Unit 7

Arrays and matrices

Summary

- Arrays as collections of elements
- Declaration of array variables
- Creation of array objects
- Access to the elements of an array object
- Expressions that represent array objects
- Parameters passed to a program
- Matrices (as arrays of arrays)

7.1 Arrays

An **array object** (or simply **array**) contains a collection of elements of the same type, each of which is indexed (i.e., identified) by a number. A **variable of type array** contains a reference to an array object.

To use an array in Java we have to:

1. declare a variable of type array that allows us to refer to an array object;
2. construct the array object specifying its dimension (number of elements of the array object);
3. access the elements of the array object through the array variable in order to assign or obtain their values (as if they were single variables).

7.2 Declaration of array variables

To use an array we first have to declare a variable that refers to the array.

Declaration of array variables

Syntax:

```
type[] arrayName;
```

where

- *type* is the type of the elements of the array to which the variable we are declaring will refer
- *arrayName* is the name of the array variable we are declaring

Semantics:

Declares a variable *arrayName* that can refer to an array object with elements of type *type*.

Example:

```
int[] a;     // a is a variable of type reference to an array of integers
```

Note that, by declaring a variable of type reference to an array we have not yet constructed the array object to which the variable refers, and hence the array cannot be used yet.

7.3 Creation of an array object

To use an array, we must first create it, by specifying the number of elements it should have.

Creation of an array object

Syntax:

```
new type[dimension]
```

where

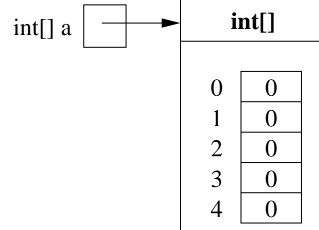
- *type* is the name of the type of the elements of the array object that we want to contruct
- *dimension* is an expression of type `int` that evaluates to a positive (or zero) value and that specifies the dimension of the array

Semantics:

Creates an array object with *dimension* elements of type *type* and returns a reference to the created object. The elements of the array are indexed from 0 to *dimension*`-1` (see later), and each one is initialized to the default value for *type*.

Example:

```
int[] a;          // a is a variable of type array of integers
a = new int[5];   // creation of an array object with 5 elements
                  // or type int associated to the array variable a
```



After we have created an array object associated to an array variable, it is possible to access the single elements of the collection contained in the array. These elements are initialized to the default value for the type (for integers, it is 0, as illustrated in the figure).

7.4 Access to the single elements of an array

Each single element of an array object can be accessed as if it were a separate variable.

Access to the single elements of an array

Syntax:

```
arrayName[index]
```

where

- *arrayName* is the name of the array variable that contains a reference to the array we want to access
- *index* is an expression of type `int` with non-negative value that specifies the index of the element we want to access.

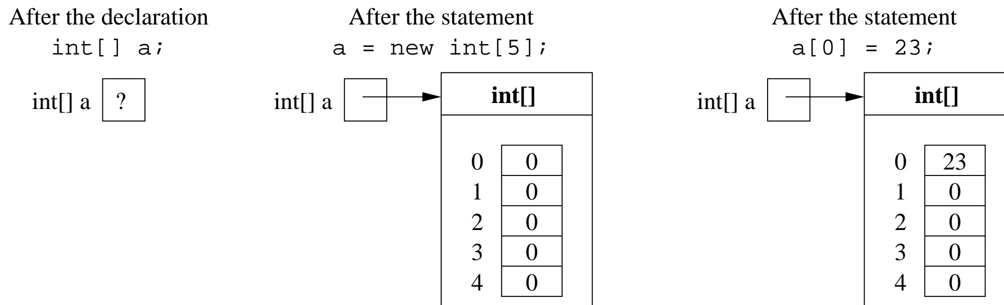
Semantics:

Accesses the element of the array *arrayName* in the position specified by *index* to read or modify it.

If the array *arrayName* contains N elements, the evaluation of the expression *index* must return an integer in the interval $[0, N-1]$; if this is not the case, a runtime error occurs when the program is run.

Example:

```
int[] a;        // a is a variable of type array of integers
a = new int[5]; // creation of an array object with 5 elements of type int;
                // the array object is referenced by the array variable a
a[0] = 23;      // assignment to the first element of the array
a[4] = 92;      // assignment to the last element of the array
a[5] = 16;      // ERROR!  the index 5 is not in the range [0,4]
```



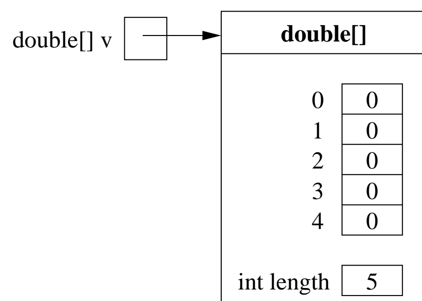
Note: It is very important to remember that, when an array contains N elements ($N = 5$ in the example), the indexes used to access the elements must necessarily be integers in the interval $[0, N-1]$. Otherwise, a runtime error occurs when the program is executed. In the example, an error is signaled when the statement `a[5]=16;`.

7.5 Number of elements of an array: instance variable `length`

Each array object contains, besides the collection of elements, a public instance variable (non modifiable) called `length` that stores the number of elements of the array. Hence, by accessing the variable `length` it is possible to obtain the number of elements of an array.

Example:

```
double[] v;
v = new double[5];
System.out.println(v.length); // prints 5
```



For brevity, we will in general not show the instance variable `length` when illustrating array objects, knowing that it is always present, as in the precedent figure.

7.6 Expressions that denote array objects

In Java, it is possible to write expressions that denote array objects, similarly to what we have seen for strings, where, e.g., `"ciao"` denotes a `String` object. An expression that denotes an array object is a list of expressions, of the type of the elements of the array, of the form:

{ *expression1* , *expression2* , ..., *expressionk* }

and it denotes an array object of *k* elements.

Example:

```
int[] v = { 4, 6, 3, 1 };
```

is equivalent to:

```
int[] v = new int[4];
v[0] = 4; v[1] = 6; v[2] = 3; v[3] = 1;
```

Note: While a literal of type `String`, such as `"ciao"` can be used anywhere in the body of a method to denote a string object, the expressions that denote an array can be used *only to initialize an array when it is declared.* The following example shows a fragment of code that is wrong:

```
int[] v;
v = { 4, 6, 3, 1 };  // ERROR!
```

7.7 Example: sum of the elements of an array of integers

We develop a static method, `sumArrayValues()`, that takes as parameter an array of integers, and returns the sum of the values of the array elements.

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

Example of usage:

```
public static void main(String[] args) {
  // creation of an array of 100 elements
  int[] x = new int[100];

  // we assign to the element of x of index i the value 2*i
  for (int i = 0; i < x.length; i++)
    x[i] = 2*i;

  // print out the sum of the 100 array elements
  System.out.println(sumArrayValues(x));
}
```

7.8 Example: exhaustive search of an element in an array

We develop a static predicate, *searchArray*, that takes as parameters an array of strings and a string, and returns `true`, if the string is present in the array, and `false` otherwise.

```
public static boolean searchArray(String[] v, String e) {
  for (int i = 0; i < v.length; i++)
    if (e.equals(v[i]))
      return true;
  return false;
}
```

Example of usage:

```
public static void main(String[] args) {
  // creation of an array x of 3 strings
  String[] x = { "one", "two", "three" };

  // search the string "two" in the array x
  if (searchArray(x, "two"))
    System.out.println("present");
  else
    System.out.println("not present");  // this will not be printed out
}
```

7.9 Example: search of the maximum element in an array of numbers

We develop a static method, `maxArray()`, which takes as parameter an array of integers and returns the maximum value in the array. We assume that the array contains at least one element. A possible realization is the following:

```
public static long maxArray(long[] v) {
  long max = v[0];
  for (int i = 1; i < v.length; i++)
    if (v[i] > max) max = v[i];
  return max;
}
```

Notice that the method uses a variable `max` to hold the current maximum while iterating over the elements of the array. The variable `max` is initialized to the value of the first element of the array, which by assumption must exist. Then `max` is updated whenever a bigger element is found. Hence, at the end of the loop, the value of `max` will coincide with the value of the maximum element of the array, and such a value is returned.

Example of usage:

```
public static void main(String[] args){
 long[] x = { 42, 97, 31, -25 };     // creation of an array x of 4 long
 System.out.println(maxArray(x));    // prints out 97
}
```

7.10 Example: inverting the contents of an array

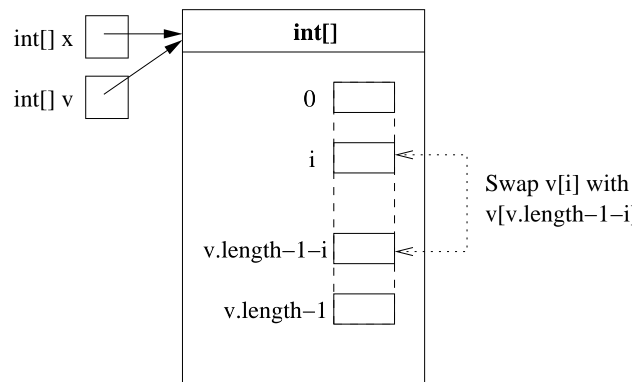
We develop a static method `invertArray()` that takes as parameter an array of integers and modifies it by inverting the order of its elements (the first one becomes the last one, the second one the last but one, etc.):

```
public void invertArray(int[] v) {
  for (int i = 0; i < v.length/2; i++) {
    int temp;
    temp = v[i];
    v[i] = v[v.length-1-i];
    v[v.length-1-i] = temp;
  }
}
```

Example of usage:

```
public static void main(String[] args) {
  int[] x = { 5, 3, 9, 5, 12};    // creation of an array x of 5 int
  for (int i = 0; i < 5; i++)     // prints out 5 3 9 5 12
    System.out.println(x[i]);
  invertArray(x);                 // inverts the array x
  for (int i = 0; i < 5; i++)     // prints out 12 5 9 3 5
    System.out.println(x[i]);
}
```

Notice that, when the array `x` is passed as an actual parameter to the method `invertArray()`, the formal parameter `v` refers to the same array object to which `x` refers. Hence, all modifications done to the array inside the method are done on the same array to which `x` refers, and hence will be visible to the calling method (in this case, `main()`).



7.11 Parameters passed to a program

By looking at the header of the `main()` method, we can observe that it takes as parameter an array of strings:

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

Such an array contains the strings that are passed as arguments to the program when it is run from command line.

Example: Program that prints out the arguments passed on the command line:

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

Example of usage:

```
Terminal — bash — 94x10
calvim:~/code calvanese$ java PrintArguments here are some command line arguments
here
are
some
command
line
arguments
calvim:~/code calvanese$ []
```

Note that the six strings `"here"`, `"are"`, `"some"`, `"command"`, `"line"`, and `"arguments"` are the six arguments that are passed to the program when it is executed through the `java` command.

7.12 Exercise: apartment owners

Specification:

Realize a Java class whose objects maintain some information on the owners of apartments. Each object of the class should contain a string that indicates the name of the owner, and an array of 10 slots of type string, indexed by the numbers from 0 to 9, that can contain each the address of an apartment owned by that owner (or `null`, if the slot is empty). The class should export the following functionalities:

- a constructor that takes as parameter the name of the owner, and creates an object with the specified owner and in which all slots of the array are initially empty;
- a method that returns the owner of an apartment;
- a method that returns the address contained in a slot (or `null`, if the slot is empty);
- a method to assign the address of an apartment to a slot;
- a method that returns the number of apartments (i.e., of non-empty slots);

- a method that reorganizes the addresses in such a way that they are contained in the first consecutive slots of the array;
- a method `toString()`, that overrides the `toString()` method inherited from `Object`, and returns a string containing the information about the object.

7.13 The class `ApartmentOwner`: representation of the objects

Let us represent the objects of the class `ApartmentOwner` by the following instance variables:

- the name of the owner, by an instance variable `owner`, of type `String`;
- the slots containing the addresses of the apartments, by an instance variable `apartments`, of type `String[]`, i.e., array of strings.

At this point we can write:

```
public class ApartmentOwner {
  // representation of the objects of the class
  private String   owner;
  private String[] apartments;

  // public methods that realize the requested functionalities
}
```

7.14 The class `ApartmentOwner`: public interface

We can now choose the interface of the class, through which the clients can make use of the objects of the class `ApartmentOwner`. Specifically, for each functionality we have to define a public method that realizes it and determine its header.

This leads us to the following skeleton for the class `ApartmentOwner`:

```
public class ApartmentOwner {
  // representation of the objects of the class
  private String   owner;
  private String[] apartments;

  // public methods that realize the requested functionalities
  public ApartmentOwner(String name) { ... }
  public String getOwner() { ... }
  public String getApartment(int slot) { ... }
  public void setApartment(String address, int slot) { ... }
  public int countApartments() { ... }
  public void reorganizeApartments() { ... }
  public String toString() { ... }
}
```

7.15 The class `ApartmentOwner`: realization of the methods

```
public class ApartmentOwner {
  private String   owner;
  private String[] apartments;

  public ApartmentOwner(String owner) {
    this.owner = owner;
    apartments = new String[10];
  }

  public String getOwner() {
    return owner;
  }

  public String getApartment(int slot) {
    return apartments[slot];
```

```
  }

  public void setApartment(String address, int slot) {
    apartments[slot] = address;
  }

  public int countApartments() {
    int num = 0;
    for (int i = 0; i < 10; i++)
      if (apartments[i] != null)
        num ++;
    return num;
  }

  public void reorganizeApartments() {
    int empty = -1;  // index of the first empty slot
    for (int i = 0; i < 10; i++) {
      if (apartments[i] == null && empty == -1)
        empty = i;
      if (apartments[i] != null && empty != -1) {
        apartments[empty] = apartments[i];
        apartments[i] = null;
        empty++;
      }
    }
  }

  public String toString() {
    String ris = "";
    ris += "Owner: " + owner;
    for (int i = 0; i < 10; i++)
      if (apartments[i] != null)
        ris += "\nApartment " + i + ": " + apartments[i];
    return ris;
  }
}
```

Example of usage:

```
public class TestApartmentOwner {

  public static void main (String[] args) {
    ApartmentOwner p = new ApartmentOwner("Mario Rossi");
    p.setApartment("Via della Camilluccia, 29", 0);
    p.setApartment("Largo di Torre Argentina, 42", 1);
    p.setApartment("P.zza Conca D'Oro, 9", 2);
    p.setApartment("Via del Corso, 30", 5);
    p.setApartment("Via Trionfale, 2500", 8);
    System.out.println(p);

    System.out.println();
    System.out.println(p.getOwner() + " has " +
                       p.countApartments() + " apartments");
    System.out.println("Apartment 2: " + p.getApartment(2));

    System.out.println();
    p.reorganizeApartments();
    System.out.println(p);
  }
}
```

7.16 Matrices

A **matrix** is a collection of elements of the same type, organized in the form of a table. Each element is indexed by a pair of numbers that identify the row and the column of the element.

A matrix can be represented in Java through an array, whose elements are themselves (references to) arrays representing the various rows of the matrix.

Declaration of a matrix

A matrix is declared in the following way (as an array of arrays):

```
int[][] m;   // declaration of an array of arrays (matrix) m
```

Creation of a matrix

As for arrays, before a matrix can be used, it must be created.

Example: Creation of a 3x5 matrix accessible through the variable `m`:

```
// creation of an array object of 3 elements,
// each of which will contain the reference to a row of the matrix
m = new int[3][];

m[0] = new int[5];  // creation of row 0 of the matrix (5 columns)
m[1] = new int[5];  // creation of row 1 of the matrix (5 columns)
m[2] = new int[5];  // creation of row 2 of the matrix (5 columns)
```

Note that, by creating the rows one at a time, it is also possible to have rows with different dimensions. This would correspond to a matrix with an irregular shape (as opposed to a rectangular matrix).

In Java, there is an equivalent, more compact way of creating a rectangular matrix, without having to construct the rows one at a time:

```
// creation of an array object of 3 elements,
// each of which is an array of 5 integers (3x5 matrix)
m = new int [3][5];
```

Note that, in this way, all rows of the matrix necessarily have the same number of elements.

Access to the elements of a matrix

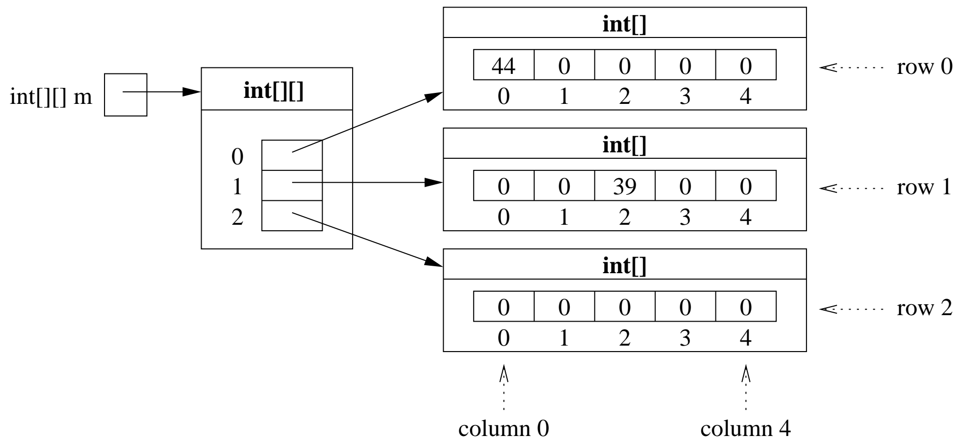
The single elements of the matrix can be treated like ordinary variables of the corresponding type. To access them, we have to specify the index of the row and the index of the column of the element.

Example:

```
// assignment of a value to the element of m in row 1, column 2
m[1][2] = 39;

// assignment of a value to the element of m in row 0, column 0
m[0][0] = 44;

// access to the element of m in row 1, column 2
System.out.println(m[1][2]);   // prints out 39
```



7.17 Expressions that denote matrix objects

Since a matrix is simply an array whose elements are themselves arrays, in the initialization of a matrix we can use an expression that denotes a matrix.

Example:

```
int[][] m = { { 3, 5  },
              { 1, -2 }
            };
```

Note that the definition of the matrix above is equivalent to:

```
int[][] m = new int[2][2];
m[0][0] = 3; m[0][1] = 5;
m[1][0] = 1; m[1][1] = -2;
```

7.18 Number of rows and columns of a matrix

Using the variable `length`, it is possible to obtain the number of rows and of columns of a matrix. Let `m` be a reference to a matrix (i.e., an array of arrays). Then

- `m.length` denotes the **number of rows** of the matrix `m`
- `m[i].length` denotes the **number of columns** of row `i` of matrix `m`

  Note: If all rows have the same number of columns (as in the previous example), we can simply use `m[0].length`.

Example:

```
double[][] v;
v = new double[15][20];
System.out.println(v.length);    // prints out 15
System.out.println(v[0].length); // prints out 20
```

7.19 Example: printing out the elements of a matrix ordered by rows and by columns

We develop two static methods, `printMatrixRows()` and `printMatrixColumns()`, that take as parameter a matrix and print out its elements ordered respectively per rows and per columns.

```
public static void printMatrixRows(short[][] m) {
  for (int i = 0; i < m.length; i++) {    // iterates over the rows
    for (int j = 0; j < m[0].length; j++) // iterates over the elements of row i
      System.out.print(m[i][j] + " ");    // prints the element
    System.out.println();                 // end of the row
  }
}
```

Note that, in order to access all elements of the matrix `m`, we use two nested `for` loops: the outermost loop iterates over the rows (using control variable `i`), while the innermost loop iterates over the elements of row `i` (using control variable `j`). At the end of each row, a newline character is printed out.

```
public static void printMatrixColumns(short[][] m) {
  for (int j = 0; j < m[0].length; j++) { // iterates over the columns
    for (int i = 0; i < m.length; i++)    // iterates over the elements of col j
      System.out.print(m[i][j] + " ");    // prints the element
    System.out.println();                 // end of the column
  }
}
```

Also in this case, we use two nested `for` loops: the outermost loop iterates over the columns (using control variable `j`), while the innermost loop iterates over the elements of column `i` (using control variable `j`). At the end of each column, a newline character is printed out.

Example of usage:

```
public static void main(String[] args) {
  short[][] A = {          // creates a matrix A of dimension 2x3
    { 1, 2, 2 },           // row 0 of A (array of 3 short)
    { 7, 5, 9 }            // row 1 of A (array of 3 short)
  };
  printMatrixRows(A);      // prints out the matrix row by row
  System.out.println();
  printMatrixColumns(A);   // prints out the matrix column by column
}
```

The program prints out the following:

```
1 2 2
7 5 9

1 7
2 5
2 9
```

7.20 Example: sum of two matrices

We develop a static method, `matrixSum()`, that takes as parameters two matrices `A` and `B` of doubles, and returns a new matrix obtained by summing corresponding elements of `A` and `B`. We assume that `A` and `B` have the same dimensions (i.e., the same number of rows and the same number of columns).

```
public static double[][] matrixSum(double[][] A, double[][] B) {
  double[][] C = new double[A.length][A[0].length];
  for (int i = 0; i < A.length; i++)
    for (int j = 0; j < A[0].length; j++)
      C[i][j] = A[i][j] + B[i][j];
  return C;
}
```

Note that, in order to access all elements of the matrices `A` and `B`, we use two nested `for` loops: the outermost one iterates over the rows (`i`), while the innermost one iterates over the elements of each row (`j`).

Example of usage:

```
public static void main(String[] args) {
  double[][] A = {            // creates matrix A of dimension 2x3
    { 1.2, 2.3, 2.3 },        // row 0 of A (array of 3 double)
    { 7.4, 5.1, 9.8 }         // row 1 of A (array of 3 double)
  };
  double[][] B = {            // creates matrix B of dimension 2x3
    { 5.0, 4.0, 1.3 },        // row 0 of B (array of 3 double)
    { 1.2, 0.3, 3.2 }         // row 1 of B (array of 3 double)
  };
```

```
    double[][] C = matrixSum(A,B);   // calculates the sum of A and B
    System.out.println(C[1][0]);       // prints out 8.6
  }
```

7.21 Example: product of two matrices

We develop a static method, `matrixProduct()`, that takes as parameters two matrices `A` and `B` of doubles and returns a new matrix obtained as the product of `A` and `B`. Assume that the matrices are square, i.e., they have the same numbers of rows and columns, and that `A` and `B` have the same dimensions.

Recall that, if `C` is the product of `A` and `B`, then the generic element `C[i][j]` is the scalar product row `i` of `A` with column `j` of `B`, i.e., `C[i][j] =` $\sum_k$`(A[i][k]*B[k][j])`.

```
  public static double[][] matrixProduct(double[][] A, double[][] B) {
    double[][] C = new double [A.length][A[0].length];
    for (int i = 0; i < A.length; i++)
      for (int j = 0; j < A[0].length; j++) {
        C[i][j] = 0;
        for (int k = 0; k < A[0].length; k++)
          C[i][j] += A[i][k] * B[k][j];
      }
    return C;
  }
```

Note that, in order to calculate the product of the matrices `A` and `B`, we use three nested `for` loops: the outermost loop iterates over the rows (`i`) of `C`, the intermediate loop iterates over the columns (`j`) of `C`, while the innermost loop calculates the scalar product of row `i` of `A` with column `j` of `B`, and assigns the calculated value to `C[i][j]`.

Example of usage:

```
  public static void main(String[] args) {
    double[][] A = {            // creates matrix A of dimension 3x3
      { 1.0, 2.0, 2.0 },
      { 7.0, 5.0, 9.0 },
      { 3.0, 0.0, 6.0 }
    };
    double[][] B = {            // creates matrix B of dimension 3x3
      { 5.0, 4.0, 1.0 },
      { 1.0, 0.0, 3.0 },
      { 7.0, 5.0, 2.0 }
    };

    double[][] C = matrixProduct(A,B); // calculates the product A*B
    System.out.println(C[1][0]);  // prints 103.0 (obtained as
                                  // 7.0*5.0 + 5.0*1.0 + 9.0*7.0)
  }
```

7.22 Example: sums of the rows of a matrix

We write a static method, `matrixSumRows()`, that takes as parameter a matrix and returns an array with one element for each row of the matrix; the element of index `i` of the array must be equal to the sum of the elements of row `i` of the matrix.

```
  public static int[] matrixSumRows(int[][] M) {
    int[] C = new int[M.length];            // creates the array
    for (int i = 0; i < M.length; i++) {    // iterates over the rows of M
      C[i] = 0;                             // initializes the accumulator
      for (int j = 0; j < M[i].length; j++) // iterates over the elements of row i
        C[i] += M[i][j];                    // accumulates the elements of row i
    }
    return C;
  }
```

Example of usage:

```
public static void main(String[] args) {
  int[][] A = {     // creates matrix A of dimension 3x3
    { 1, 2, 2 },
    { 7, 5, 9 },
    { 3, 0, 6 }
  }

  int[] B = matrixSumRows(A);
  for (int i = 0; i < B.length; i++)
    System.out.print(B[i] + " ");     // prints out the array B
  System.out.println();
}
```

The program prints out the following:

```
5 21 9
```

Exercises

Exercise 7.1. Write a method `static double scalarProduct(double[] A, double[] B)` that calculates the scalar product of two arrays `A` and `B`, assuming they have the same length. We recall that the *scalar product* of `A` and `B` is obtained as the sum of the products `A[i]*B[i]`, for all `i`, with $0 \leq i <$ `A.length`.

Exercise 7.2. Write a method `static int[] intersection(int[] A, int[] B)` that returns a new array containing the intersection of two arrays `A` and `B`, i.e., exactly those elements that are present both in `A` and in `B` (independently of their position). We may assume that `A` and `B` do not contain duplicates, i.e., elements that appear more than once in the array.

Exercise 7.3. Write a method `static double[][] transposeMatrix(double[][] M)` that returns a new matrix that is the transpose of the matrix `M`. We recall that the *transpose* of a matrix `M` is obtained by exchanging the rows with the columns, which corresponds to exchange `M[i][j]` with `M[j][i]`, for each pair of valid indexes `i` and `j`.

Exercise 7.4. Write a method, `static int[] matrixSumColumns(int[][])`, that takes as parameter a matrix and returns an array with one element for each column of the matrix; the element of index `i` of the array must be equal to the sum of the elements of column `i` of the matrix.

Exercise 7.5. Write a predicate `static boolean equalArrays(int[] A, int[] B)` that returns `true` if the two arrays `A` and `B` are equal (i.e., `A[i]` is equal to `B[i]`, for each `i`), and `false` otherwise.

Exercise 7.6. A *duplicate* in an array is a value that appears more than once as element of the array. Write a method `static int numberOfDuplicates(int[] A)` that returns the number of distinct duplicates in the array `A`. Write also a method `static int numberOfDistinctValues(int[] A)` that returns the number of distinct values in the array `A`.

Exercise 7.7. Write a method `static int[] removeDuplicates(int[] A)` that returns a new array obtained from `A` by removing all duplicates. The duplicates should be removed by keeping only the first occurrence of each distinct element, and shifting remaining elements upwards when a duplicate is removed.

Exercise 7.8. What does the following method calculate?

```
public static int mystery(int[] A) {
  int x = 0;
  for (int i = 0; i < A.length; i++)
    if (A[i] % 2 == 1) x++;
  return x;
}
```

Exercise 7.9. A matrix `M` is said to be *symmetric* if it is square (i.e., has the same number of rows and columns) and `M[i][j]` is equal to `M[j][i]`, for each pair of valid indexes `i` and `j`. Write a predicate `static boolean symmetric(int[][] M)` that returns `true` if the matrix `M` is symmetric, and `false` otherwise.

Exercise 7.10. A matrix `M` is said to be *lower triangular* if all elements `M[i][j]` with `i<j` (i.e., that are "above" the main diagonal) are equal to 0. Write a predicate `static boolean lowerTriangular(int[][] M)` that returns `true` if the matrix `M` is lower triangular, and `false` otherwise.

Exercise 7.11. A matrix `M` is said to be *diagonal* if all elements `M[i][j]` with `i` different from `j` (i.e., that are not on the main diagonal) are equal to 0. Write a predicate `static boolean diagonal(int[][] M)` that returns `true` if the matrix `M` is diagonal, and `false` otherwise.

Exercise 7.12. Realize a Java class `Apartment`, whose objects maintain the following information: an integer that indicates the size in square meters of the apartment, the address, and a list of maximal 10 names of persons that live in the apartment. Each person living in the apartment has an associated number between 0 and the number of persons currently living in the apartment minus 1. The class should export the following functionalities:

- a constructor that takes a size and an address and creates an apartment that is initially empty;
- a method that returns the size of the apartment;
- a method that returns the address of the apartment;
- a method that returns the number of persons currently living in the apartment;
- a method that takes as parameter the name of a person, and adds the person to those living in the apartment; the person gets assigned the next available number, if there is still room; if the apartment is full (i.e., 10 persons already live there), the method does nothing;
- a method that takes as parameter a nonnegative integer number and returns the name of the person associated to that number and living in the apartment; if the number is greater than or equal to the number of persons in the apartment, the method returns null;
- a method that takes as parameter a nonnegative integer number and removes the person associated to that number from the apartment; the number associated to all following persons should be decreased by one; if the number is greater than or equal to the number of persons in the apartment, the method does nothing.
- a method `toString()` that does overriding of `toString()` inherited from object, and returns a string with all the information about the apartment.