UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escola Tècnica Superior d'Enginyeria Informàtica

etsinf

Unit 7. Arrays: definition and applications

Introduction to Computer Science and Computer Programming
Introducción a la Informática y la Programación (IIP)

Year 2017/2018

Departamento de Sistemas Informáticos y Computación

Contents

Contents

Introduction: Unidimensional arrays

Frequently it is necessary to store and reference variables that represent a set of values, in order to treat them in a uniform way

For example:

- Obtaining basic statistics on daily mean temperatures
- Manage a collection of homogeneous objects; e.g., a `Parking` class can have associated a set of `Vehicle` objects

Java provides the *array* structure to group data of homogeneous datatype (primitive or references)

Introduction: Unidimensional arrays

- ***Definition***: an ***array*** is a collection of ***homogeneous*** elements (same datatype) consecutively grouped in memory

- ***Features***
  - Each ***element*** in an array has an associated ***index***
  - An ***index*** is a non-negative integer number that identifies the element and allows accessing to it
  - The ***size*** (number of elements) of the array is established in its declaration
  - Array size is ***invariable*** during the execution
  - The access to the elements is ***direct***, by using the index

- Arrays are ***data structures*** (group data)

- Arrays are appropriate for random (i.e., indexed data) or sequential access

Introduction: Unidimensional arrays

Declaration and use

- ***Declaration and creation***:
  - Declaration: `type [] arrayVar;`   `type arrayVar[];` valid but not recommended
  - Initialisation: `arrayVar = new type[size];`
  - Altogether: `type [] arrayVar = new type[size];`

  - *type*: primitive or reference datatype
  - *arrayVar*: valid identifier
  - *size*: expression evaluated to a positive integer that gives the number of components

- ***Array access operator*** `[]`: to access the components

  `arrayVar [ index ]`

  Where *index* is an expression that evaluates to a valid value for *arrayVar*

- ***Literal values***: sequence of elements between braces (`{}`) separated by commas (`,`), `{`e_0`, `e_1`, ..., `e_{n-1}`}`

Introduction: Unidimensional arrays
Declaration and use

- All arrays have a constant attribute ***length*** (size of the array):

$$arrayVar.\mathit{length}$$

- Valid indexes for an array go from 0 till ***length***`-1`

- An array variable is a reference to the memory address of the first element of the array

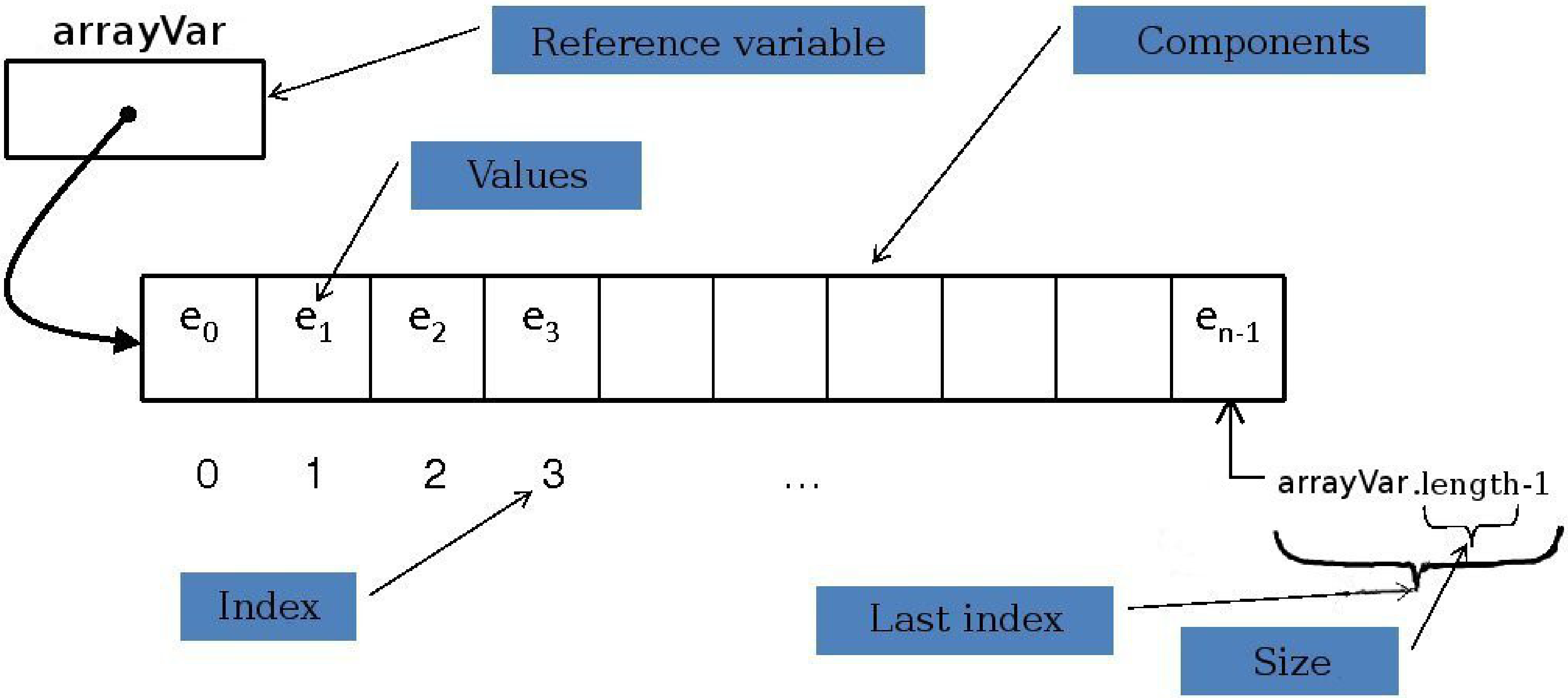
- The `null` reference can be used in arrays (just like with the other references)

- For a ***numerical array***, all its components get initialised to 0

- For a ***reference array*** (e.g., array of `String`), all its components get initialised to `null`

Introduction: Unidimensional arrays
Declaration and use

A possible graphical representation of an array:



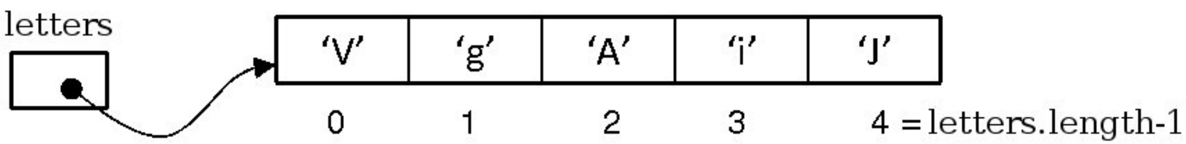
Where $\texttt{n = arrayVar.length}$ and all $\texttt{e}_i$ $(0 \leq i < \texttt{n})$ are of the same datatype

Introduction: Unidimensional arrays

Declaration and use

Some examples:

- Array of 5 characters:
  - Reference var: `char [] letters;`
  - Array object (init components): `letters = new char[5];`
  - Altogether: `char [] letters = new char[5];`



- Two array of integers:

```
int[] numbers1 = new int[5000], numbers2 = new int[50];
```

Introduction: Unidimensional arrays
Declaration and use

Some examples:

- Array of 20 `String`: `final int NUM = 10;`
  `String[] names = new String[NUM*2];`

- Array of real numbers where its size is given by keyboard input:
  `double[] costs = new double[kbd.nextInt()];`

- Creating and initialising an array with 4 integer numbers:
  `int[] v={-5,6,10,3};`

Introduction: Unidimensional arrays

Exceptions

- In an array of size `N`, valid indexes are in the interval `[0, N-1]`
- Accessing outside this interval will ***raise*** an ***exception*** (execution error)
- For example:

```
int[] n = new int[10];
int a = n[10];              // Raises an exception
a = n[-1];                  // Raises an exception
```

- The access to the elements of an array must be controlled by using the proper lower (0) and upper (`varArray.length-1`) bounds of the indexes

Introduction: Unidimensional arrays
Memory

- An array reference (`arrayVar`) is in the memory space where it was defined (attributes in heap, local variables in stack,. . .)
- ***Dynamic variables*** are those created in execution time by using `new`
- Thus, components of the array are always in the ***heap*** (dynamic var)
- Components can be accessed only when there is a reference towards them
- An array can be suggested to be destroyed by derreferencing it (`v = null`)
  - Reminder: the ***garbage collector*** automatically frees the memory in the heap without references (i.e., data that cannot be accessed)
  - The garbage collector can be suggested to be executed by using `System.gc()`

Introduction: Unidimensional arrays

Assignment

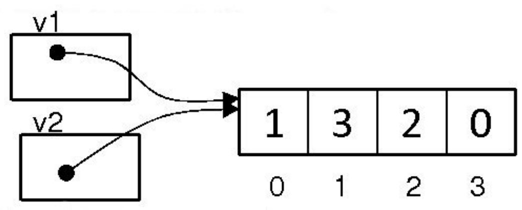
- Array components can be considered as variables of the array datatype

  `arrayVar[index] = expression;`

  `expression` must be of a datatype compatible with that of `arrayVar`

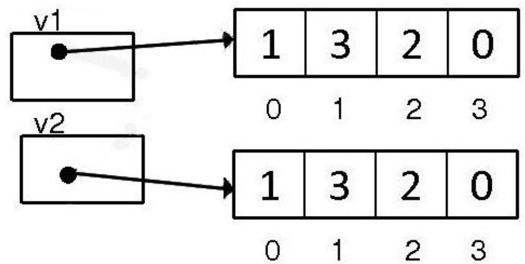
- Assignment between arrays only affects the references:

```
int[] v1 = {1,3,2,0};
int[] v2;
v2 = v1;
boolean eq = (v1==v2); // eq is true
```



- To copy the contents, create a new array and assign values individually

```
v2 = new int[4];
v2[0]=v1[0]; v2[1]=v1[1];
v2[2]=v1[2]; v2[3]=v1[3];
eq = (v1==v2); // eq is false
```



Introduction: Unidimensional arrays
Methods

- ***Formal parameters***: datatype (with `[]`) and name must be indicated

```
public static int method1(int[] v1, int[] v2) { ... }
public static void main(String[] args) { ... }
```

- ***Actual parameters***: in the call, only the name of the array is used:

```
int [] a1 = new int[N], a2 = new int[N+5];
int i = method1(a1,a2);
```

- The reference actual parameter is copied into the formal parameter
- Methods can return arrays (reference to components); for example:

```
public char[] method2(int[] v1) {
    char[] newArr = new char[v1.length+10];
    ...
    return newArr;
}
```

Introduction: Unidimensional arrays

Example: passing arrays as parameters

```
public class PassOfParameters {

  public static void main(String[] args) {
    double[] theArray = {5.0, 6.4, 3.2, 0.0, 1.2};
    method1(theArray);  // array not modified
    method2(theArray);  // first component is now 0.1
  }

  public static void method1(double[] copy) {
    copy = new double[5]; // This array dissapears when
                          // the method terminates
  }

  public static void method2(double[] copy) {
    copy[0] = 0.1;
  }
}
```

Introduction: Unidimensional arrays

Arrays of objects

Arrays of objects can be declared

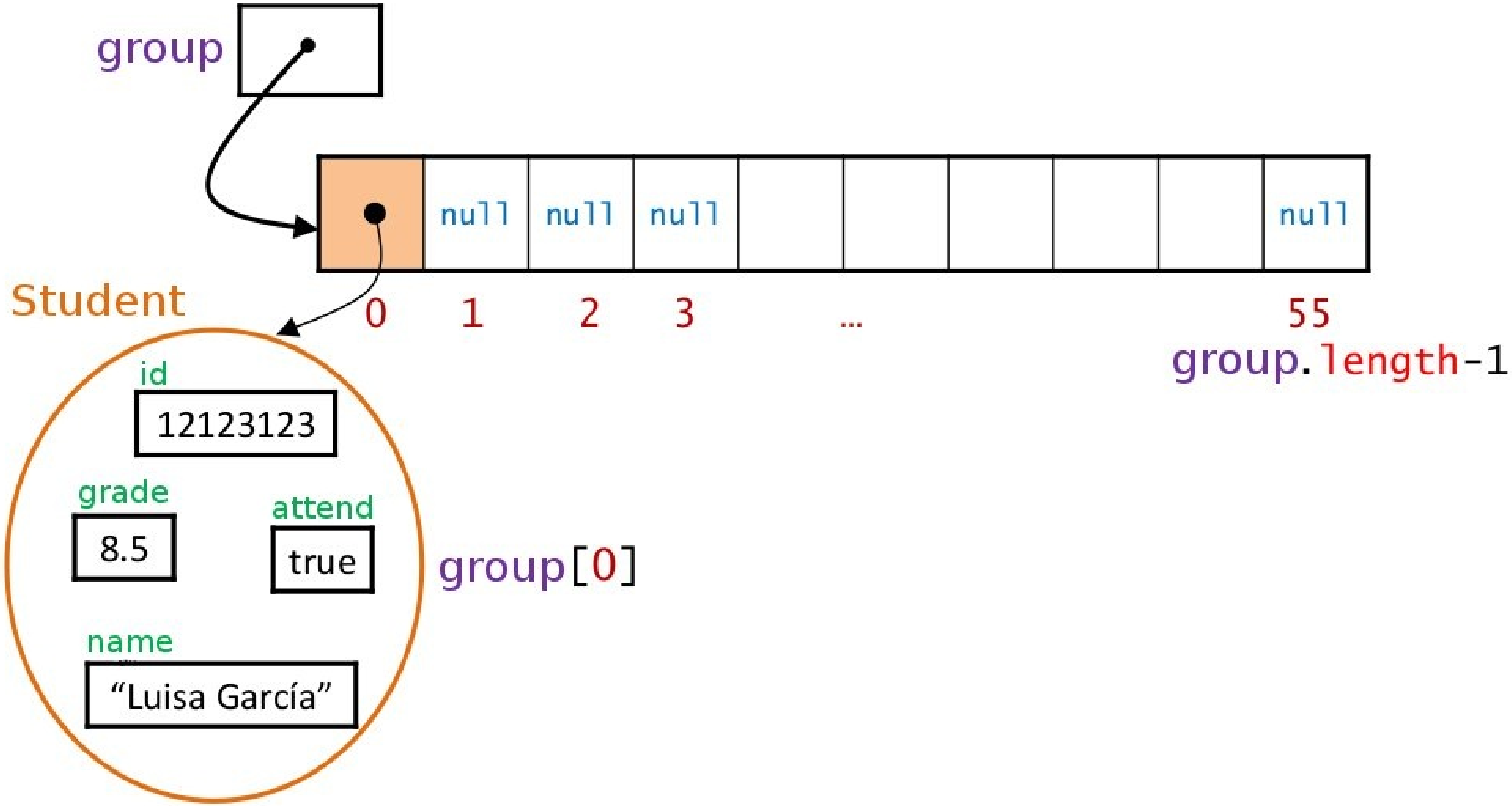
In this case, the components of the array are initially `null`

```
class Student {
  private long id;
  private double grade;
  private String name;
  private boolean attend;
  ...
}
```

```
public static void main(String[] args) {
  Student group[] = new Student[56];
  group[0] = new Student();
  group[0].setId(12123123);
  group[0].setGrade(8.5);
  group[0].setName("Luisa Garcia");
  group[0].setAttend(true);
}
```

Introduction: Unidimensional arrays
Arrays of objects

Graphically:



Introduction: Unidimensional arrays
Arrays of objects

Example: assuming that `Point2D` class implements the `dist` method

```
public class TestObjectArray {
  public static final int NUM_POINTS = 20;

  public static void main(String[] args) {
    Point2D[] path = new Point2D[NUM_POINTS];
    for(int i=0; i<path.length; i++) path[i] = new Point2D();
    // 20 Point2D objects of the path were created
    ...
    System.out.println("Dist between 0 and 1: " + path[0].dist(path[1]));
    Point2D[] copyP = copyPoint2D(path);
  }

  public static Point2D[] copyPoint2D(Point2D[] o) {
    Point2D[] c = new Point2D[o.length];
    for (int i=0; i<o.length; i++) c[i] = new Point2D(o[i].getX(),o[i].getY());
    return c;
  }

}
```

Contents

Operations on unidimensional arrays

In linear structure, two different accesses exist: ***direct*** and ***sequential***

- ***Direct***: elements accessed by their position (no specific access pattern)
  - Common examples: problems that use the linear structure as a set of counter or as position references
  - Complex examples: problems such as binary search and ordering algorithms, that use direct access and order properties

- ***Sequential***: elements are accessed positionally, one after the other
  - Example problems: listing, sequential search

Arrays are ***direct access structures***: the two types may be used

Operations on unidimensional arrays
Direct access: array of counters

Suppose we want to count the number of times that a user writes numbers from 0 to 9.

Innapropriate solution:

- Define ten variables (`count0`, `count1`, . . . , `count9`)
- Implement a set of `if` or `switch` structures that allow to increment the corresponding counter

Smarter solution: by using an array of counters

Operations on unidimensional arrays

Direct access: array of counters

Possible implementation:

```
import java.util.*;
public class CounterExample {
    public static void main(String[] args) {
        Scanner kbd = new Scanner(System.in).useLocale(Locale.US);
        final int MAX_VAL = 9;
        int[] counters = new int[MAX_VAL+1];
        boolean end = false;
        int val;
        do{
            val=kbd.nextInt();
            if (val>=0 && val<=counters.length-1) counters[val]++;
            else end=true;
        } while(!end);
    }
}
```

Operations on unidimensional arrays
Sequential operations: listing and searching

- ***Listing***: process that visits ***all*** the elements of the array to solve a problem
- ***Searching***: looks for the first element that accomplishes a condition
- Array listing is used to solve problems that need to process all data items to determine their solution
  - Obtain ***maximum*** or ***minimum*** element of a set of numbers
  - Obtain the ***sum*** or ***product*** of all the elements of a set of numbers
  - Obtain the ***mean***
  - ...
- Listing is performed by using indexes to access the positions
- Control on visited positions and the total to be visited to solve the problem

Operations on unidimensional arrays
Ascendent listing

Iterative ascendent listing with a `while` loop:

```
int i = initP;
while (i <= endP) {
  operateWith(a[i]);  // Operations with i-th element
  goUp(i);
}
```

Iterative ascendent listing with a `for` loop:

```
for (int i = initP; i <= endP; goUp(i)) {
  operateWith(a[i]);  // Operations with i-th element
}
```

Method `goUp` represents the increment of the index (usually, `i++`)

Operations on unidimensional arrays
Ascendent listing

Warning!: in arrays of references, `operateWith` must avoid `null` references

Iterative ascendent listing of array of references with a `while` loop:

```
int i = initP;
while (i <= endP) {
  if (a[i] != null)      // Checks null value
    operateWith(a[i]);  // Operations with i-th element
  goUp(i);
}
```

Operations on unidimensional arrays
Descendent listing

Iterative descendent listing with a `while` loop:

```
int i = endP;
while (i >= initP) {
  operateWith(a[i]);  // Operations with i-th element
  goDown(i);
}
```

Iterative descendent listing with a `for` loop:

```
for (int i = endP; i >= initP; goDown(i)) {
  operateWith(a[i]);  // Operations with i-th element
}
```

Method `goDown` represents the decrement of the index (usually, `i--`)

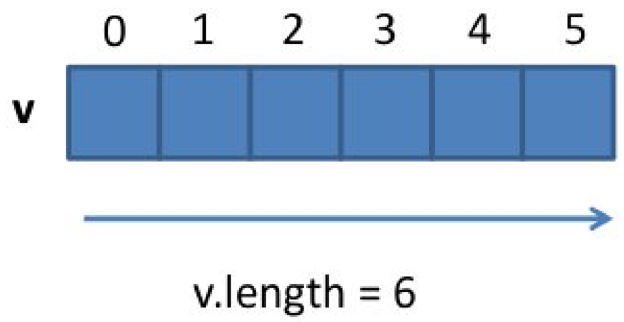
Arrays of references must verify `null` value

Operations on unidimensional arrays
Listing examples

Examples of ***iterative ascendent listing***:

Method to calculate the sum of an array of integers

```
public static int sumIteAsc(int[] v) {
   int sum=0;
   for (int i=0; i<v.length; i++)
      sum = sum + v[i];
   return sum;
}
```



Operations on unidimensional arrays
Listing examples

Method to calculate the mean of an array of integers: slightly modification of the previous

```
public static double meanIteAsc(int[] v) {
    double sum=0;
    for (int i=0; i<v.length; i++) sum = sum + v[i];
    return sum/v.length;
}
```

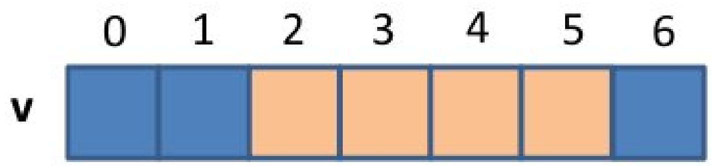
Operations on unidimensional arrays

Listing examples

Method to process a subpart of an array: method for calculating the arithmetical mean of a subarray `a[b .. e]`

```
public static double meanPartial(int[] a, int b, int e){
  double sum = 0;
  for (int i=b; i<=e; i++) sum+=a[i];
  return sum/(e-b+1);
}
```

The number of elements between `b` and `e` ($b \leq e$) are `b-e+1`

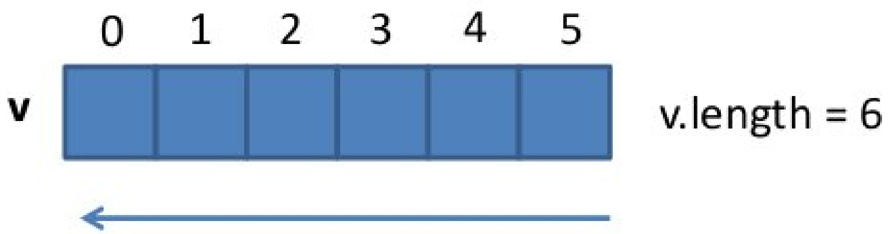


There are 4 elements in `v[2 .. 5]` (5-2+1)

Operations on unidimensional arrays
Listing examples

Examples of ***iterative descendent listing***: method to calculate the sum of an array of integers

```
public static int sumIteDesc(int[] v) {
  int sum=0;
  for (int i=v.length-1; i>=0; i--)
      sum = sum + v[i];
  return sum;
}
```



The loop stops when `i`< 0, after processing element 0 of the array (i.e., `i==-1`)

Operations on unidimensional arrays

Listing examples

Examples of ***iterative ascendent listing***: method to calculate the maximum in the array of `String` `a`

```
public static int max(String[] a) {
  int posMax = 0;
  for (int i=1; i<a.length; i++)
    if (a[i].compareTo(a[posMax])>0) posMax = i;
  return posMax;
}
```

Listing starts in position 1, since position 0 was treated as assuming that the initial default maximum is in the first position

Reminder: primitive datatypes are compared with `<`, `<=`, `>`, `>=`, `==`, `!=`, whereas references (objects) are compared by using `compareTo` or `equals`

This solution supposes there are no `null` positions

Operations on unidimensional arrays

Listing examples

Examples of ***iterative descendent listing***: method to calculate the position of the maximum in the array of `double a`

```
public static int maxP(double[] a) {
  int posMax = a.length-1;
  for (int i=a.length-2; i>=0; i--)
     if (a[i]>a[posMax]) posMax = i;
  return posMax;
}
```

Listing starts in position `a.length-2`, since position `a.length-1` was treated as assuming that the initial default maximum is in the last position

Operations on unidimensional arrays
Searching schemes

Searching: determine if an element in the array accomplishes a given property

Problems that can obtain the solution without knowing all data:

- Obtain the ***first element*** that accomplishes a property
- Operate on data items ***until a condition is achieved***
- . . .

Searching access the different positions (similar to listing process), but uses `boolean` datatype to check the finishing condition

Required control on visited and must-be-visited positions to solve the problem

Operations on unidimensional arrays
Searching schemes

There are two main searching schemes:

- ***Linear search***: size of data items to be searched reduced in one in each step; usable in any array

- ***Binary (or dichotomic) search***: size of data items to be searched reduced by a half in each step; only on ***ordered arrays*** (e.g., from lower to higher), but it is much faster

Operations on unidimensional arrays

Ascendent searching

Ascendent searching that uses a ***boolean variable*** that represents if any element `a[i]` accomplishes a property given by method `prop`:

```
int i=beginning, j=end;
boolean found=false;
while (i<=j && !found) {
   if (prop(a[i])) found=true;
   else goUp(i);
}
// Solve search
if (found) ...    // a[i] accomplishes the property
else ...          // no element accomplishes the property
```

Operations on unidimensional arrays

Ascendent searching

For arrays of reference `null` positions must be avoided

```
int i=beginning, j=end;
boolean found=false;
while (i<=j && !found) {
   if (a[i] != null && prop(a[i])) found=true;
   else goUp(i);
}
// Solve search
if (found) ...    // a[i] accomplishes the property
else ...          // no element accomplishes the property
```

`a[i] != null` must be the first comparison to avoid errors (shortcut operator)

Operations on unidimensional arrays

Descendent searching

Descendent searching that uses a ***boolean variable*** that represents if any element `a[i]` accomplishes a property given by method `prop`:

```
int i=beginning, j=end;
boolean found=false;
while (j>=i && !found) {
   if (prop(a[j])) found=true;
   else goDown(j);
}
// Solve search
if (found) ...    // a[j] accomplishes the property
else ...          // no element accomplishes the property
```

Operations on unidimensional arrays

Searching strategies

Ascendent searching that ***breaks the loop*** by using `break`

```
int i=beginning, j=end;
while (i<=j) {
 if (prop(a[i])) break; // Successful search
 goUp(i);
}
if (i<=j) ...  // Successful
else ...       // Since i>j, no element accomplishes the property
```

Operations on unidimensional arrays
Searching strategies

Ascendent searching that ***breaks the loop*** by using `return` and ***terminating the current method***

```
int i=beginning, j=end;
while (i<=j) {
 if (prop(a[i])) return i; // Successful search
 goUp(i);
}
// i>j and no element accomplishes the property
return -1;
```

Only possible for methods that return `int` values (positions of the array)

Operations on unidimensional arrays
Searching strategies

Ascendent searching with ***sure success***: there is an element `a[i]` (`i` unknown) that is known to accomplish the property `prop` beforehand

```
int i=beginning;
while (!prop(a[i]))
  goUp(i);

// Solve the search: the element in
// position i accomplishes the property
```

Usually an element that fulfils the property is included in the array: ***sentinel***

Consequently the search is called ***sentinel-based search***

Operations on unidimensional arrays

Searching strategies

Linear iterative search structure (without success guarantee) with a ***guard that evaluates the property*** `prop`

```
int i=beginning, j=end;
while (i<=j && !prop(a[i]))
      goUp(i);

// Loop finishes when:
// i <= end -> a[i] accomplishes property, or
// i is end+1

// Solve the search
if (i<=end) ...   // prop(a[i]) is true
else ...          // no element a[i]
                  // accomplishes prop(a[i])
```

Operations on unidimensional arrays

Searching examples

Ascendent search in a `String` array without `null` references ans without success guarantee (-1 is returned when not found)

```
public static int searchPos(String[] a, String data) {
  int i = 0;
  while (i<a.length && !a[i].equals(data)) i++;
  if (i<a.length) return i;
  else return -1; // or directly: return -1;
}
```

Descendent search in a `double` array without success guarantee (-1 is returned when not found)

```
public static int searchPos(double[] a, double data) {
  int i = a.length-1;
  while (i>=0 && a[i]!=data) i--;
  return i;
}
```

Operations on unidimensional arrays
Searching examples

Check if all elements of the array are greater than a given data item

```
public static boolean itIsGreater(double[] a, double data) {
  int i = 0;
  while (i<a.length && a[i]>data) i++;
  return (i>=a.length);
}
```

Returns the position of the first element of the array whose value is greater than the sum of the previous elements (-1 when it does not exist)

```
public static int searchPosGreatSum(int[] a) {
  int i = 0, sum = 0;
  while (i<a.length && a[i]<=sum) { sum+=a[i]; i++; }
  if (i<a.length) return i;
  else return -1;
}
```

Operations on unidimensional arrays
Combining searching and listing

Some problems require combining the two strategies

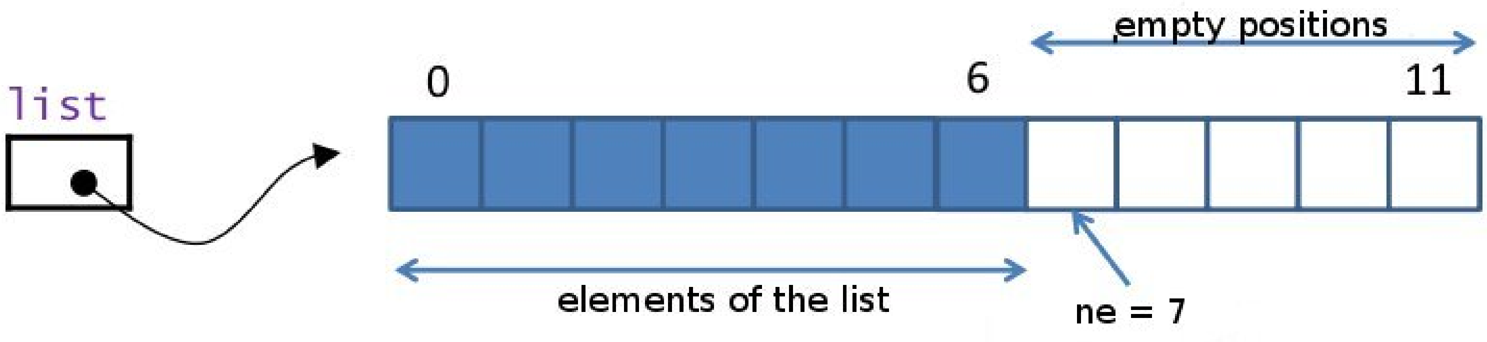
E.g., for a `String` array, determine for each `String` its first repetition

Result: a `String` with the corresponding `String` and the two positions

```
public static String listDuplicates(String[] a) {
  String res = "";
  for (int i=0; i<a.length; i++) {
    int j = i+1;
    while (j<a.length && !a[i].equals(a[j])) j++;
    if (j<a.length)
      res+=a[i] + " duplicated in: " + i + " and " + j + "\n";
  }
  return res;
}
```

Operations on unidimensional arrays
Dynamic data structures by using arrays

An array can represent a list of items where elements can be added and removed:



Elements in consecutive positions `[0, ne-1]` (first free position is `ne`)

The array size must be the maximum number of elements to be stored

```
datatype[] list = new datatype[MAX_ELEM];
int ne = 0;
```

Operations on unidimensional arrays

Dynamic data structures by using arrays

To add a new element to the list:

```
if (ne<MAX_ELEM) list[ne++] = x;
else ... // Full list: element x cannot be added
```

Element removal: the continuity of the elements and their order must be kept

All elements after the removed position must move to the left one position

```
int i = 0;
while (i<ne && list[i] differentFrom x) i++;

if (i<ne) {
   for (int j=i+1; j<ne; j++) list[j-1] = list[j];
   ne--;
}
else ... // Element not found: removal not possible
```

Operations on unidimensional arrays
Dynamic data structures by using arrays

It is usual to implement it in a class, e.g., dynamic list of integers:

```
public class ListInt {
  private static final int MAX_ELEM=100;
  private int[] list;
  private int ne;

  public ListInt() { list = new int[MAX_ELEM]; ne = 0; }
  public void add(int x) { if (ne<MAX_ELEM) list[ne++] = x; }
  public void remove(int x) {
    int i = 0;
    while (i<ne && list[i]!=x) i++;
    if (i<ne) {
       for (int j=i+1; j<ne; j++) list[j-1] = list[j];
       ne--;
    }
  }
}
```

Contents

Multidimensional arrays
Motivation

Possible example: measure temperatures in a zone for all the days of the year

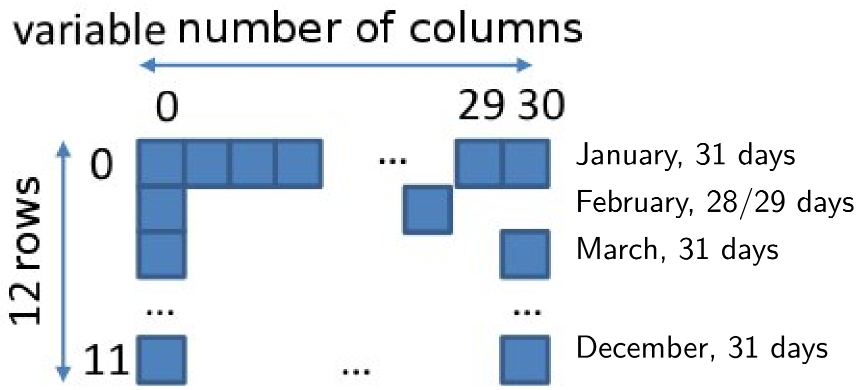
Possible solution: array `temp`, size 366 (for leap years) which stores in correlative order the temperature data for each day

Jan 1st → `temp[0]`   Feb 3rd → `temp[33]`   Jul 2nd → `temp[183]`

Problem of this representation: difficult access (e.g., which position is March 17th?)

Multidimensional arrays
Motivation

Alternative solution: multidimensional array, 12 rows and different number of columns



- Simpler calculations for accesing to each day data
- Effective memory usage: size of the array fits to the number of data items
  - Array with 12 components
  - Each component is itself an array with a different number of components

This is an example of ***multidimensional array*** (array of arrays)

Multidimensional arrays
Definition

- ***Definition***: a ***multidimensional array*** is an array whose components are itself arrays
- The ***dimension*** of an array is the number of nested definitions that includes
- Each nested array is one dimension lower
- The dimension defines how many indexes are needed to access to the basic data items stored in the array

Multidimensional arrays

Declaration and use

- ***Declaration and creation***
  - Declaration: *type* `[][]...[]` ***arrayVar***; (as many `[]` as dimensions)
  - Initialisation: ***arrayVar*** `= new` *type*`[`s_1`][`s_2`]...[`s_n`]`; (only s_1 mandatory)
  - Altogether: *type* `[][]...[]` ***arrayVar*** `= new` *type*`[`s_1`][`s_2`]...[`s_n`]`;

- ***Array access operator***: `[]` must be repeated for each dimension

  arrayVar `[` $\textbf{\textit{index}}_1$ `] [` $\textbf{\textit{index}}_2$ `] ...[` $\textbf{\textit{index}}_n$ `]`

  $\textbf{\textit{index}}_k$ must evaluate to a valid index for dimension k of array ***arrayVar***

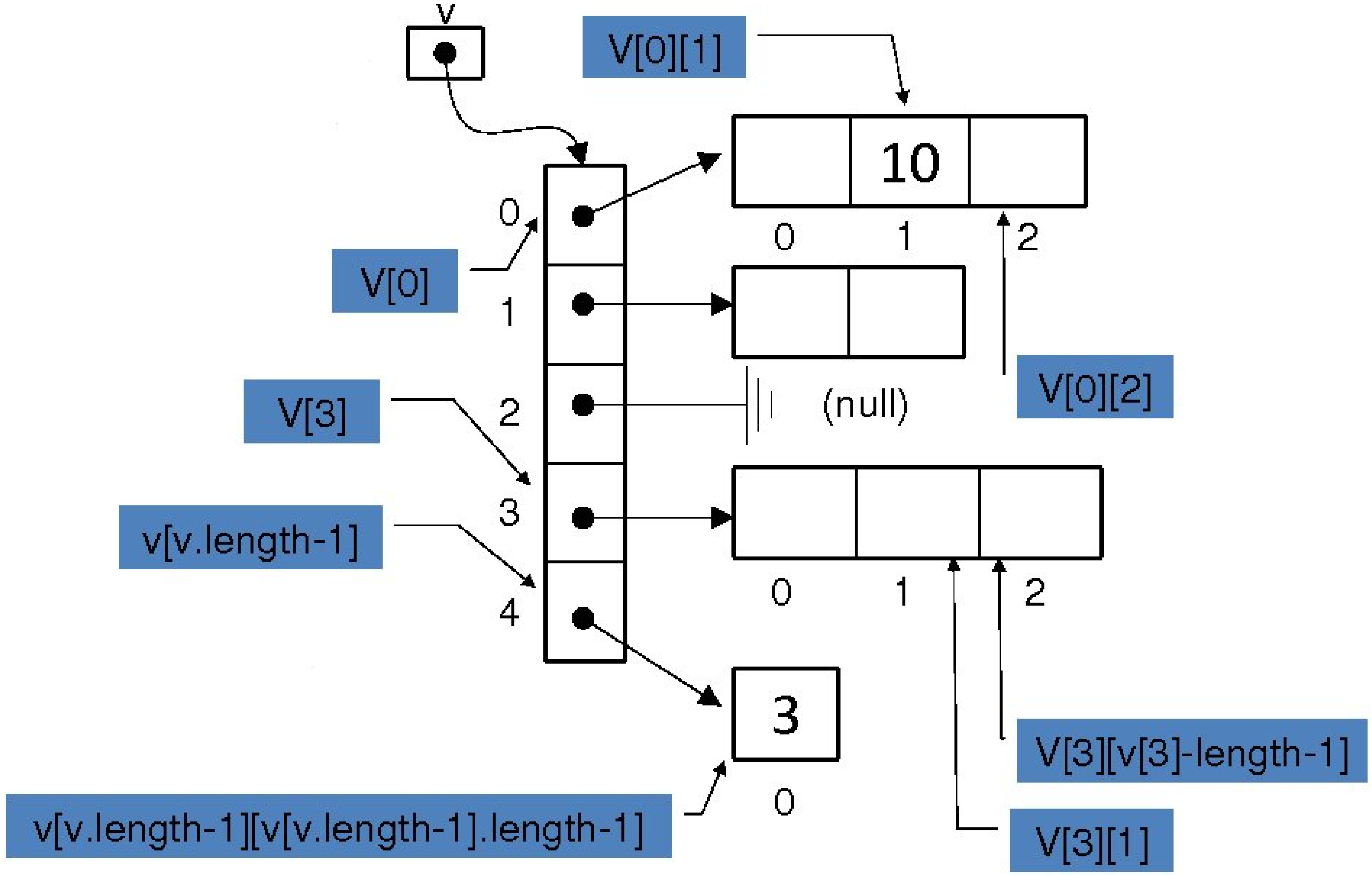
- ***Literal values***: represented as a sequence of elements between braces (`{}`) and separated by comma

  E.g., `{{1,2},{1,2,3},{1}}` is a 2-dim integer array with three 1-dim arrays

Multidimensional arrays

Bidimensional arrays

```
int[][] v;
v = new int[5][];
v[0] = new int[3];
v[1] = new int[2];
v[3] = new int[3];
v[4] = new int[1];
v[0][1] = 10;
v[4][0] = 3;
```



Errors:

```
v[2] = -3;      // Compilation error
v[4][1] = -3;   // Execution error: ArrayIndexOutOfBoundsException
```

Multidimensional arrays
Example: temperature

Multidimensional array with 12 rows and different number of columns:

```
double[][] meanTemp = new double[12][]; // meanTemp.length is 12
meanTemp[0] = new double[31]; // Indexes from 0 to 30
meanTemp[1] = new double[29]; // Indexes from 0 to 28
// ...
meanTemp[2][4] = 78.0; // Temperature for Mar 5th
```

Multidimensional arrays
Example: temperature

Another implementation previously defining the number of days for each month

```
final int[] NUM_DAYS = {31,28,31,30,31,30,31,31,30,31,30,31};
// NUM_DAYS[i] = number of days for each month, 0<=i<=11

double[][] meanTemp = new double[12][];
// meanTemp[i] represents month, 0<=i<=11

for (int i=0; i<meanTemp.length; i++)
        meanTemp[i] = new double[NUM_DAYS[i]];
// number of elements of meanTemp[i] is NUM_DAYS[i], 0<=i<=11
```

`meanTemp[i][j]` represents the mean temperature for day `j+1` in month `i+1` (e.g., `meanTemp[3][14]` is the mean temperature for Apr 15th)

Multidimensional arrays

Example: temperature

Alternative using indexes from 1:

```
final int[] NUM_DAYS = {0,31,28,31,30,31,30,31,31,30,31,30,31}
// NUM_DAYS[i] = number of days for each month, 1<=i<=12
// NUM_DAYS[0] = 0 (no month associated)

double[][] meanTemp = new double[13][];
// meanTemp[i] represents month, 1<=i<=12
// meanTemp[0] must be null (no month associated)

for (int i=1; i<meanTemp.length; i++)
        meanTemp[i] = new double[NUM_DAYS[i]+1];
// number of elements of meanTemp[i] is NUM_DAYS[i]+1, 1<=i<=12
```

Now, `meanTemp[i][j]` represents the mean temperature for day `j` in month `i` (e.g., `meanTemp[3][14]` is the mean temperature for Mar 14th)

Multidimensional arrays

N-dimensional arrays

For dimensions greater than two the same rules apply:

- Three dimensions:

```
int[][][] three = {{{1}},{{1,2},{3,4}},{{1},{2}}};
int[][] two = {{5,4},{6,7}};
three[1] = two;
three[1][0][1] = 6;
```

- Four dimensions:

```
int[][][][] four = new int[10][2][][];
for(int i=0; i<four.length; i++)
   for(int j=0; j<four[i].length; j++)
      four[i][j] = new int[3][3];
```

Or:

```
int[][][][] four = new int[10][2][3][3];
```

Multidimensional arrays
Listing multidimensional arrays

Listing and searching processes need as many nested loops as dimensions

E.g., average temperature of the year (last representation):

```
double s=0.0, avg;
for (int m=1;m<meanTemp.length;m++)
  for (int d=1;d<meanTemp[m].length;d++)
    s+=meanTemp[m][d];
avg=s/365;
```

E.g., minimum of three dimensional `double` array `m`:

```
double min=m[0][0][0];
for (int i=0;i<m.length;i++)
  for (int j=0;j<m[i].length;j++)
    for (int k=0;k<m[i][j].length;k++)
      if (m[i][j][k]<min) min=m[i][j][k];
```

Multidimensional arrays

Searching multidimensional arrays

E.g., looking for an element lower than the first element of an `int` bidimensional array `a`:

```
int i=0, j=0;
while (i<a.length && a[i][j]>=a[0][0]) {
  j=0; while (j<a[i].length && a[i][j]>=a[0][0]) j++;
  if (j>=a[i].length) i++;
}
```

E.g., looking for a subarray of a `double` bidimensional array `b` that sums a negative number:

```
double s=0;
int i=0, j;
while (i<b.length && s>=0) {
  s=0; for (j=0;j<b[i].length;j++) s+=b[i][j];
  if (s>=0) i++;
}
```

Multidimensional arrays
Searching multidimensional arrays

E.g., temperatures; search for a day that was 40^{o}

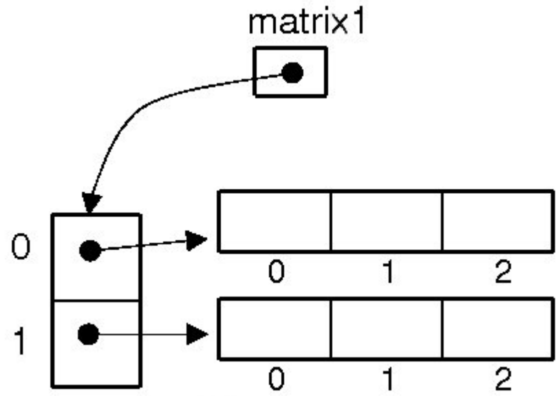
```
public static void main(String[] args) {
  final int[] NUM_DAYS = {0,31,28,31,30,31,30,31,31,30,31,30,31};
  double[][] meanTemp = new double[13][];
  for (int i=1; i<meanTemp.length; i++) meanTemp[i] = new double[NUM_DAYS[i]+1];
  .....
  int i = 1, j; boolean found = false;
  while (i<meanTemp.length && !found) {
    j = 1;
    while (j<meanTemp[i].length && !found) {
      found = (meanTemp[i][j]==40);
      if (!found) j++;
    }
    if (!found) i++;
  }
  if (found) System.out.println("40 degrees in day " + j + " of month " + i);
  else System.out.println("No day with 40 degrees");
}
```

Contents

Matrices

- *Matrices*: 2-dim array with all second dimension arrays of same size
- They can be defined with an only `new` sentence
- E.g., matrix with two rows and three columns:

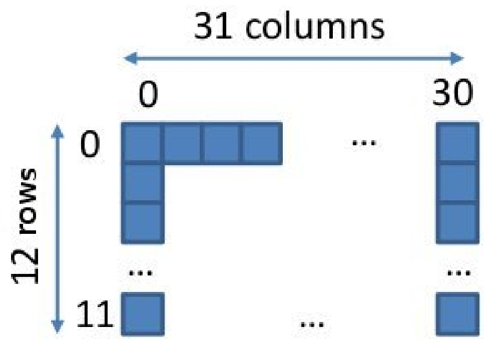
```
double[][] matrix1 = new double[2][3];
```



- Another option (similar to general bidimensional arrays):

```
double[][] matrix1 = new double[2][];
matrix1[0] = new double[3];
matrix1[1] = new double[3];
```

Temperature example represented by a matrix of 12×31:



- Accessing to each day data as simple as for general bidimensional
- Simpler definition (an only `new`)
- This representation wastes memory

Matrices

Listing matrices

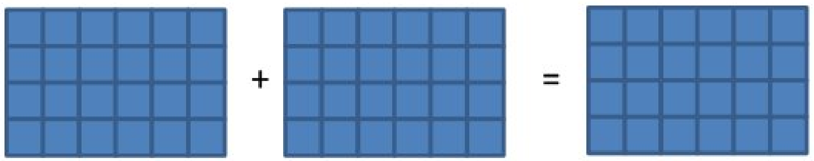
Similar scheme to that of general bidimensional arrays

E.g., keyboard initialisation:

```
for(int i=0; i<matrix1.length; i++)
  for(int j=0; j<matrix1[i].length; j++)
      matrix1[i][j] = kbd.nextDouble();
```

E.g., summing two double matrixes of size N×M, assuming that both `m1` and `m2` are initialised and they are of the same size

```
public static double[][] sumM(double[][] m1, double[][] m2) {
    double[][] res= new double[m1.length][m1[0].length];
    for (int i=0; i<m1.length; i++)
      for (int j=0; j<m1[i].length; j++)
          res[i][j]= m1[i][j] + m2[i][j];
    return res;
}
```

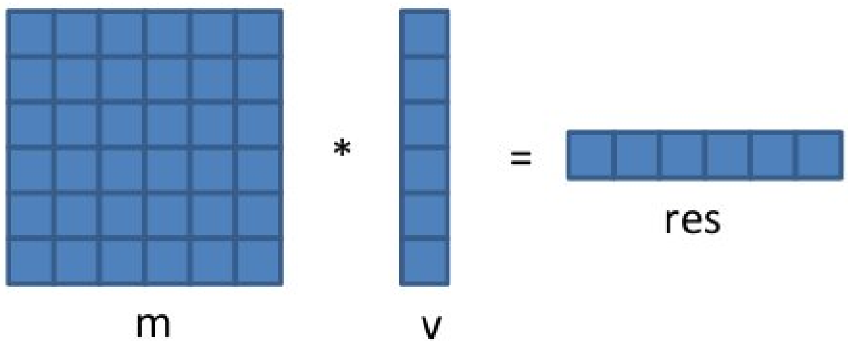


Matrices

Listing matrices

E.g., matrix-vector product; number of columns of the matrix must be equal to number of components of the vector (unidimensional array)

```
public static double[] matTimesVec(double[][] m, double[] v) {
    double[] res = new double[m.length];
    for (int i=0; i<m.length; i++)
      for (int j=0; j<v.length; j++)
         res[i] += m[i][j] * v[j];
    return res;
}
```



Matrices

Searching matrices

Similar scheme to that of general bidimensional arrays

E.g., detecting if a square `int` matrix `m` is upper triangular

```
public static boolean upTriang(int [][] m) {
  int i=0, j;
  boolean ut=true;

  while (i<m.length && ut) {
    j=0;
    while (j<i && ut) {
      if (m[i][j]!=0) ut=false;
      j++;
    }
    i++;
  }
  return ut;
}
```

Contents

Other direct access problems

- Arrays allow access in constant time to each component given its position; i.e., given `a[0..1000]`, the time for accessing `a[0]` is (theoretically) the same that for accessing `a[999]`

- This allows to implement some very efficient algorithms:
  - Represent a set of natural numbers
  - Calculate the mode (most frequent value) of a set of natural numbers

Other direct access problems

Set of natural numbers

Set of natural numbers S of the interval `[0..n]`

Methods: `add`, `cardinal`, `pertains`, `randomSet`

Classic representation: with array of `int` and size (`int`)

```
public class Set{
  private int [] set;
  private int last;

  public Set(int l) {
    set = new int[l]; last = 0;
  }

  public void add(int x) {
    if (!pertains(x) && last < set.length) {
      set[last] = x; last++;
    }
  }
```

Other direct access problems

Set of natural numbers

```
  public int cardinal() { return last; }

  public boolean pertains(int x) {
    int i = 0;
    while ( (i < last) && (set[i] != x) ) i++;
    return i < last;
  }

  public void randomSet() {
    int l=0, x;
    last = (int) Math.floor(Math.random()*set.length);
    while (l < last) {
      x = (int) Math.floor(Math.random()*Integer.MAX_VALUE);
      if (!pertains(x)) { set[l] = x; l++; }
    }
  }
}
```

Other direct access problems

Set of natural numbers

Alternative representation: with a `boolean` array (`set[i]` is `true` when `i`$\in S$)


```
public class Set {
  private boolean[] set;
  private int last;

  public Set(int l) {
     set = new boolean[l+1]; last = l;
  }

  /** 0<=x<=last */
  public void add(int x) { set[x] = true; }
```

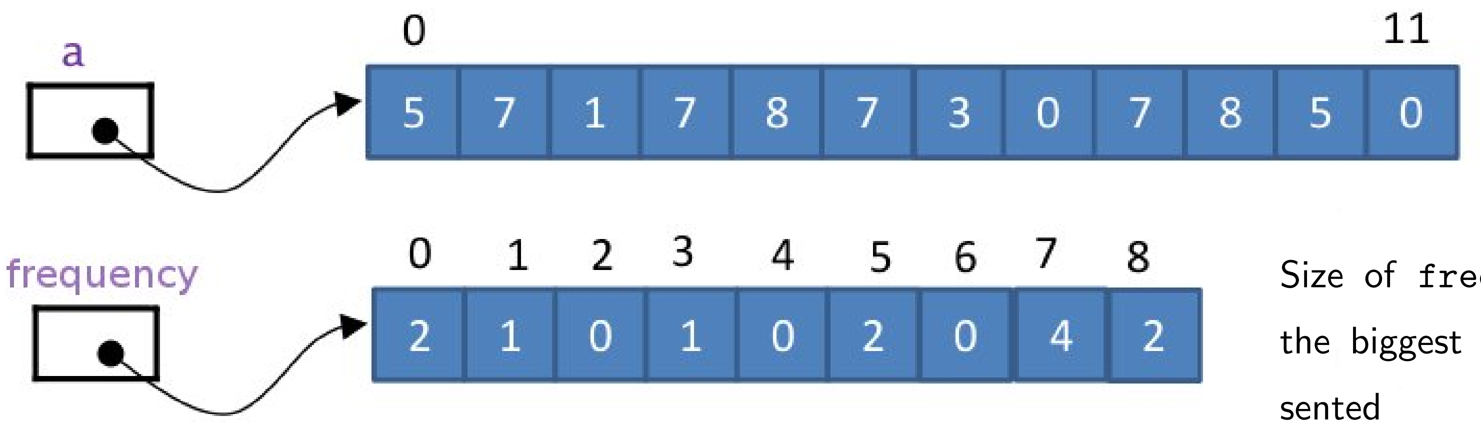
Other direct access problems
Set of natural numbers

```
  public int cardinal() {
      int card = 0;
      for (int i=0; i<set.length; i++)
         if (set[i]) card++;
      return card;
  }

  /** 0<=x<=last */
  public boolean pertains(int x) { return set[x]; }

  public void randomSet() {
      for (int i=0; i<set.length; i++)
         set[i] = (Math.random()>=0.5);
  }
}
```

With this representation the membership of a natural number to the set can be obtained in a very fast way (time independent of the number of elements)

Other direct access problems
Mode of a set of natural numbers

- The mode of a set is the element that appears more frequently
- Two-step strategy:
  1. Sequential ascendent listing and auxiliar array with counters `frequency`, (`frequency[i]` gives the number of times that `i` appears in the array)
  2. The mode is the position maximum value of the `frequency` array
- E.g., for a={5,7,1,7,8,7,3,0,7,8,5,0} with n = 8 (biggest number in data)



Size of `frequency` according to the biggest number to be represented

Other direct access problems

Mode of a set of natural numbers

Calculate the mode of an array that contains elements in the range `[0..n]`

```
public static int mode0ToN(int[] a, int n) {
    // Build array between 0 and n
    int[] frequency = new int[n+1];

    // Listing a and obtaining frequencies
    for (int i=0; i<a.length; i++) frequency[a[i]]++;

    // The mode is the index of the maximum of the frequency array
    int mode = 0;
    for (int i=1; i<frequency.length; i++)
        if (frequency[i]>frequency[mode]) mode = i;

    return mode;
}
```