Buoys for Variables in Java

Table of Contents

by Ulrich Grude
Beuth University of Applied Sciences

Abstract: Buoys are a graphical representation of variables. They were invented with the programming language Algol68, but can be used to represent the variables of any programming language. With this notation some otherwise hard problems will become easy to teach and to understand. For Java this includes the following: Of how many parts does a variable consist? What is the difference between a variable of a primitive type and one of a reference type? Since the value of an `int`-variable is an `int`-number, why is it that the value of a `String`-variable is *not* a `String`-object? What is the difference between the operation `==` and the method `equals`? What is the difference between an *empty array* and a *null reference*? What is the difference between a *nested array* and a *multidimensional array*? etc.

Buoys lend themselves to illustrate notions like *variable*, *value*, *reference*, *equality*, *identity* etc. and to test a deeper understanding of those notions with problems of the form:
"Draw the buoy(s) of the following variable(s): ...".

The term *exer* in this paper is meant to encompass everything which is used to execute a program (e.g. compilers, interpreters, operating systems, various kinds of hardware etc.). The *exer* also may be a human being (e.g. the reader) who executes a program with a pencil and paper.

1. How to represent variables as buoys

The concept of a variable containing a value, which may be replaced by another value any number of times is arguably the most important and fundamental concept of most programming languages. Many phenomena pertaining to the realm of programming can only be comprehended with a precise mental image of how such a modifiable variable looks like. So-called **buoys** provide a graphical representation of variables, which supports and facilitates such a precise image. Variables of all mainstream programming languages can be represented by buoys. Buoys have been invented together with the programming language Algol 68 (towards the end of the nineteen-sixties) and have been slightly improved by students of the *Beuth University of Applied Sciences*. This paper is specifically about the programming language Java and its variables (and at the same time an introduction to buoys in general).

1.1. Types, variables and values in Java

In Java it is useful to distinguish

Primitive types (e.g. `int`, `double`, `boolean`, ...) and
Reference types (e.g. `String`, `List<String>`, `String[]`, `String[][][]`, ...).

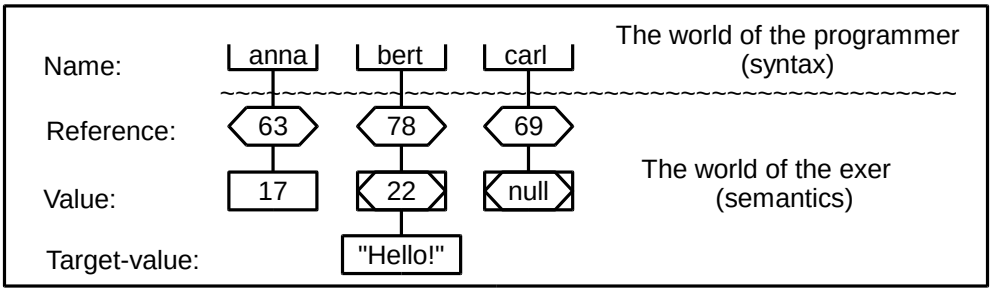
Primitive variables (i.e. variables of a primitive type) and
Reference variables (i.e. variables of a reference type),

Primitive values (i.e. values which may be assigned to a primitive variable) and
Reference values (i.e. values which may be assigned to a reference variable).

Example-01: One *primitive variable* and two *reference variables* represented as buoys:

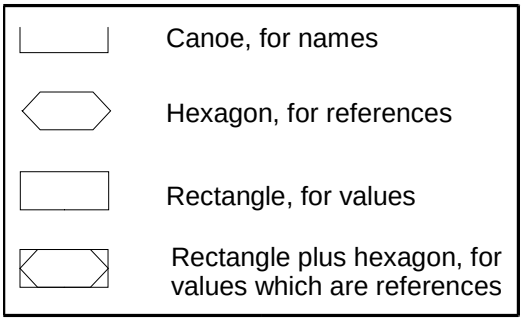
```
1 int           anna = 17;                          // a primitive variable
2 StringBuilder bert = new StringBuilder("Hello!"); // a reference variable
3 StringBuilder carl = null;                        // a reference variable
```

As buoys these variables may look as follows:



Every variable consists of at least two parts: A **reference** and a **value**. Every variable may (or may not) have a **name**. In addition, reference variables may (or may not) have a **target-value**. Thus *primitive variables* may consist of 2 or 3 parts and *reference variables* may consist of 2, 3 or 4 parts.

Only the **name** of a variable (if present) belongs to the *world of the programmer*. The other parts belong to *the world of the exer*, because only he can generate and manipulate them, and the programmer will never see them directly (at most he may see certain shadows of values or target-values, e.g. on a screen).



In a buoy 4 geometric shapes are used to represent the (at most) 4 parts of a variable. **Names** are placed into so-called *canoes*. **Values** are enclosed in *rectangles*. **References** are enclosed in *hexagons*. The *value of a reference variable* is at the same time a **reference** and a **value**. Therefore it is represented by a *hexagon* within a *rectangle*.

The special reference-value `null` does *not refer* to a target value. Every other reference-value *does* refer to a target value. In Java, every target value is an object.

References are chosen (not by the programmer but) by the *exer*. He may chose them *any way he likes*, but has to guarantees two things:

1. References of variables are unique (i.e. different variables have different references).
2. The special value `null` is never used as the *reference* of a variable
(it is used only as the *value* of reference variables).

In Java, there are *no* operations which operate on the *references of variables*. And there are only *three* operations, which operate on *reference values*: `==` and `!=` (the equality and the inequality operation, respectively) and `=` (the assignment operation). The programmer can not apply any other operation to reference values (e.g. he can **not** do a computation with them, convert them to an `int`-value or to a `String`-object or output them to the screen etc.).

Note: Even a program written in Jasmin (the assembler language for the Java Virtual Machine) can only compare reference values with `==` and `!=` or assign them to a variable.

1.2. Assignment statements

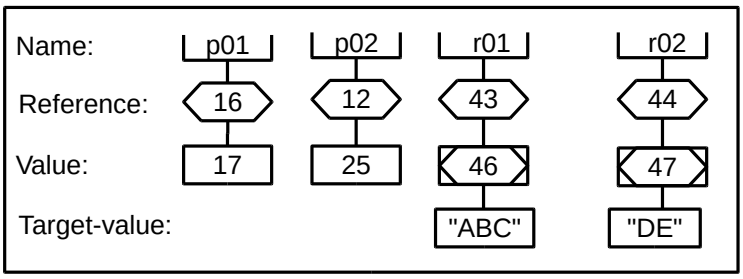
An assignment statement of the form `x = y;` always copies the **value** of variable `y` into the value-box of variable `x`. This has radically different consequences depending on whether `x` and `y` are *primitive variables* or *reference variables*.

Example-02: Assignments between primitive variables vs. between reference variables

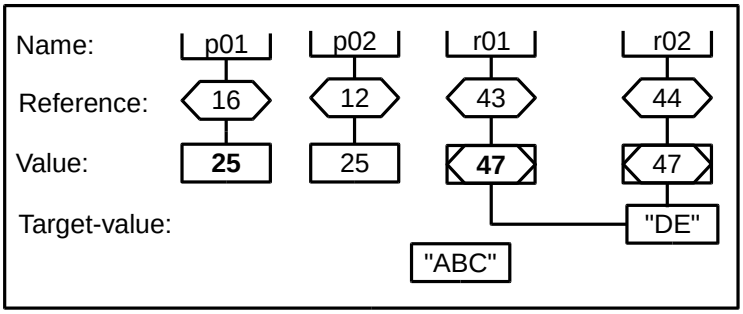
```
1 int           p01 = 17;                       // primitive variable
2 int           p02 = 25;                       // primitive variable
3 StringBuilder r01 = new StringBuilder("ABC"); // reference variable
4 StringBuilder r02 = new StringBuilder("DE");  // reference variable
5
6 p01 = p02; // Assignment 1 (between primitive variables)
7 r01 = r02; // Assignment 2 (between reference variables)
```

As buoys these 4 variables may look (before and after the assignments) as follows:

Before Assignment 1 and Assignment 2:



After Assignment 1 and Assignment 2:



Remember: The *references* `<16>`, `<12>`, etc. and the *reference values* `[<46>]`, `[<47>]` have been chosen by the exer following his particular taste. Do not question his taste.

After the assignment `p01 = p02;` the variables `p01` and `p02` have equal values (viz. `25` and `25`). These values can be modified independently from each other (e.g. with additional assignments).

After the assignment `r01 = r02;` the variables `r01` and `r02` also have equal values (viz. `[<47>]` and `[<47>]`). With these values they refer to (one and) the *same* object (not to two *equal* objects!). If you change this object, you change the target-value of `r01` and of `r02`.

Problem-01: What is being output to the screen?

```
1 r01.append("XY");        // Appends "XY" to the target-value of r01
2 System.out.println(r02); // Outputs the target-value of r02
```

1.3. Different equals-methods

Every Java class contains an object method (a non-static method) with profile

`public boolean equals(Object ob)`

What this `equals`-method does (exactly when it will return `true` and when `false`) is decided by the programmer of the class in question, and therefore may vary widely from class to class.

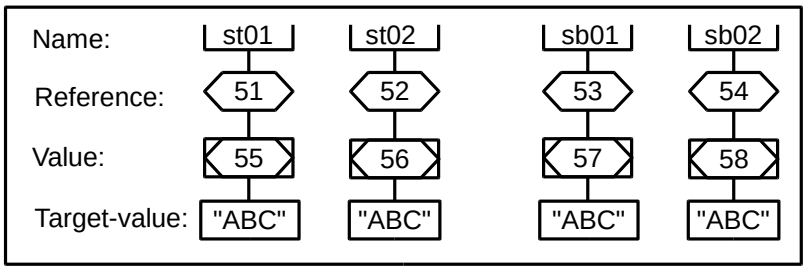
Example-03: `equals`-methods of classes `String` and `StringBuilder`

```
1 String        st01 = new String("ABC");
2 String        st02 = new String("ABC");
3 StringBuilder sb01 = new StringBuilder("ABC");
1 StringBuilder sb01 = new StringBuilder("ABC");
```

new-rule: Each time the `new`-command is called, it generates a new object and returns a reference, which refers to (or: points to) this object. The reference will be different from all references existing so far. It may be assigned to a reference variable (of the appropriate type) as its new value.

From this rule it follows, that the four variables `st01`, `st02`, `sb01`, `sb02` are guaranteed to have four *different values* (because `new` has been called four times).

As buoys the four variables may look as follows:



Please note: The target-values of `st01` and `st02` are `String`-objects, whereas the target-values of `sb01` and `sb02` are `StringBuilder`-objects. The considerable differences between `String`-objects and `StringBuilder`-objects are not shown by the buoys (they have to be inferred from the context).

The programmer of class `String` has decided, that his `equals`-method compares *target-values* (i.e. `String`-objects), not the *values* of `String`-variables. From this it follows that the expression

`st01.equals(st02)`

evaluates to `true` (since `"ABC"` equals `"ABC"`).

The programmer of class `StringBuilder` has decided, that his `equals`-method compares *values* of `StringBuilder`-variables, not *target-values*. From this it follows that the expression

`sb01.equals(sb02)`

evaluates to `false` (since `[<57>]` is not equal `[<58>]`).

End of Example-03.

To check, if two `StringBuilder`-variables refer to equal character sequences, you can convert their target-values (of type `StringBuilder`) to `String`-objects and then compare the `String`-objects, e.g. like that:

`sb01.toString().equals(sb02.toString())`

This expression evaluates to `true` (since `"ABC"` equals `"ABC"`).

Recommended: Before you compare objects of a class `C` with `equals`, you should read the documentation of `C::equals` (i.e. of the object-method `equals` of class `C`).

1.4. The one and only equality operation

The equality operation ==, when applied to variables, always compares their *values* and never their *names*, *references* or *target-values*.

Example-04: The operation ==

In this example, the variables defined in the previous example, are used.

The expression `st01 == st02` evaluates to `false` (because `[<55>]` is not equal `[<56>]`)
The expression `sb01 == sb02` evaluates to `false` (because `[<57>]` is not equal `[<58>]`)

To learn, what the *one* (and only) equality operation `==` does, is much easier than to learn what the *numerous* `equals`-methods are actually doing.

The *inequality operation* `!=` works as follows: Whenever `x == y` throws an exception, `x != y` throws the same exception. In all other cases, `x != y` returns the value `!(x == y)`.

1.5. Special rules for the type String

For each reference type there is a literal `null`. `String` is the only reference type, which has more than this `null`-literal: `"ABC"`, `"How are you?"` and `""` are examples of additional `String`-literals.

`String`-variables may therefore be initialized in two ways:

- with the `new`-command (like variables of other reference types)
- with a `String`-literal (and without `new`).

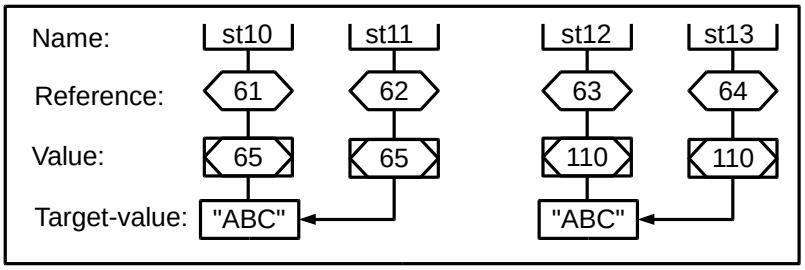
When a `new`-command is used, the **new-rule** applies ("each call of the `new`-command returns a new reference", see above).

StringLiteral-rule: All `String`-variables initialized with the same `String`-literal have *equal values* (and with these values refer to one and the *same* `String`-object).

Example-05: `String`-variables initialized with a literal only, with `new` or with another variable:

```
1 String st10 = "ABC";
2 String st11 = "ABC";
3 String st12 = new String("ABC");
4 String st13 = st12;
```

As buoys these variables may look as follows:



Following the **StringLiteral-rule** the variables `st10` and `st11` have equal values, because they are initialized with the same `String`-literal `"ABC"`.

Following the **new-rule** the value of the variable `st12` is different from the value of `st10`.

The variable `st13` is initialized with the value of `st12` (i.e. `[<110>]`). Therefore `st12` and `st13` refer to the same `"ABC"`-object, but to a different `"ABC"`-object than `st10` and `st11`.

2. How to represent arrays as buoys

The representation of variables by buoys can illustrate the structure of arrays and clarify (among other things) the following differences:

- between arrays with *primitive elements* and arrays with *reference elements*
- between *nested arrays* (native in Java) and *multidimensional arrays* (not native in Java)
- between an *empty array* and a `null` reference

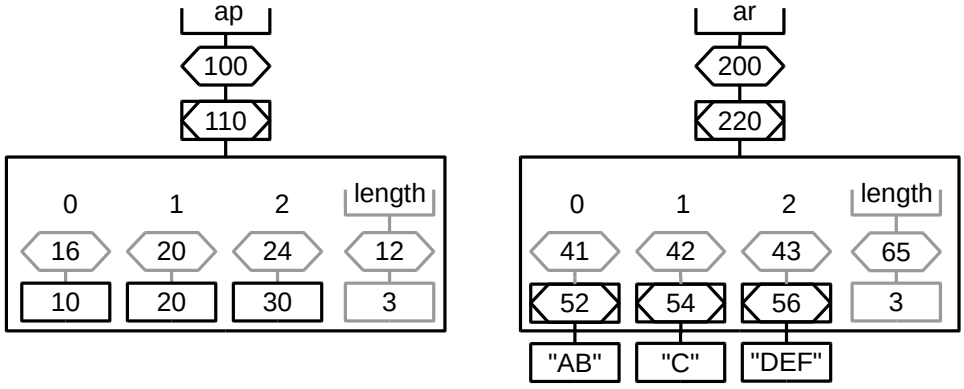
2.1. Primitive or reference elements

An array of type `int[]` contains variables (or: elements) of type `int`. Those elements are located completely *inside* the array. An array of type `String` contains `String`-variables, but the target-values of those variables (the `String`-objects) are located *outside* of the array (not inside). Thus:
1. An "array of objects" does not really contain objects, but only *references* referring to objects.
2. An object may belong to any number of arrays at the same time (whereas a primitive value like `17` or `true` may belong to at most one array, other arrays may only contain copies of it).

Example-01: Two arrays represented by buoys:

```
5    int[]    ap = {10, 20, 30};          // An array with primitive elements
6    String[] ar = {"AB", "C", "DEF");    // An array with reference elements
```

As buoys the variables `ap` and `ar` may look as follows:



Elaborate and abbreviated buoys: Buoys of Java arrays come in two forms: *elaborate* (as shown here) and *abbreviated* (if the gray parts are left out). Thus *references of array elements* are optional, as are all three parts of the variable `length`. Everything else is mandatory. The elaborate form is more realistic, the abbreviated form more convenient. We will use the convenient form most of the time.

The array element `ap[0]` is a *primitive variable* without a name (but we can refer to it with the expression `ap[0]`). Here this variable has the reference `<16>` and the value `[10]` and all its parts are located *within* the array `ap` (which is represented by a largish black rectangle).

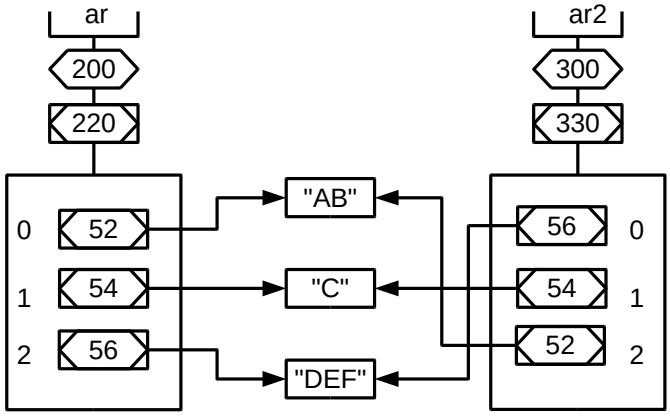
The array element `ar[0]` is a *reference variable* without a name (but we can refer to it with the expression `ar[0]`). Here this variable has the reference `<41>`, the value `[<52>]` and the target-value `["AB"]`. Note, that the target-value is located *outside* the rectangle of the array `ar`.

Remember: All *references* like `<16>`, `<20>`, ... etc. and all *reference values* like `[<52>]`, `[<54>]`, ... etc. have been chosen by the exer following his particular taste.

The array object `ar` (the target-value of the variable `ar`) contains 3 elements and additionally an `int`-variable named `length` with value `3`. In all arrays this `length`-variable is *unmodifiable*.

The buoy of `ar` should make it clear, that the array does not contain `String`-objects, but only *references* that refer to such objects. Thus it is possible, that a `String`-object (which may be very large) could belong to several arrays at the same time, being represented in each of the arrays only by a (relatively small) reference value.

Example-02: Two arrays "containing" the same `String`-objects (buoys in abbreviated form):



The array `ar` "contains" 3 `String`-objects, sorted in ascending order ("AB", "C", "DEF").
The array `ar2` "contains" the same 3 `String`-objects, sorted in descending order ("DEF", "C", "AB").

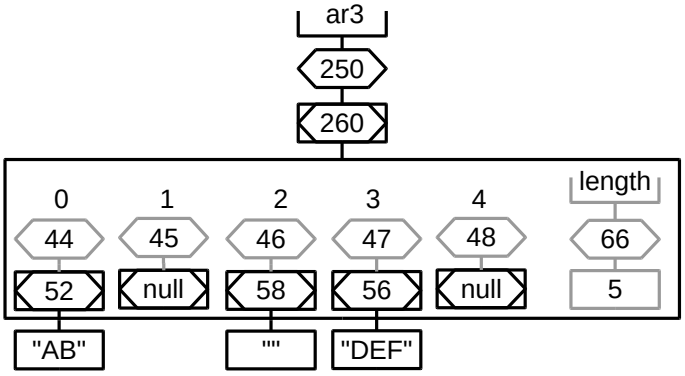
In this example the three `String`-objects are rather small. But if each of them had a size of 1MB, the two arrays `ar` and `ar2` together would not occupy 6 MB of memory, but only slightly more than 3 MB.

An array with elements of a *reference type* may contain `null`-elements. Such elements do *not* refer to any target-value.

Example-03: An array with 2 `null`-elements and an empty-`String`-element

```
7 String[] ar3 = {"AB", null, "", "DE", null);
```

As a buoy (in elaborate form) `ar3` may look as follows:



A `null`-element (like e.g. `ar3[1]` or `ar3[4]`) is fundamentally different from an empty `String`-object (like e.g. `ar3[2]`). A `null`-element does not refer to a target-value. But even an empty `String`-object is a full-blown object, which contains more than 60 methods. Only the number of characters it contains is the smallest possible (i.e. 0).

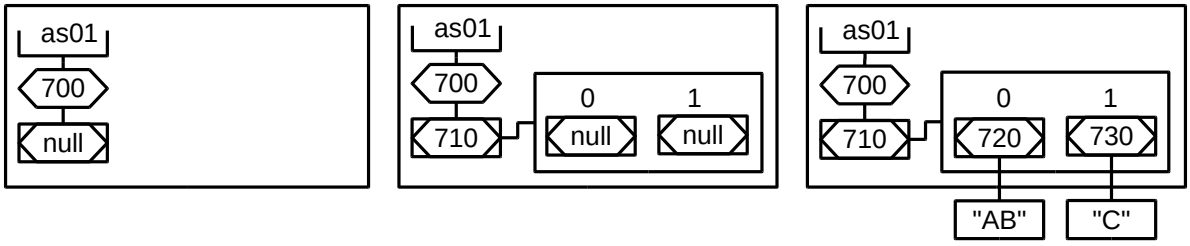
Analogy: To have an *empty cup of coffee* is different from having *no cup* at all.

2.2. Constructing an array in 3 steps or in 1 step

The following sequence of commands will first construct an array `as01` in 3 steps, and then a very similar array `as02` in a single step. The 3-step-construction will be illustrated with 3 snapshots, i.e. buoys, that show how `as01` looks after each step.

```
1    String[] as01 = null;          // step 1
2    as01    = new String[2];       // step 2
3    as01[0] = "AB";                // step 3, part 1
4    as01[1] = "C";                 // step 3, part 2
5
6    String[] as02 = {"AB", "C"};   // All in 1 step
```

The 3 snapshots (in abbreviated form) of `as01` may look as follows:



after step 1 after step 2 after step 3

After step 1: There is an array variable `as01` (which *may* refer to an array), but no array yet.

After step 2: Now the array variable `as01` refers to an array, but the array contains only null-elements.

After step 3: Now the array elements refer to `String`-objects

Problem-02: Draw `as02` as buoy. Which parts are equal to those of `as01` and which parts are guaranteed to be different?

2.3. Nested arrays

Nesting-Rule-1: For every type `T` there is an *array type* `T[]` (pronounced: array of T).
`T` is called the *element type* of the type `T[]`.

This rule applies to all types, including array types: For the element type `T[]` there is the array type `T[][]`, for the element type `T[][]` there is the array type `T[][][]` etc.

In Java every array type (and every array) has a *nesting depth*. This depth is equal to the number of pairs of square brackets `[]` in the type name.

Example-01: Some array types, the pronunciation of their names, nesting depth and element type

Array type	**pronounced**	**nesting depth**	**element type**
`int[]`	array of int	1	`int`
`int[][]`	array of arrays of int	2	`int[]`
`int[][][]`	array of arrays of arrays of int	3	`int[][]`
`String[]`	array of String	1	`String`
`String[][]`	array of arrays of String	2	`String[]`
`String[][][]`	array of arrays of arrays of String	3	`String[][]`

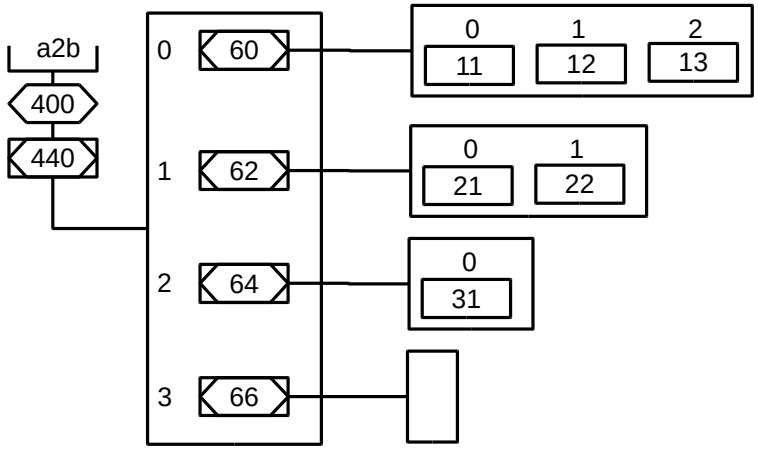
Def.: A **nested array** is an array with nesting depth 2 or greater. Arrays with a nesting depth of 1 are sometimes called *non-nested*.

Example-02: Nested arrays, and a buoy for one of them:

```
1   int[][] a2a = {{11, 12, 13}, {21, 22, 23}, {31, 32, 33}};
2   int[][] a2b = {{11, 12, 13}, {21, 22}, {31}, {}};
```

The array `a2a` has 3 elements of type `int[]` (and indirectly contains 9 `int`-elements).
The array `a2b` has 4 elements of type `int[]` (and indirectly contains 6 `int`-elements).

As a buoy (in abbreviated form) the array `a2b` may look as follows:



This example shows, that the elements of a nested array may be (arrays) of *different lengths*.

Remember: An array of objects does not really contain objects, but only references, which refer to objects. A nested array is an array of array-objects. It does not contain array-objects as its elements, but references, which refer to array-objects.

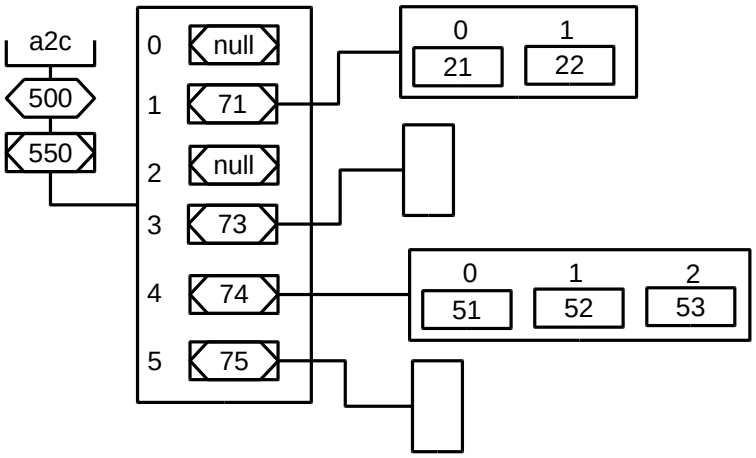
Problem-03: Draw a buoy (in abbreviated or in elaborate form, as you like) which represents the array variable `a2a` of **Example-02**.

Besides "plain vanilla elements" a nested array may contain elements of more exciting flavors too .

Example-03: A nested array with *empty elements* and `null`-*elements*:

```
3   int[][] a2c = {null, {21, 22}, null, {}, {51, 52, 53}, {}};
```

As a buoy (in abbreviated form), the array variable `a2c` may look as follows:



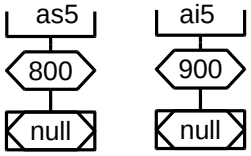
The array `a2c` contains six elements, of which two are empty arrays (`a2c[3]` and `a2c[5]`) and two are `null`-elements (`a2c[0]` and `a2c[2]`).

2.4. Nested arrays and new-commands

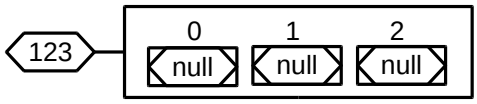
The following declarations

```
1    string[][][][][] as5 = null;
2       int[][][][][] ai5 = null;
```

tell the exer to create 2 variables. As buoys these variables may look like shown on the right. Although the declaration contains a generous amount of square brackets, the buoys look like any other reference variable. But they have a hidden special trait, which is not represented in the buoys, but is well known to the exer.



The values of `as5` and `ai5` seem to be equal, since they both are represented by `null`. But in reality, these `null`-values are of different types and thus incomparable (an expression like `as5 == ai5` would be a syntax error). The `null`-value of `as5` is of the type `String[][][][][]`, whereas the `null`-value of `ai5` is of type `int[][][][][]`.



The following `new`-command:

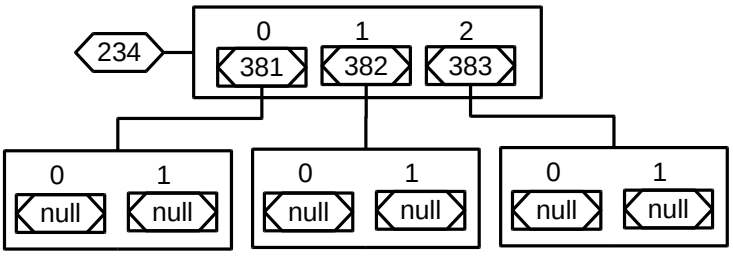
```
... new String[3][][][][] ...
```

generates an array of nesting depth 1, which as a buoy may look like shown on the left, and returns a reference, which refers to this array (here this is the reference `<123>`). This reference is of type `String[][][][][]` and thus may be assigned to the variable `as5`: `as5 = new String[3][][][][];`

The `new`-command `... new int[3][][][][] ...` generates a very similar array and returns a reference referring to it. That reference is of type `int[][][][][]` and therefor may be assigned to `ai5`: `ai5 = new int[3][][][][];`

Problem-04: Of which type are the `null`-elements in the array referred to by the reference `<123>`?

The following `new`-command
`... new int[3][2][] ...`
generates an array with nesting depth 2 as shown on the right, and returns a reference, which refers to this array (here: `<234>`). This reference is of type `int[][][]`. The `null`-elements in the element arrays are all of type `int[]`.



Problem-05: The following `new`-command will return a reference R referring to to a newly generated array A:

```
... new int[2][5][3][][][] ...
```

1. Of what type is R?

2. What depth of nesting does A have?

3. At its bottom the array A will contain some `null`-values. Of which type are they?

2.5. Sometimes nesting is for the birds

For a mathematician, it's an easy exercise to regard *all objects* as arrays. Using an old trick he would call an object, which really is not an array, "an array of nesting depth 0". And he would proudly point out, that with this funny way of speaking the following simple rule would make sense:

Nesting-rule-2: An array has nesting depth n, if all its elements have nesting depth n-1.

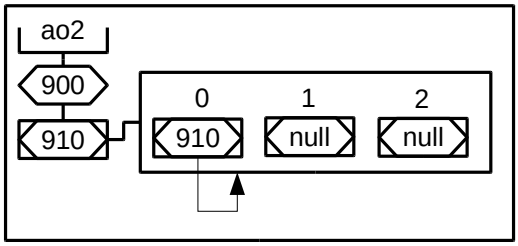
Self-evident as this rule may sound: In some cases it fails to work.

Example-01: An array with a dubious depth of nesting

```
1 String   vs  = "Hello!";              // depth of nesting: 0
2 String[] as  = {"A", "B"};            // depth of nesting: 1
3 int[][]  ai  = {{11, 12}, {21, 22}};  // depth of nesting: 2
4
5 Object[] ao1 = {as, ai, vs};          // depth of nesting?
```

In this example, the array `ao1` is the culprit. It contains as its elements arrays of different depths of nesting. Therefore, its own depth is not defined by **Nesting-rule-2**.

Even worse: An array of type `Object[]` may contain any array as element, *even itself*.



Example-02: An array which contains itself as an element

```
1 Object[] ao2 = new Object[3];
2 ao2[0] = ao2;
```

The **Nesting-rule-2** does not define a nesting depth for the array `ao2`.

2.6. Multidimensional arrays

Apparently, many people are against discrimination, even against discrimination between *nested arrays* and *multidimensional arrays*. This is unfortunate, because the difference between them is interesting and of practical relevance. Roughly speaking: Nested arrays are more powerful (they can solve more problems), but multidimensional arrays (if applicable) are more efficient (they need less memory and execution time).

There is no difference between *non-nested arrays* (with a nesting depth of 1) and *1-dimensional arrays*. But nested arrays (with a nesting depth of 2 or greater) and truly multidimensional arrays (with 2 or more dimensions) are different creatures.

Arrays in Java are nested (or: arrays of arrays, not multidimensional)
(see p. 4 of https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf).
In Fortran and Pascal, there are only multidimensional arrays. In Ada, C++ and C#, both nested and multidimensional arrays are native features.

The important differences between the two kinds of arrays:

A *nested array* contains arrays, which may be of different lengths.

A *multidimensional array* does not contain arrays, but elements of a non-array-type (like `int`, `float`, `String`, ... etc.). To access an element, one needs to supply several indices, one for each dimension, e.g. `mar[3, 5]` would access the element in the 3rd line and 5th column of a 2-dimensional array `mar`.

A multidimensional array has a fixed length for each of its dimensions. E.g. a 2-dimensional array may consist of 10 lines of 15 rows each (all lines have to have the same length). A 5-dimensional array has to "resemble" the shape of a 5-dimensional cuboid.

A nested array is usually implemented as an *array of references* (which refer to arrays).
A multidimensional array is usually implemented as a non-nested (or: 1-dimensional) array plus some fancy index-fiddling.

People at IBM have extended Java by some commands for the generation and handling of multidimensional arrays (see http://researcher.watson.ibm.com/researcher/files/us-bacon/Bacon98JaLA.pdf), but these extensions require a special IBM-Java-compiler.

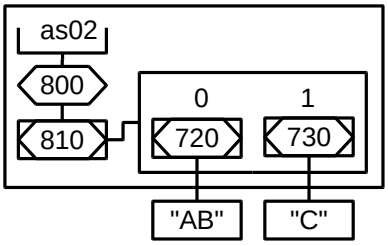
3. Solutions for the Problems

Solution for Problem-01: What is being output to the screen?

```
1 r01.append("XY");        // Appends "XY" to the target-value of r01
2 System.out.println(r02); // Outputs the target-value of r02
```

Output: DEXY

Solution for Problem-02: Draw as02 as buoy. Which parts are equal to those of as01 and which parts are guaranteed to differ?



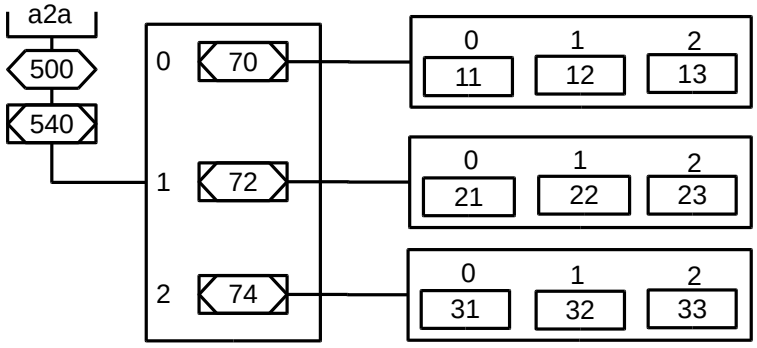
The reference <800> and value [<810>] of as02 are guaranteed to be different from those of as01.

The values [<720>] and [<720>] of as02[0] and as02[1] are guaranteed to equal the values of as01[0] and as01[1].

Solution for Problem-03: Draw a buoy (in abbreviated or in elaborate form, as you like) which represents the array variable a2a of **Example-02**.

```
1    int[][] a2a = {{11, 12, 13}, {21, 22, 23}, {31, 32, 33}};
```

The array variable a2a in abbreviated form:



Solution for Problem-04: Of which type are the null-elements in the array referred to by the reference <123>?

They are of type int[][][][].

Solution for Problem-05: The following new-command will return a reference R referring to to a newly generated array A:

```
... new int[2][5][3][][][] ...
```

1. Of what type is R? **Of type int[][][][][]**
2. What depth of nesting does A have? **A has nesting depth 3**
3. At its bottom the array A will contain some null-values. Of which type are they? **They are of type int[][]**