Introductory Programming
Arrays, sections 6.0-6.3

Anne Haxthausen[a], **IMM, DTU**

a. Parts of this material are inspired by/originate from a course at ITU developed by Niels Hallenberg and Peter Sestoft on the basis of a course at KVL developed by Morten Larsen and Peter Sestoft.

What is an array?

Basic concept:

An *array* is an indexed (Danish: nummereret) collection of variables, called *elements*.

The elements are indexed with integers 0, 1, 2, ...

All elements have the same type.

Example: graphical illustration of an array having 12 elements of integer type:

0	1	2	3	4	5	6	7	8	9	10	11
5	4	1	7	19	2	5	1	0	13	0	0

In Java: an array is a special kind of object, for which elements are similar to field variables.

The remaining part of this lecture concerns arrays in Java.

Array types

When the elements of an array have type `T`, then the array itself has type `T[]`.

In the example above the type was **int**`[]`.

Summary: kinds of types

- primitive types:
  - integer types: **int**, ...
  - floating point number types: **double**, ...
  - the character type: **char**
  - the Boolean type: **boolean**
- reference types:
  - class types: `String`, `Time`, ...
  - interface types: `TimeInterface`, ...
  - array types **int**`[]`, `Time[]`, ...

How to declare and create an array?

Declararing an array variable (but not creating the array itself):

```
int[] days;
```

Creating an array having 12 integer elements (all initialized to 0)
and assigning to the array variable a reference to that array:

```
days = new int[12];
```

days → | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Simultaneous declaration, creation and assignment:

```
int[] days = new int[12];
```

The array itself is an object, while the array variable `days` contains a reference to the array.

How to initialize an array?

Declaration, creation and initialization in one step:

```
int[] days = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

0	1	2	3	4	5	6	7	8	9	10	11
31	28	31	30	31	30	31	31	30	31	30	31

Operations on arrays

Length of an array

The length (i.e. the number of elements) of the `days` array is obtained by:

```
days.length
```

In the example `days.length` evaluates to 12.

Indexing (looking up elements)

Find the contents of element number 3:

```
days[3]
```

The expression inside `[...]` is called an *array index*.

Legal index values are `0, 1, 2, ..., days.length-1`.

If the index has an illegal value, the program is interrupted with the message:

```
java.lang.ArrayIndexOutOfBoundsException: ...
```


Element assignment

Assign the value 29 to element number 1:

```
days[1] = 29;
```


Example1: using arrays

A method that takes the number of a month (1–12) as argument and returns the number of days in the month (in a non-leap year):

```
static int monthlen(int mth) {
  int[] days = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
  return days[mth-1];
}
```

Example2 (exercise 17): using arrays

```
public class Date {
  final static String[] danishMonths =
    {"januar", "februar", "marts", "april", "maj", "juni", "juli",
      "august", "september", "oktober", "november", "december"};

  private int year, month, day; //fields
  ...
  public String danishText() {
    return day +". "+ danishMonths[month-1] +" "+ year;
} }

public class TestDate {
  public static void main(String[] args) {
    Date today = new Date(2001, 10, 23);
    System.out.println("Danish text: " + today.danishText());
}
```

`TestDate` writes: `Danish text: 23. oktober 2001`

Array aliasing

Two array variables can refer to the same array object.

Example

```
int[] days1 = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
int[] days2 = days1
```

Implications of array aliasing:

If the contents of an array is changed, then it implies a change for *all* the array variables that has a reference to the array.

Example

```
days1[1] = 29;
```

now `days1[1] == 29` and `days2[1] == 29`

How to compare the contents of arrays?

Example

Assume given:

```
int[] days1 = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
int[] days2 = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

How can we investigate whether the arrays that `days1` and `days2` refer to, have the same contents?

How to compare the contents of arrays, continued?

Wrong:

```
days1 == days2
```

does *not* work – it decides whether `days1` and `days2` refer to the same array, i.e. are aliases. (The returned value is **false**.)

Correct: Use `java.util.Arrays.equals(days1, days2)` or make your own method:

```
static boolean equals(int[] array1, int[] array2) {
  boolean same;
  same = (array1.length == array2.length);
  for (int i=0; same && i < array1.length; i = i+1)
     same = (array1[i] == array2[i]);
  return same;
}
```

`equals(days1, days2)` now returns **true**, as the contents of those arrays that `days1` and `days2` refer to is the same.

Arrays as arguments of methods.
Example: how to copy the contents of one array into another array

Exercise: write a method, `copy`, that copies the elements from one integer array to another integer array of the same length.

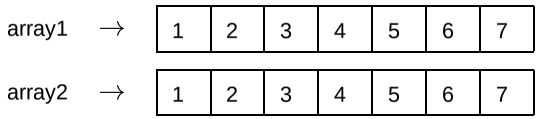
Example: Assume given

```
int[] array1 = { 1,  2,  3,  4,  5,  6,  7};
int[] array2 = { 8,  9, 10, 11, 12, 13, 14};
```

After execution of

```
copy(array1, array2);
```

we have:



Arrays as arguments of methods.
Example cont.: how to copy the contents of one array into another array

Which of the following methods are right?

```
static void copy(int[] from, int[] to) { to = from; }

static void copy(int[] from, int[] to) {
  for (int i = 0; i < from.length; i = i+1)
    to[i] = from[i];
}
```

What is the other method doing?

What happens with variables
that are used as actual parameters of methods?

Java uses *call-by-value* when a value is passed as an actual parameter to a method — corresponds to copying the value of the *actual* parameter to the *formal* parameter in the method. After a call of the method, the actual parameter (here: variable) has the same value as before the call of the method.

Implication for arrays as arguments of methods:

- when variables of array types are arguments of a method, the *reference* to the array is copied, not the array itself
- if the method body changes the formal parameter reference (let it refer/point to a new array), it will *not* change the argument
- if the method body changes the state (the contents) of the array that the formal parameter is refering to, the same happens to the argument

The parameter `args` in `main` methods

`args` in

```
public static void main (String[] args) { ... }
```

is an array of string objects that should be provided as command-line arguments when the program is executed. Inside the `main` method you can refer to the strings as `args[0]`, `args[1]`, etcetera.

Example:

```
class Pension {
  public static void main (String[] args) {
    double pension =  Integer.parseInt(args[0]) * 0.1;
    System.out.println("Your pension is: " + (int) pension);
  }
}
```

Example of program execution:

```
$ java Pension 300000
Your pension is: 30000
```

Arrays having objects as elements

An array can contain all kinds of data; also references to objects.

Example:

Declaration of a variable for arrays having references to Time objects:

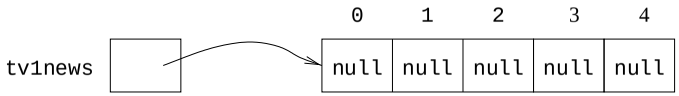
```
Time[] tvnews;
```


Arrays having objects as elements

Declaration and initialization of array having space for 5 references to Time objects:

```
Time[] tvnews = new Time[5];
```

No `Time` objects are created by this. All the elements in the array `tvnews` are initialized to **null**.



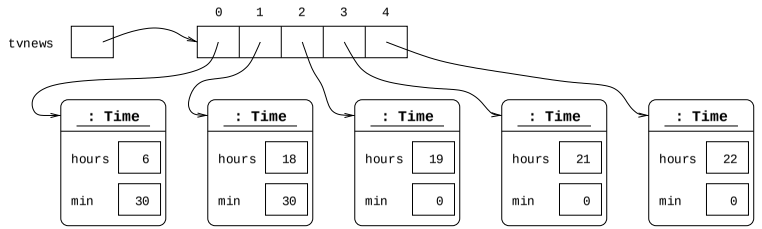
Time objects have to be created explicitly, e.g.:

```
tvnews[0] = new Time(6, 30);
tvnews[1] = new Time(18, 30);
tvnews[2] = new Time(19, 0);
tvnews[3] = new Time(21, 0);
tvnews[4] = new Time(22, 0);
```


Arrays having objects as elements

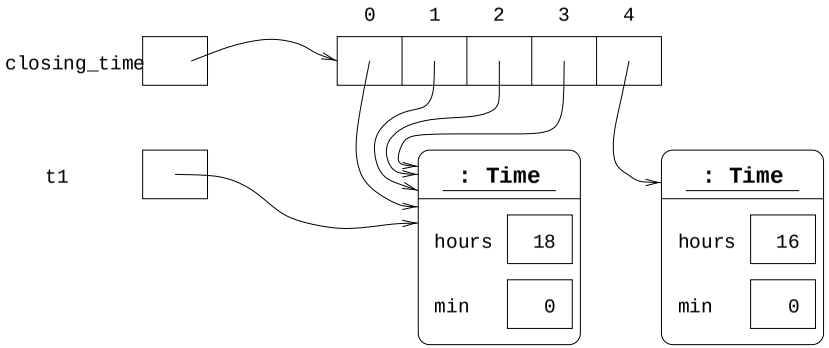
Declaration and initialization with references to Time objects (that are created at the same time):

```
Time[] tvnews = { new Time(6, 30), new Time(18, 30),
                  new Time(19, 0), new Time(21, 0),
                  new Time(22, 0) };
```



Several elements of an array can refer to the same object

```
Time t1 = new Time(18, 0);
Time[] closing_time = { t1, t1, t1, t1, new Time(16, 0) };
```



Multidimensional arrays

The elements of an array can also be arrays. That gives a multidimensional array.

Example: a cinema can be modelled as an array of rows; a row as an array of seats. Each seat is registered as occupied (**true**) or vacant (**false**).

Ergo: a cinema is a two-dimensional array of **boolean**s.

Declaration of a two-dimensional array of **boolean**s:

```
boolean[][] bio;
```

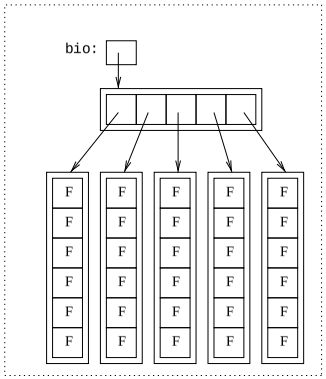
Creation of a "5×6" cinema (all elements are automatically initialized to **false**):

```
bio = new boolean[5][6];
```

Number of rows: `bio.length`

Reservation of row 2 seat 5: `bio[2][5] = true;`

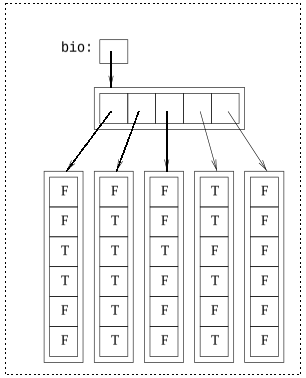
Is row 3 seat 1 vacant? `bio[3][1]`



Multidimensional arrays

Declaration, creation and initialization in one step:

```
boolean[][] bio =
{{false,false,true, true, false,false},
 {false,true, true, true, true, true },
 {false,false,true, false,false,false},
 {true, true, false,true, false,true },
 {false,false,false,false,false,false}};
```



More multidimensional arrays

The rows in a two-dimensional array *need not* have the same number of elements:

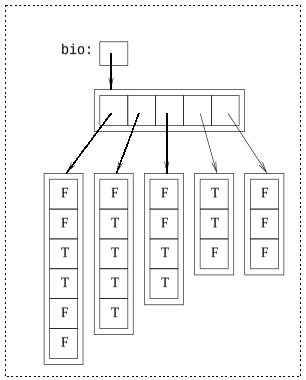
```
boolean[][] bio =
  {{false,false,true,true,false,false},
   {false,true, true,true,true},
   {false,false,true,true},
   {true, true, false},
   {false,false,false}};
```

Row 0 has 6 seats.

Row 1 has 5 seats.

Row 2 has 4 seats.

Rows 3 and 4 have 3 seats.



Command line and multidimensional arrays (1)

Exercise: write a program `CinemaDecl` that reads a sequence of integers.

- Each integer states how many seats there are on a given row in a cinema.
- The number of integers states the number of rows in the cinema.
- The program should print the layout of the cinema.

Example of execution:

```
$ java CinemaDecl 6 5 4 3 3
0 0 0 0 0 0
0 0 0 0 0
0 0 0 0
0 0 0
0 0 0
```


Command line and multidimensional arrays (2)

```
public static void main (String[] args) {
  boolean[][] bio = new boolean[args.length][];

  for (int i=0; i<args.length; i=i+1)
  {
    int seatsOnRow = Integer.parseInt(args[i]);
    bio[i] = new boolean[seatsOnRow];
  }

  for (int row=0; row<bio.length; row=row+1)
  {
    for (int seat=0; seat<bio[row].length; seat=seat+1)
      System.out.print("O ");
    System.out.println();
  }
}
```


Arrays of variable length

In the CD example in section 6.1 of the book, it is shown how you can increase the length of an array dynamically. (See the `increaseSize` method in listing 6.8.)

Using arrays for sorting

A well-known subject in computer science is *sorting* items (e.g. integers) in an array so that smaller items come before larger ones.

Example: Sorting the array

0	1	2	3	4
3	9	6	1	2

gives

0	1	2	3	4
1	2	3	6	9

Formal definition: a sorting algorithm is an algorithm that makes a permutation of the items in an array `a`, such that `a[i]` $\preceq$ `a[i+1]` for `0 <= i <= a.length-2`, where $\preceq$ is a given total ordering relation (e.g. `<=` for integers).

Many algorithms for sorting exist. The book shows two of them:
selection sort and *insertion sort*.

Idea of selection sort

Start from one end. Find the smallest item of those items that are not yet placed where they should be. Put it in its final position. Repeat this until all items are in their final position.

Start with 3. 1 is smallest. Interchange 1 and 3.

3	9	6	1	2

Start with 9. 2 is smallest. Interchange 2 and 9.

1	9	6	3	2

Start with 6. 3 is smallest. Interchange 3 and 6.

1	2	6	3	9

Start with 6. 6 is smallest. Interchange 6 and 6.

1	2	3	6	9

1	2	3	6	9

The selection sort algorithm

```
public static void selectionSort(int[] numbers) {
   int min, temp;

   for(int i=0; i<numbers.length-1 ; i=i+1)
   {  min = i;
      //find index min of smallest item in A[i .. ]
      for(int scan=i+1; scan<numbers.length; scan=scan+1)
         if (numbers[scan] <numbers[min])
            min=scan;
      //swap the items at index i and min
      temp=numbers[min];
      numbers[min]=numbers[i];
      numbers[i] = temp;
   }
}
```