CSE 1030

Yves Lespérance

Lecture Notes

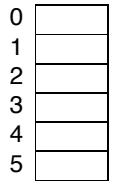
Week 10 — Arrays

Recommended Readings:
Van Breugel & Roumani Ch. 8 and Savitch Ch. 6

Arrays

In many applications, we need to work with large collections of similar pieces of data, e.g. the marks of a class in a test, a `ClassSection` object that contain the names and student number of students in a class, etc. We need a way of representing and organizing such data.

A commonly used data structure for this is *arrays*. Array types are built into Java. Think of an array as a sequence of boxes each containing one piece of data.



Each box is called an *element* of the array. The *size* of the array (number of boxes) is fixed. The boxes are numbered from 0 to *size* – 1; this number is called the *index* or *subscript* of the array element.

Arrays are said to be homogenous and ordered data structures — their elements are all of the same type (homogenous) and are kept in a specific order. This is in contrast to the data structuring mechanism provided by aggregation in objects, where the attributes or fields can be of different types and are accessed by name, not position.

When declaring an array variable, one gives the type of the elements and the name of the array, e.g.

```
double[] marks;
```

When the array is constructed, one supplies the size of the array, e.g.

```
marks = new double[40];
```

Of course, declaration and construction can be done in a single statement, e.g.

```
int[] hoursWorked = new int[31];
```

Once an array `a` has been created, one can refer to the element with index i by `a[`i`]`, e.g.

```
marks[0] = 75.6;
marks[1] = 80.1;
System.out.println(marks[1]);
marks[0] = marks[0] + 3.5;
int i = 2;
marks[i] = 83.5;
marks[i-1] = 74.3;
```

Every array has a `length` attribute that contains its size as given when it was created. E.g. you can retrieve the size of an array `a` by writing `a.length`.

When using arrays, one must be very careful to always use an index that is within the array bounds. Evaluating `a[`i`]` where i has a value that is out of range, i.e. where $i < 0$ or i >= `a.length`, is an error.

You can also create and initialize an array by enumerating the values to be stored in it, e.g.

```
int[] temperatures = {3, 6, 10,
             12, 4, 3, 5};
```

The size of the array is automatically determined.

Note that arrays are objects, so the array variable contains a reference to the array object.


```
public class Eg
{  public static final int MAXMARKS = 40;

   public static void main(String[] args)
   {  Scanner in = new Scanner(System.in);
      double[] marks = new double[MAXMARKS];
      double mark;
      int marksSize = 0;  // array's actual size
      while(true)
      {  System.out.print("Enter mark (negative to stop): ");
         mark = in.nextDouble();
         if(mark < 0) break;
         if(marksSize == marks.length)
         {  System.out.println("Array is full.");
            break;
         }
         marks[marksSize] = mark;
         marksSize++;
      }
```



```
      if (marksSize == 0)
        System.out.println("No data.");
      else            // calculate average
      { double sum = 0;
         for(int i = 0; i < marksSize; i++)
            sum += marks[i];
         System.out.println("The class average is " +
            sum / marksSize);
      }
   }
}
```

Ready-Made Methods for Array Manipulation

Several methods provided by class `Arrays` in `java.util`:

`Arrays.copyOf(a,length)` returns a new array which is a copy of the existing array `a` truncating or padding with default values so that the returned array has the specified `length`; can also copy a subrange.

`Arrays.sort(a)` sorts the elements of fully-populated array `a` into ascending order (using quicksort); can also sort a subrange.

`Arrays.binarySearch(a,key)` searches a sorted and fully-populated array `a` (using binary search) for `key` and returns its index if found otherwise returns negative integer.

`Arrays.fill(a,v)` fills array `a` with value `v`; can also fill a subrange.

`Arrays.equals(a,b)` compares arrays `a` and `b`.


```
public class Eg
{  public static void main(String[] args)
   {  int classSize = 3;
      String[] studNames = new String[classSize];
      studNames[0] = "Paul";
      studNames[1] = "John";
      studNames[2] = "Mary";
      Arrays.sort(studNames);
      int pos = Arrays.binarySearch(studNames,"Mary");
      System.out.println("Mary is now at index " + pos);
      // inserting "Lynn" using arrayCopy
      // easy but inefficient
      classSize++;
      String[] tmp = Arrays.copyOf(studNames,classSize);
      tmp[classSize] = "Lynn";
      Arrays.sort(tmp);
      studNames = tmp;
      // another search
      int pos = Arrays.binarySearch(studNames,"Mary");
      System.out.println("Mary is now at index " + pos);
   }
}
```


Passing Arrays as Parameters

Arrays are objects, even arrays whose elements are of a primitive type. So an array variable only contains a reference to the array object. When you use this array variable as an argument in a method call, the only thing that gets copied into the parameter is the array reference, not the array itself. Both the argument and parameter refer to the same array object. So the method is working on the original array. Any change to the parameter's elements also changes the argument's elements.

This is not a bad thing because copying an array, something that may be quite large, can be costly in terms of time and space.

```
public class Eg
{  public static void main(String[] args)
   {  int[] a = new int[3];
      a[1] = 0;
      int n = 0;
      System.out.println("in main a[1] = " + a[1]);
      System.out.println("in main n = " + n);
      myMeth(a,n);
      System.out.println("in main a[1] = " + a[1]);
      System.out.println("in main n = " + n);
   }

   public static void myMeth(int[] b, int m)
   {  m++;
      b[1]++;
      System.out.println("in myMeth b[1] = " + b[1]);
      System.out.println("in myMeth m = " + m);
   }
}
```

Multidimensional Arrays

Arrays may be indexed along several dimensions. E.g.

```
public class Eg
{  static final int ROWS = 2;
   static final int COLS = 3;

   public static double[][] readMatrix()
   {  Scanner in = new Scanner(System.in);
      double[][] m = new double[ROWS][COLS];
      for(int i = 0; i < ROWS; i++)
      {  for(int j = 0; j < COLS; j++)
         {  System.out.print("Enter element " + i +
                        "," + j + ": ");
            m[i][j] = in.nextDouble();
         }
      }
      return m;
   }
```



```
public static void printMatrix(double[][] a)
 {  for(int i = 0; i < a.length; i++)
    {  for(int j = 0; j < a[i].length; j++)
          System.out.print(a[i][j],"7.2");
       System.out.println("");
    }
 }


 public static double[][] addMatrices(
    double[][] a, double[][] b)
 {  assert(a.length == b.length &&
                a[0].length == b[0].length);
    double[][] sum =
       new double[a.length][a[0].length];
    for(int i = 0; i < a.length; i++)
       for(int j = 0; j < a[i].length; j++)
          sum[i][j] = a[i][j] + b[i][j];
    return sum;
 }
```



```
   public static void main(String[] args)
   {  System.out.println("Reading 1st matrix");
      double[][] a = readMatrix();
      System.out.println("Reading 2nd matrix");
      double[][] b = readMatrix();
      double[][] sum = addMatrices(a,b);
      System.out.println("Printing sum matrix");
      printMatrix(sum);
   }
}
```