Data structures and programming structures

Primitive data types
Reference data types
Operators and expressions
Statements
Methods and passing of arguments
Arrays
Strings
Main method

Primitive data types

- we directly manipulate the values of these objects
  we cannot have access to their memory address

- the memory size of these types is imposed by Java → platform independent

boolean : **true** or **false** value on 1 bit

char : Unicode character coded on 16 bits → 65536 characters at most
= expands ASCII code and Latin-1 code

uxxxx with xxxx from 0000 to FFFF

from 0000 to 00FF : 256 Latin-1 characters (from 00 to FF)
from 00FF to FFFF : all other alphabet characters and ideograms…

examples:

'a' **'A'** **'\n'**

'\' == '\u005c' **'"' == '\u0022'**

byte , short , int , long : signed integers on 8, 16, 32, 64 bits

note: **byte** ranges from –128 to +127

float , double : floating point numbers on 32, 64 bits

we can cast the type of a variable into another type (like in C)
BUT it is forbidden to cast a char into a byte or short and vice versa

example:

x is of type **double**

→ **(float) x** is of type **float**

Reference data types

- we manipulate these objects by implicitly handling their memory address, or reference

  → a variable which is declared as being a certain object actually contains the address of this object
  = *abuse of notation*, formally wrong but very useful !

  BUT the value of this address is never directly accessible to us and we cannot know the size of the object
  → unlike C, no pointer arithmetic, no sizeof → no more memory errors!

- data handled by reference: object (instance of a class) , array , string

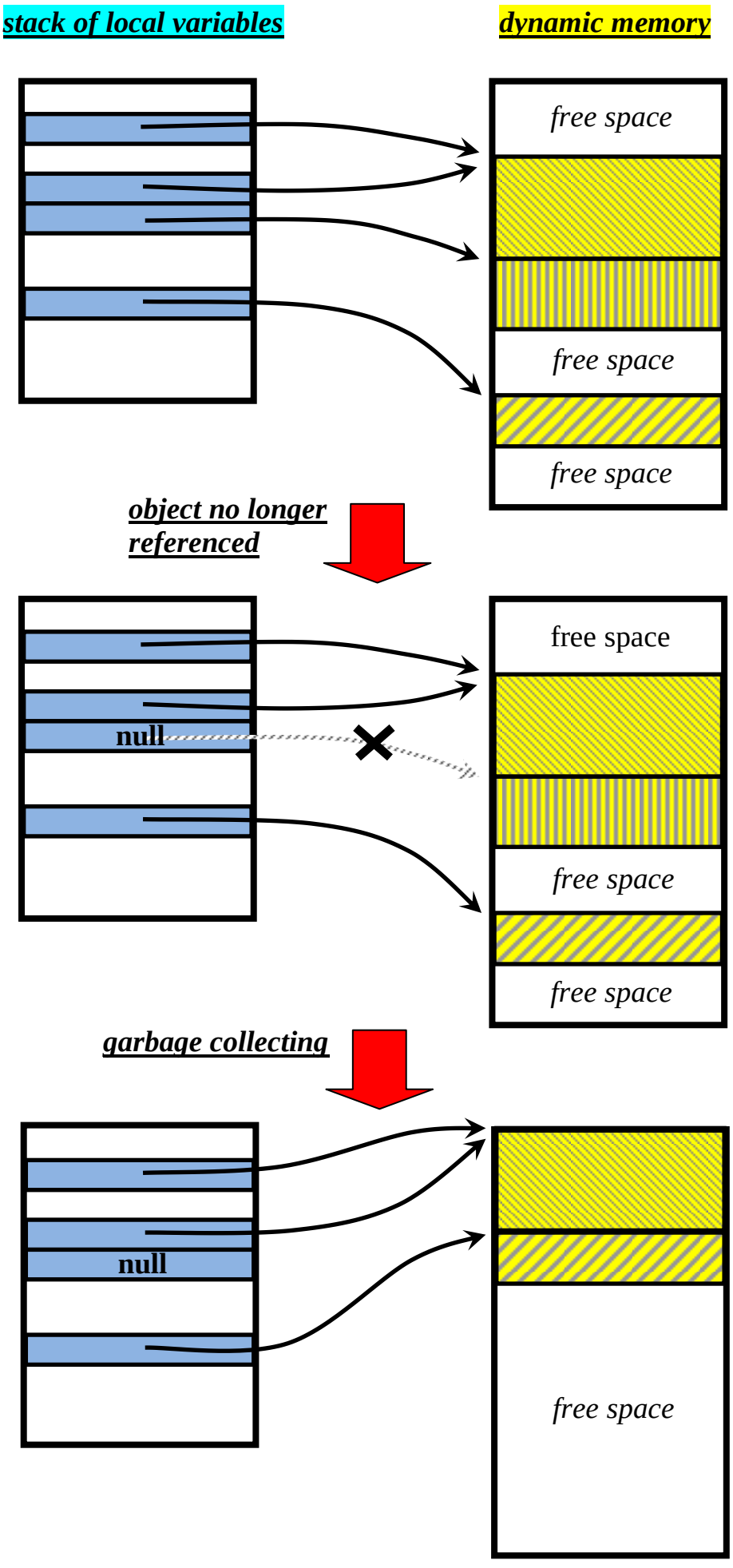
  strings are instances of class **String**
  → treated in a dedicated way

- if a variable which is declared as an object (or array, string) does not actually refer to any object (or array, string), then its value is **null**

- ALL data handled by reference -even arrays- are allocated in the dynamic memory and not in the stack of function calls as local variables are.

garbage collection:

the JVM automatically detects when an object is no longer referenced by any variable
→ the JVM deallocates the place occupied in memory by this object
→ the programmer is no longer in charge of deallocation (no more **free**)


stack of local variables
dynamic memory
free space
free space
free space
object no longer referenced
null
free space
free space
free space
garbage collecting
null
free space

Operators and expressions

arithmetic expression: formula whose result is a number

→ operators:

+ , - , * , / , % (remainder of integer division)

boolean expression: formula whose result is a boolean (true or false)
= must be used for all tests of conditional statements!

→ operators:

== , != on arithmetic expressions or references of objects

< , <= , > , >= on arithmetic expressions

! , && , || on boolean expressions

conditional expression:

boolean ? value$_1$: value$_2$

= if **boolean** is true, then yield **value$_1$** else yield **value$_2$**

Statements

comment:

// rest of line

or **/* . . . . . */** (cannot be nested)

declaration :

type variable;

or **type variable$_1$, … variable$_n$;**

at any place in the code, unlike in C (where declarations must be the first statements of a block)

assignment:

variable = expression;

or **variable operator expression;**
with **operator** being **+=** , **−=** , ***=** , **/=** , **%=**

or **variable++;**

or **variable−−;**

declaration - assignment:

type variable = expression;

or **type variable$_1$, variable$_2$ = expression , variable$_3$;**

conditional constructs:

```
if (boolean) statement;
```

```
if (boolean)  { block of statements }
```

```
if (boolean)  . . . . .  else  . . . . .
```

int x; .... **if (x)** . . . . .
and **if (1)** . . . . . are not allowed !

switch according to the value
of an expression of type **byte**, **char**, **short**, **int** or **long** :

```
switch (expression)
{
   case value1 :  . . . . .  break;
   . . . . . .
   case valuen :  . . . . .  break;
   default:       . . . . .  break;
}
```

loop constructs:

while (boolean) statement;

while (boolean) { block of statements }

do . . . . . while (boolean);

example:

```
double x = 0.0;
while (x < 4.0) { System.out.println("x=" + x); x += 1.0; }
```

→ x=0.0
x=1.0
x=2.0
x=3.0

int x; . . . . . while (x) . . . . .
and **while (1) . . . . .** are not allowed !

```
for (i = 0 ; i < n ; i++)  . . . . .
```

initialize two loop variables:

```
for (i = 0 , j = 0 ; i < n ; i++ , j++)  . . . . .
```

declare and initialize the loop variable:

```
for (int i = 0 ; i < n ; i++)  . . . . .
```

declare and initialize two loop variables:

```
for (int i = 0 , j = 0 ; i < n ; i++ , j++)  . . . . .
```

note: we cannot mix multiple initializations with multiple declarations(-initializations)

get out of the nearest enclosing loop construct: **break;**

continue to the next iteration: **continue;**

example:

```
for (int i = 0 , j = 6 ; i < j ; i++ , j--)
 {
    System.out.println(“ i=” + i + “   j=” + j);
    if (i + 2 == j) break;
 }
```

→ i=0 j=6
i=1 j=5
i=2 j=4

Methods and passing of arguments

- Java method is quite similar to C function
  but a Java method is ALWAYS defined as a member of a certain class
  and we have either *instance* methods or *static* methods...

- the order in which the methods are written in the file
  has strictly no importance (unlike in C)

method returning nothing:

```
modifier(s)  void  method_name ( type1 arg1 , . . . . . , typen argn )
{
   . . . . .
}
```

method returning value:

```
modifier(s)  return_type  method_name ( type1 arg1 , . . . . . , typen argn)
{
   . . . . .
   return value;
}
```

modifier: keyword providing additional information about the method

modifier keyword	meaning
none	accessible only within its package
public	accessible anywhere (if the class is also **public**)
private	accessible only within its class (or a class within its class) (and not within a subclass)
protected	accessible only within its package and within subclasses of its class (that may be outside its package)
static	static or class method
final	may not be overriden (redefined)
abstract	no body provided for the method (must be overriden)

passing of arguments:

- argument is of a primitive data type (**int**, **double**, ...) → we directly pass its value
- argument is of a reference data type (object, array, string) → we implicitly pass the value of its reference

we DO NOT have to specify that what we pass is the address (like in C)

example:

***Java*:**	***C*:**
Object a;	**struct Object a;**
. . . . .	. . . . .
method(a);	**func(&a);**
. . . . .	. . . . .
void method(Object a)	**void func(struct Object *pa)**
{ . . . . . **}**	**{** . . . . . **}**

→ we DO NOT recopy the whole object, array or string!

example:

```
Object a;
method(a);
System.out.println(a.x);      →   prints  25

. . . . . .

void method(Object a)
{  a.x = 25;  }
```

→ can be used to return a result from a method

Arrays

- an array is handled by reference (like in C)
- it is always allocated in dynamic memory
- it is automatically deallocated when no longer referred to

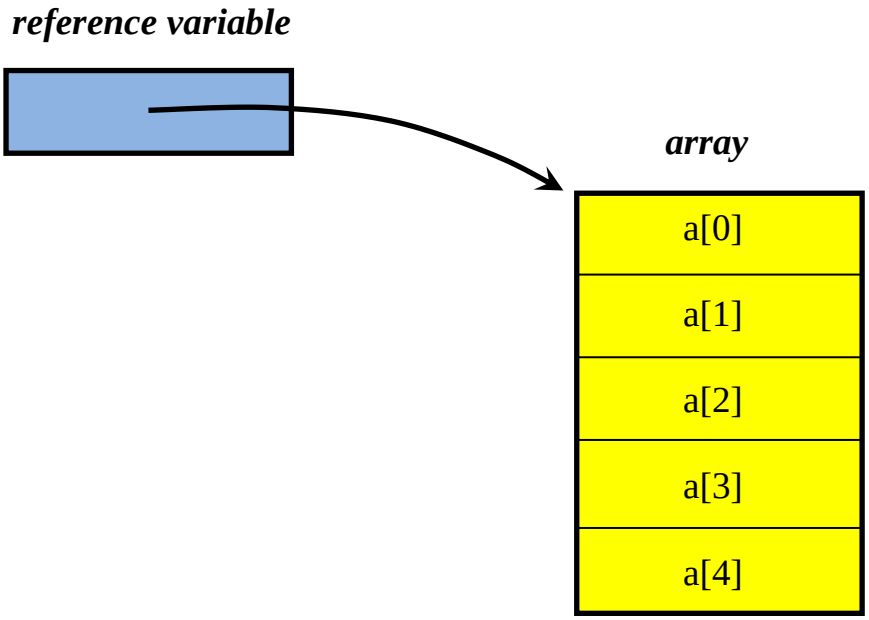
number of elements of the array: **array_name.length** = can only be read

index starts from 0: $0 \leq \text{index} < \textbf{array\_name.length}$

the JVM checks at runtime that the index is within its bounds
→ slow but no memory error possible!

element of index i: **array_name[i]**

3 steps:
- declaration of the reference variable
- creation of the array = allocation of a place in memory
- initialization of the elements of the array



- declaration:

element_type array_name[];

or **element_type[] array_name;** (preferred)

- creation:

array_name = new element_type[n]; (after declaration)

- declaration and creation:

element_type[] array_name = new element_type[n];

- declaration, creation and initialisation:

element_type[] array_name = { element$_0$, ... , element$_{n-1}$ };

(arbitrary expressions for the values of the elements)

- creation and initialization:

→ *anonymous array*

= not necessary to declare a variable containing the array reference!

new element_type[] { element$_0$, ... , element$_{n-1}$ }

examples:

```
short[] a;

a = new short[5];

int[] b = new int[a.length];

int[] c = { 100 , 101 , 102 , 103 , 104 };

for (int i=0 ; i<c.length ; i++)
{
   b[i] = c[i];
   a[i] = (short )b[i];
}
```

```
System.out.println("element 1 = " + (new int[] { 10 , 20 , 30 , 40 })[1] );
```

→ element 1 = 20

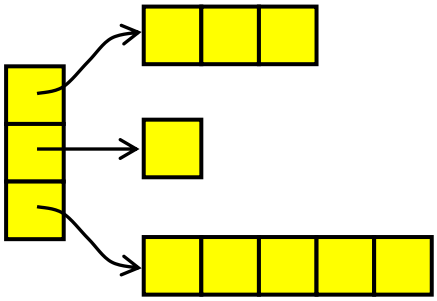
```
method(new double[] { 11.1 , 22.2 , 33,3 });

. . . . . .

void method(double[] a)
{ . . . . . }
```

multi-dimensional arrays:

- 1D array (leftmost dimension) of 1D arrays (of 1D arrays…)
- the size may vary for each array element of the primary array:



element of indices i,j: **array_name[i][j]**

element_type[][] array_name = new element_type[n][m];

initially, we can allocate either all the dimensions
or, partially, only some left side dimensions (at least the leftmost one!)

→ the remaining dimensions shall be allocated later:

element_type[][] array_name = new element_type[n][];

then, for each value of the first index i:

array_name[0] = new element_type[m_0];
. . . . . .
array_name[n-1] = new element_type[m_{n-1}];

new int[10][][5][] is not valid!

examples:

```
float[][][] a = new float[2][][];

a[0] = new float[10][100];

a[1] = new float[3][];

a[1][0] = new float[10000];

a[1][1] = new float[1000];

a[1][2] = new float[2000];
```

```
int[][] triangle = new int[4][];
for (int i=0 ; i<triangle.length ; i++)
{
   triangle[i] = new int[i+1];
   for (int j=0 ; j<triangle[i].length ; j++)
      triangle[i][j] = i+j;
}
```

→ { { 0 }
, { 1 , 2 }
, { 2 , 3 , 4 }
, { 3 , 4 , 5 , 6 } }

```
int[][] array = {  { a , b , c }
                 , { d }
                 , { e , f , g , h , k , l } };
```

→ **array.length** == 3
array[0].length == 3
array[1].length == 1
array[2].length == 6

Strings

- a string is an object instance of the **String** class
- general handling of String objects follows the object oriented approach

declaration of a String variable: **String s;**

concatenation operator: **s1 + s2**

a String is immutable:

once created in dynamic memory, a **String** object cannot be modified

→ we must create a new **String** object for each new modification

printing:

System.out.print(string);

or **System.out.println(string);** (newline added at the end)

in general, the string is of the form
string + number + string + object + number ……

→ automatic conversion of the objects and primitive data types into their string representations

example:

String s = "abcd"; String s1 = s + "ef"; String s2 = "AA" + s1;

int i=123; double x=123.456;

System.out.println("s2 = " + s2 + " ; i = " + i + " ; x = " + x);

→ s2 = AAabcdef ; i = 123 ; x = 123.456

Main method

execution of a program with **n** arguments:

java program_name arg_0 … arg_{n-1}

→ call to the main method with an array of strings as argument:

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

number of arguments: **arg.length**

argument n. **i** coded as a string: **arg[i]**

example:

```
java toto ab cd ef


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

→ argument 0 = ab
argument 1 = cd
argument 2 = ef