**Arrays – (Declarations and Rules)**

One dimensional:

a) int**[]** iarr = new int[5];  
b) int iarr**[]** = new int[5];

Multi dimensional:  
the concept of multi dimensional arrays in Java is „arrays in array“.  
That means for example in a two dimensional integer array, that the first dimension of the array includes arrays filled with integers:  
  
a) int**[][]** iarr = new int[5][3];  
b) int iarr**[][]** = new int[5][3];  
c) int**[]** iarr**[]** = new int[5][3];

for multi-dimensional arrays only the first dimension has to be declared.  
Further dimensions can be assigned later:  
  
int[][][] iarr = new int**[5][][];**int[][] ia= new int[2][3];  
iarr[0] = ia;

dimensions must be declared from left to right order without any gaps:   
  
int[][][] iarr = new int[5]**[]**[4]; **Error**int[][][] iarr = new int[5]**[3]**[]; **OK**

**Initializing Arrays**

1. **using an initializer block:**

int[] numbers = **{1,2,3}**; **Ok**

int[][] twoDimNumbers = { {1,3},{7,8,9},{33,66,88,1} }; **Ok**

But look out!  **This cannot be in two statements!**For instance the following will provide an error:

int[] numbers;  
numbers = **{1,2,3};** **Error**

Solution:

int[]numbers;  
numbers = **new int[]{1,2,3};** **Ok** assigning an anonyme array

1. **Using a loop**

for(int i=0; i < ia2.length; i++){  
 ia2[i]=i;  
}

Look out! The Enhanced for-loop is not suitable for initialization

int i = 0;  
 for(int a : ia2){  
 a = i++;  
}

This assigns a value to the copy of the array element, rather than to the array element itself.  
The same applies with reference types.

**Variable Arguments (Varargs):**

Used to define an array as parameter of a method.

The syntax of Varags definition as a method parameter:   
**int ... a** *or* **int...a** *or*  **int ...a**

Example:

void printArray(**int ... a**) {  
 for(int e : a) {  
 System.out.println(e);  
 }

Valid calls of above method:

a) passing an anonyme array:

printArray( **new int[] {1, 2, 5, 8, 9, 22}** );

b) passing a parameter list:

printArray (**10,20,30,40,50,60**);

c) passing nothing –> an empty array will be passed (Not a null-array!):

printArray**()**;

**Important:**

1. If a method has an array as parameter like printArray(**int[] a**), we have to pass an array into it. Parameter list or empty call is not possible as it would be in case of using Varargs!
2. If additional parameters must be defined, those must be before the Varargs definition  
   In othe words: the Varargs parameter must be the last parameter in a method definition:

printArray(**int i,** **int... a**) **OK;**

printArray(**int... a, int i**) **Error;**

**Casting primitive type arrays:**

No conversion is possible to each other even if their element types are  
convertible

int[] intArray = new int[6];

byte[] byteArray = new byte[4];

**intArray = byteArray;** **Error**

**Casting reference type arrays:**

Given the following class hierarchy:

class A {}  
 class C {}  
 class B extends C {}

a) Casting the whole array of a reference type

Object[] o = new A[3]; **up-casting**

A[] a = **(A[])**o; **down-casting**

b) Up-casting elements only   
  
Object[] objarr = new Object[2];

objarr[0]= new A(); **Ok**

objarr[1]= new A(); **Ok**

A[] arr = **(A[])**objarr; **Compiler Ok**, but **Run Time Exception**

Look out! Although all elements of the Object type array are of type A, we still cannot down-cast the Object type array to an array of type A. This will fail at run time!

c) Elements of an up-casted array are also up-casted  
  
C[] carr = new B[2]; **up-casting**

B[] bbb = {new B(), new B()};

carr = bbb; **now all Elements of carr are of type C**

B b = **(B)**carr[1]; **and have to be down-casted**

d) casting with null-elements

C[] carr2 = new C[2]; **Elements in C are null**

B b2 = **(B)**(carr2[1]); **down-casting works. No exception at Run Time!**

But in this case we get run time exception:

C[] carr2 = new C[2];

carr2[1] = new C(); **Element type is C**

B b2 = **(B)**(carr2[1]); **Compiler OK**, but **Run Time Exception  
 Element type is C (not B) !**

e) using an Array as Object

C[] carr = new B[2];

Object obj = carr; **OK - arrays are Objekts**

Object obj = new int[3]; **OK - primitive type arrays are Objects too.**

int[] z = (int[])obj; **OK to down-cast back to primitive type array**

f) Look out with Wrappers:

Integer[] f = new int[3]; **Error**

no conversion between primitive type arrays and wrapper type arrays as it would be the case for elements (auto boxing)!

Remember: Integer i = 3;

h) Reference type arrays are convertible, only if the element type is convertibleC[] cc = new C[3]; **Ok** B[] bb = new B[6]; **Ok**

**cc = bb**;

But look out! Into the array „cc“ we can now assign only B-type elements or its descendants:  
cc[0]=**new B();** **Ok**cc[0]=**new C();** **Error** **runtime exception**

i) Look out with empty arrays:

int[] j = **{}**;

or

int[] j = new int**[0]**;

j[0] = 5; **Error ArrayIndexOutOfBoundsException**

j) look out with value range of primitive types

byte[] ba2 = new byte[3];

ba2[0]=**130**; **Error – byte value can be max. 127**

k) if asssigning primitive arrays to each other the number of dimensions left (DimL) and right  
 (DimR) in the assignment must be the same!

int**[][]** iar = new int**[3][]**; **Ok**

DimL == DimR

l) if assigning reference arrays to Object arrarys the number of dimensions of the reference   
 array must be **same or higher** than the dimension of the Object array.

Object[][] oar = new String[3][]; **Ok**

= new String[3][][]; **Ok**

= new String[3] **Error**

DimL < = DimR

m) if assigning primitive arrays to Object arrarys the number of dimensions of the primitive array must be **higher** than the dimension of the Object array.

Object[][] oar = new int[3][][]; **Ok**

= new int[3][]; **Error**

= new int[3] **Error**

DimL < DimR