Arrays: Review and implementation

(contains some material from slides accompanying Horstmann: Java for Everyone: Late Objects, John Wiley and Sons Inc)

Arrays

- A Computer Program often needs to store a list of values and then process them
- For example, if you had this list of values, how many variables would you need?
  - `double input1, input2, input3….`
- Arrays to the rescue!

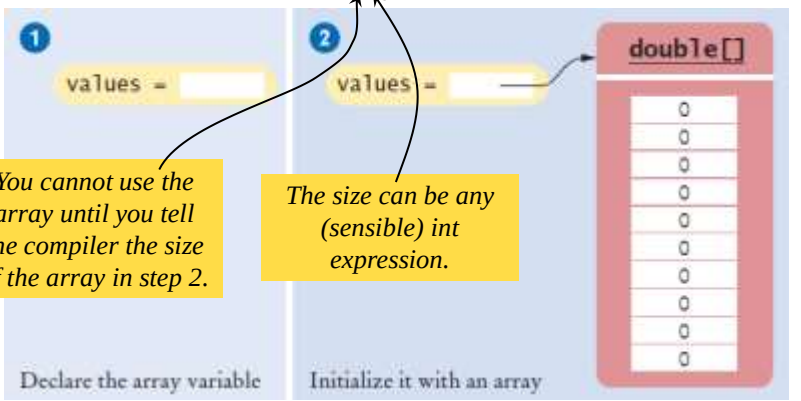
32
54
67.5
29
35
80
115
44.5
100
65

An array collects sequences of values of the same type.

Declaring an Array

- Declaring an array is a two step process

```
1) double[] values;   // declare array variable
2) values = new double[10];  // initialize array
```



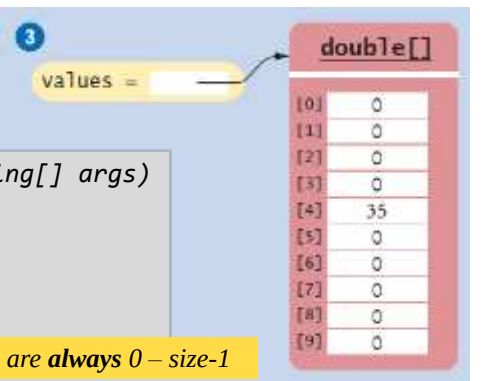
Accessing Array Elements

- Each element is numbered
  - We call this the *index*
  - Access an element by:

```
Name of the array
Index number
values[i]
```

In `values[i]` the `i` can be any (sensible) int expression

Elements in the array `values` are accessed by an integer index `i`, using the notation `values[i]`.



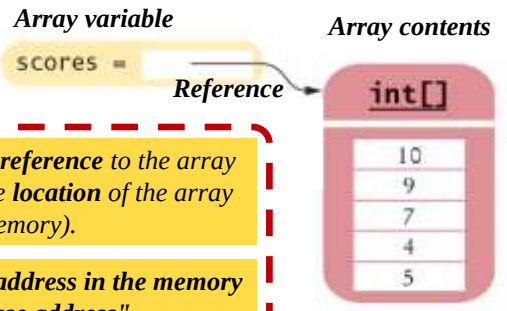

```
public static void main(String[] args)
{
  double values[];
  values = new double[10];
  values[4] = 35;
}
```

*Indices are **always** 0 – size-1*

Array References

- Make sure you see the difference between the:
  - Array variable: The named 'handle' to the array
  - Array contents: Memory where the values are stored

```
int[] scores = { 10, 9, 7, 4, 5 };
```

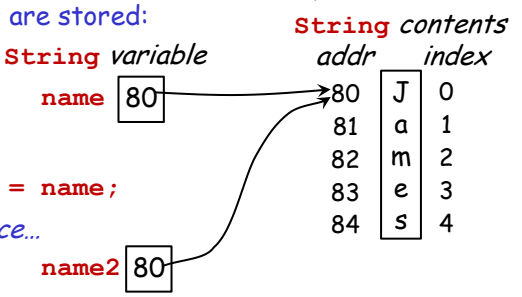


*An array variable contains a **reference** to the array contents. The reference is the **location** of the array contents (in memory).*

The location is normally the address in the memory of the first byte: "base address".

An analogy with Strings

- The concepts of variable, reference and contents apply to Java `String`s too
  - A string variable does ***not*** "contain" the string
  - It contains a reference to an area of memory where the string's characters are stored:



- An assignment `name2 = name;`
  - *Copies the reference…*
  - *NOT the string!*
  - Both variables point to the *same* string
  - **The same applies to arrays: e.g. `oldScores = scores;`**

Comparing Strings

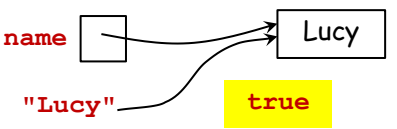
- In Java `==` compares "primitive" data values
  - Usually simple values such as `int`s, `float`s, `boolean`s
- But strings are not "primitive"
  - The "primitive" value of a variable containing a string is the *reference* and *not* the string itself
  - The "primitive" value of a string is the reference to where the string's characters are stored
- So, these compare *two references*

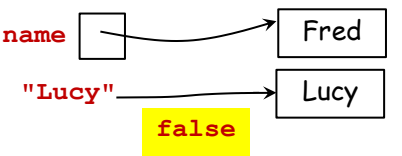
```
name == "Lucy"
name == name2
"Lucy" == "Lu" + "cy"
```

  - Comparing references may give unexpected results…

Comparing Strings

- So, `name == "Lucy"` compares *two references*
  - It asks whether the strings `name` and "Lucy" are *the same string*





  - Also will be `false` if `name` contains "Lucy" but at a *different memory location:*

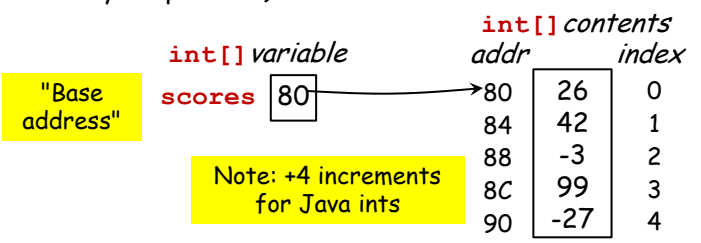
- Again, the same applies to arrays
- Compare strings for *content equality* using `name.equals("Lucy")`

Allocating memory for arrays and strings

- The memory locations for the *contents of* arrays and strings are *not* allocated in the run time stack:
  - The compiler cannot necessarily know how big they will be
  - And they are *not* allocated/dealloc in "stack order"
  - But the variables *that reference them* are in the stack
- Arrays and strings are allocated sections of a separate memory area known as "*the heap*"
  - Not used in a structured way, unlike the stack
  - So it has an "untidy" name!
- Unused sections of the heap are allocated by **`new`**
  - And may be recycled when *no more variables refer to them*
  - "*Garbage collection*" (a JVM responsibility)
- So, array and string *variables* have memory locations in the stack
  - But hold *references to memory blocks in the heap*

Addressing array elements in memory

- Consider the layout in heap memory of a Java **`int`** array (with 4 bytes per **`int`**)



- Compiling expression **`scores[i] + 3`** :

```
Load scores[i] into Ra
MOV 03 -> Rb
ADDI Ra, Rb -> Ra
```

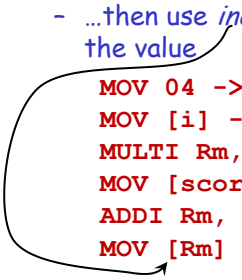
But how? The compiler cannot know an address for **`scores[i]`**

Addressing array elements in memory

- Compiling `scores[i] + 3` :

  `Load scores[i] into Ra`

  - `scores` contains the "base address" of the array memory
  - `i` contains how many blocks of four bytes must be "skipped" to find the `int` that we need
  - Can ***calculate*** the address…
  - `scores + 4 * i`
  - …then use *indirect addressing* to fetch the value



```
MOV 04 -> Rm
MOV [i] -> Rn
MULTI Rm, Rn -> Rn      4 * i
MOV [scores] -> Rm
ADDI Rm, Rn -> Rm       scores + 4 * i
MOV [Rm] -> Ra          load scores[i] into Ra
```

addr		*index*
80	26	0
84	42	1
88	-3	2
8C	99	3
90	-27	4

Indirect addressing and other addressing

- "Addressing modes" determine the kinds of *operands* that may appear in machine instructions
  - See Wikipedia: Addressing mode
- The original Brookshear machine has:
  - Register: e.g. MOV 35 -> R1
  - Immediate/literal: e.g. MOV 35 -> R1
  - Absolute/direct: e.g. MOV [80] -> R1
- Now we have seen: (not in original Brookshear)
  - ***Register indirect***: e.g. MOV [R1] -> R2
    - Means: "use the memory location whose address is contained in R1" – so computable, not fixed
- And also usually available is: (not in original Brookshear)
  - "Base + offset"/indexed: e.g. MOV [R1]+2 -> R2

End of lecture