة
44          Link current = first;
45
46          // التكرار حتى نهاية القائمة
47          while(current != null) {
48              // عرض العقدة الحالية
49              current.displayLink();
50
51
52
```

```
            // الانتقال للعقدة التالية
            current = current.next;
        }

        System.out.println("");
    }
}
```

شرح طريقة insertFirst خطوة بخطوة:

1 إنشاء عقدة جديدة:

```
Link newLink = new Link(id, dd);
```

تنشئ عقدة جديدة تحتوي على البيانات الممررة.

2 ربط العقدة الجديدة بأول عقدة حالية:

```
newLink.next = first;
```

تجعل مؤشر next في العقدة الجديدة يشير إلى العقدة التي كان first يشير إليها سابقاً.

3 تحديث مؤشر first:

```
first = newLink;
```

يجعل first يشير إلى العقدة الجديدة، لتصبح هي أول عقدة في القائمة.

مثال توضيحي لعملية الإدراج:

لو كان لدينا قائمة تحتوي على العقدة A فقط، ونريد إدراج العقدة B في البداية:

قبل الإدراج: first → A
الخطوة 1: إنشاء العقدة B
الخطوة 2: B.next = A (B تشير إلى A)
الخطوة 3: first = B (first يشير إلى B)

بعد الإدراج: first → B → A

3. الأكوام (Stacks)

المفهوم الأساسي:

الكومة (Stack) هي هيكل بيانات يعمل بمبدأ LIFO (Last In, First Out) - آخر من يدخل هو أول من يخرج. تشبه كومة من الأطباق.

فئة الكومة (StackX):

تنفذ الكومة باستخدام مصفوفة — **فئة StackX**

```
class StackX {
    private int maxSize;        // الحجم الأقصى للكومة
    private long[] stackArray;  // المصفوفة التي تخزن العناصر
    private int top;            // مؤشر لأعلى عنصر

    // ------ المنشئ ------
    public StackX(int s) {
        maxSize = s;                    // تعيين الحجم
        stackArray = new long[maxSize]; // إنشاء المصفوفة
        top = -1;                       // لا توجد عناصر بعد
    }

    // ------ إضافة عنصر إلى أعلى الكومة ------
    public void push(long j) {
        // زيادة top ثم إدراج العنصر في الموقع الجديد
        stackArray[++top] = j;
    }

```

```
20      // ------ إزالة عنصر من أعلى الكومة ------
21      public long pop() {
22          // إرجاع العنصر في top ثم إنقاص top
23          return stackArray[top--];
24      }
25
26
27      // ------ النظر إلى أعلى عنصر بدون إزالته ------
28      public long peek() {
29          return stackArray[top];
30      }
31
32      // ------ التحقق إذا كانت الكومة فارغة ------
33      public boolean isEmpty() {
34          return (top == -1);
35      }
36
37
38      // ------ التحقق إذا كانت الكومة ممتلئة ------
39      public boolean isFull() {
40          return (top == maxSize - 1);
41      }
42  }
43
44
45
46
47
48
49
```

كيف يعمل متغير top:

1 الحالة الأولية (كومة فارغة):

top = -1 (يشير إلى عدم وجود عناصر)

2 بعد push(10):

top = 0 (العنصر 10 في الفهرس 0)

3 بعد push(20):

top = 1 (العنصر 20 في الفهرس 1)

4 بعد pop():

ترجع 20، ثم top = 0 (العنصر 10 في الأعلى)

5 بعد pop() أخرى:

ترجع 10، ثم top = -1 (الكومة فارغة)

تطبيق عملي: عكس كلمة باستخدام الكومة

فئة Reverser — تستخدم الكومة لعكس النصوص

```
class Reverser {
    private String input;    // النص المدخل
    private String output;   // النص المعكوس

    public Reverser(String in) {
        input = in;
    }

    public String doRev() {
        // تحديد حجم الكومة بناءً على طول النص
        int stackSize = input.length();

        // إنشاء كومة بالحجم المناسب
        StackX theStack = new StackX(stackSize);

        // إدخال جميع الحروف في الكومة
        for(int j=0; j<input.length(); j++) {
            char ch = input.charAt(j);
            theStack.push(ch);
        }

        output = "";

        // إخراج الحروف من الكومة (بترتيب عكسي)
```

```
        while ( !theStack.isEmpty() ) {
            char ch = theStack.pop();
            output = output + ch;
        }

        return output;
    }
}
```

مثال: عكس كلمة "Java"

الخطوة 1: إدخال الحروف في الكومة: J → a → v → a
الخطوة 2: إخراج الحروف من الكومة: a ← v ← a ← J
النتيجة: "avaJ" (Java معكوسة)

4. الطوابير (Queues)

المفهوم الأساسي:

الطابور (Queue) هو هيكل بيانات يعمل بمبدأ FIFO (First In, First Out) - أول من يدخل هو أول من يخرج. يشبه طابور الانتظار.

فئة الطابور (Queue):

فئة Queue — تنفذ طابوراً دائرياً باستخدام مصفوفة

```
class Queue {
    private int maxSize;        // الحجم الأقصى
    private int[] queArray;     // المصفوفة
    private int front;          // مؤشر المقدمة
    private int rear;           // مؤشر المؤخرة
```

```
6       private int nItems;          // عدد العناصر
7
8       // ------ المنشئ ------
9       public Queue(int s) {
10          maxSize = s;
11          queArray = new int[maxSize];
12          front = 0;          // المقدمة تبدأ من 0
13          rear = -1;          // المؤخرة تبدأ من -1 (فارغة)
14          nItems = 0;         // لا توجد عناصر
15      }
16
17      // ------ إدراج عنصر في المؤخرة ------
18      public void insert(int j) {
19          // إذا وصل rear (دائري) إلى نهاية المصفوفة، يعود للبداية
20          if(rear == maxSize - 1)
21              rear = -1;
22
23          // زيادة rear ثم إدراج العنصر
24          queArray[++rear] = j;
25
26          nItems++;    // زيادة عدد العناصر
27      }
28
29      // ------ إزالة عنصر من المقدمة ------
30      public int remove() {
31          // حفظ العنصر المطلوب إزالته
32          int temp = queArray[front++];
33
34          // إذا وصل front إلى نهاية المصفوفة، يعود للبداية
35          if(front == maxSize)
36              front = 0;
37
38          nItems--;    // تقليل عدد العناصر
39
40          return temp;    // إرجاع العنصر المحذوف
41      }
42
43      // ------ التحقق إذا كان الطابور فارغاً ------
44      public boolean isEmpty() {
45
46
47
48
```

```
49            return (nItems == 0);
50        }
51
        // ------ التحقق إذا كان الطابور ممتلئاً ------
        public boolean isFull() {
            return (nItems == maxSize);
        }

        // ------ الحصول على عدد العناصر ------
        public int size() {
            return nItems;
        }
    }
```

كيف يعمل الطابور الدائري:

1. **التهيئة:**
   front = 0, rear = -1, nItems = 0

2. **بعد insert(10):**
   rear = 0, queArray[0] = 10, nItems = 1

3. **بعد insert(20):**
   rear = 1, queArray[1] = 20, nItems = 2

4. **بعد remove():**
   ترجع 10, front = 1, nItems = 1

5. **مثال الالتفاف:**
   إذا كان maxSize = 3 وكان rear عند 2، عند insert() جديد: rear = -1 ثم rear = 0 (يعود للبداية)

مقارنة بين الطابور العادي والدائري:

النوع	المشكلة	الحل
الطابور العادي	عندما نصل لنهاية المصفوفة، لا يمكن إضافة عناصر جديدة حتى لو كانت هناك أماكن فارغة في البداية	إزاحة جميع العناصر للأمام (O(n))
الطابور الدائري	نفس المشكلة	عند الوصول للنهاية، نعود للبداية (O(1))

5. الأشجار الثنائية (Binary Trees)

المفهوم الأساسي:

الشجرة الثنائية هي هيكل بيانات هرمي حيث كل عقدة لها على الأكثر طفلين: أيسر (أصغر) وأيمن (أكبر). تستخدم بكثرة في عمليات البحث والفرز.

فئة العقدة (Node):

فئة Node — تمثل عقدة في الشجرة الثنائية

```
class Node {
    public int iData;            // مفتاح العقدة
    public double dData;         // بيانات العقدة
    public Node leftChild;       // الطفل الأيسر
    public Node rightChild;      // الطفل الأيمن

    // ------ طريقة لعرض العقدة ------
    public void displayNode() {
        System.out.print("{" + iData + "," + dData + "}");
    }
}
```

طريقة الإدراج في الشجرة الثنائية للبحث (BST):

تدرج عقدة جديدة في المكان الصحيح

طريقة insert() في فئة Tree

```
public void insert(int id, double dd) {
    // 1. إنشاء عقدة جديدة
    Node newNode = new Node();
    newNode.iData = id;
    newNode.dData = dd;

    // 2. إذا كانت الشجرة فارغة، تصبح العقدة الجذر
    if(root == null)
        root = newNode;

    // 3. إذا كانت الشجرة غير فارغة، ابحث عن المكان المناسب
    else {
        Node current = root;     // ابدأ من الجذر
        Node parent;             // لتذكر الأب

        while(true) {
            parent = current;

            // إذا كان المفتاح الجديد أصغر من المفتاح الحالي
            if(id < current.iData) {
                current = current.leftChild;

                // إذا وصلنا لنهاية الفرع الأيسر
                if(current == null) {
                    parent.leftChild = newNode;
                    return;
                }
            }
```

```
33                  // إذا كان المفتاح الجديد أكبر من المفتاح الحالي
34                  else {
35                      current = current.rightChild;
36
37                      // إذا وصلنا لنهاية الفرع الأيمن
38                      if (current == null) {
39                          parent.rightChild = newNode;
                            return;
                        }
                    }
                }
            }
        }
```

مثال: إدراج العقد [50, 30, 70, 20, 40]

1 **إدراج 50:** تصبح الجذر

```
    50
   \  /
null  null
```

2 **إدراج 30:** 30 < 50، تذهب لليسار

```
    50
   \  /
null  30
     \ /
  null null
```

3 **إدراج 70:** 70 > 50، تذهب لليمين

```
    50
   \  /
 70    30
 \ /  \ /
```

```
null null null null
```

4 **إدراج 20:** 20 < 50 ← اذهب لليسار، 20 > 30 ← اذهب لليسار

```
        50
       \   /
    70     30
    \ /  \ /
null null null 20
            \ /
        null null
```

5 **إدراج 40:** 40 < 50 ← اذهب لليسار، 40 > 30 ← اذهب لليمين

```
        50
       \   /
    70     30
    \ /  \ /
null null 40  20
            \ /\ /
null null null null
```

طرق اجتياز الأشجار الثنائية:

الاجتياز التصاعدي (Inorder) — يسار ← جذر ← يمين

```java
private void inOrder(Node localRoot) {
    if(localRoot != null) {
        // 1. اجتياز الفرع الأيسر
        inOrder(localRoot.leftChild);

        // 2. زيارة الجذر (العرض)
        System.out.print(localRoot.iData + " ");

        // 3. اجتياز الفرع الأيمن
        inOrder(localRoot.rightChild);
```

```
    }
}
```

نتائج الاجتياز لمثالنا السابق:

- **Inorder (تصاعدي):** 20 30 40 50 70 (مُرتَّب تصاعدياً)
- **Preorder (مسبق):** 50 30 20 40 70 (الجذر أولاً)
- **Postorder (لاحق):** 20 40 30 70 50 (الجذر أخيراً)

6. خوارزميات الفرز (Sorting Algorithms)

المفهوم الأساسي:

خوارزميات الفرز تُرتِّب العناصر في تسلسل معين (تصاعدي أو تنازلي). تختلف الخوارزميات في الكفاءة والتعقيد الزمني.

فرز الفقاعة (Bubble Sort):

ترتيب العناصر بالمقارنات المتتالية — **طريقة bubbleSort()**

```
public void bubbleSort() {
    int out, in;

    // الحلقة الخارجية: تبدأ من النهاية وتتجه للبداية
    for(out = nElems - 1; out > 1; out--) {

        // الحلقة الداخلية: تبدأ من البداية وتتجه للنهاية
        for(in = 0; in < out; in++) {

            // إذا كان العنصر الحالي أكبر من العنصر التالي
            if(a[in] > a[in + 1]) {
```

```
                    // تبديلهما
                    swap(in, in + 1);
                }
            }
        }
    }
```

مثال: فرز [5, 3, 8, 1] باستخدام Bubble Sort

1 التكرار الأول (out = 3):

[5, 3, 8, 1] → 3 < 5؟ نعم ← تبديل ← [3, 5, 8, 1]

[3, 5, 8, 1] → 8 < 5؟ لا ← [3, 5, 8, 1]

[3, 5, 8, 1] → 1 < 8؟ نعم ← تبديل ← [3, 5, 1, 8]

2 التكرار الثاني (out = 2):

[3, 5, 1, 8] → 5 < 3؟ لا ← [3, 5, 1, 8]

[3, 5, 1, 8] → 1 < 5؟ نعم ← تبديل ← [3, 1, 5, 8]

3 التكرار الثالث (out = 1):

[3, 1, 5, 8] → 1 < 3؟ نعم ← تبديل ← [1, 3, 5, 8]

4 النتيجة النهائية:

[1, 3, 5, 8] (مرتب تصاعدياً)

فرز الاختيار (Selection Sort):

طريقة selectionSort() — اختيار أصغر عنصر ووضعه في مكانه الصحيح

```
public void selectionSort() {
    int out, in, min;

    // الحلقة الخارجية: تمر على جميع العناصر عدا الأخير
    for(out = 0; out < nElems - 1; out++) {

        min = out;    // افترض أن العنصر الحالي هو الأصغر
```

```
8
9           // الحلقة الداخلية: ابحث عن أصغر عنصر في الجزء غير المرتب
10          for (in = out + 1; in < nElems; in++) {
11
12              // الحالي min إذا وجدت عنصراً أصغر من
13              if (a[in] < a[min]) {
14
15                  // تحديث min ليشير للعنصر الأصغر
16                  min = in;
17              }
18          }

            // تبديل العنصر الأصغر مع العنصر الحالي
            swap(out, min);
        }
    }
```

مثال: فرز [5, 3, 8, 1] باستخدام Selection Sort

1 التكرار الأول (out = 0):

أبحث عن أصغر عنصر في [5, 3, 8, 1] ← العنصر 1 في الفهرس 3

أبدل العنصر 0 مع العنصر 3 ← [1, 3, 8, 5]

2 التكرار الثاني (out = 1):

أبحث عن أصغر عنصر في [3, 8, 5] (بدون العنصر 0) ← العنصر 3 في الفهرس 1

التبديل لا يغير شيئاً (نفس الموقع) ← [1, 3, 8, 5]

3 التكرار الثالث (out = 2):

أبحث عن أصغر عنصر في [8, 5] ← العنصر 5 في الفهرس 3

أبدل العنصر 2 مع العنصر 3 ← [1, 3, 5, 8]

4 النتيجة النهائية:

[1, 3, 5, 8] (مرتب تصاعدياً)

🎯 ملخص شامل وتوصيات

📊 مقارنة هياكل البيانات:

الهيكل	مبدأ العمل	أفضل استخدام	التعقيد الزمني
📦 **المصفوفة**	عناصر في ذاكرة متجاورة	عندما يكون الحجم معروفاً مسبقاً	الوصول: O(1) الإدراج/الحذف: O(n)
🔗 **القائمة المرتبطة**	عقد مرتبطة بمؤشرات	عند تكرار الإدراج/الحذف	الإدراج/الحذف: O(1) البحث: O(n)
⬆️ **الكومة (Stack)**	LIFO (آخر يدخل أول يخرج)	التراجع، استدعاء الدوال	جميع العمليات: O(1)
➡️ **الطابور (Queue)**	FIFO (أول يدخل أول يخرج)	جدولة المهام، طوابير الانتظار	جميع العمليات: O(1)
🌳 **الشجرة الثنائية**	عقد مرتبطة هرمياً	البحث، الفرز، التصنيف	البحث/الإدراج/الحذف: O(log n)

💡 نصائح للاختيار:

- ✓ استخدم **المصفوفات** عندما تعرف الحجم مسبقاً وتحتاج وصولاً سريعاً
- ✓ استخدم **القوائم المرتبطة** عندما يكون الحجم متغيراً وتحتاج إدراجاً/حذفاً متكرراً

- ✓ استخدم **الأكوام** للعمليات العكسية (مثل التراجع)
- ✓ استخدم **الطوابير** للمهام المتسلسلة (مجدولة)
- ✓ استخدم **الأشجار الثنائية** للبحث السريع والبيانات المرتبة

🎓 **تذكر:** اختر الهيكل المناسب بناءً على متطلبات تطبيقك وطبيعة البيانات التي تتعامل معها

🏆 **تهانينا! لقد أتممت دراسة أساسيات هياكل البيانات**

الآن يمكنك تطبيق هذه المعرفة في مشاريعك البرمجية بثقة