Multidimensional arrays:

```java
public class K4B13E_Matrix {
   public static void main(String[] args) {

      int n = Integer.parseInt ( args [0] );
      int m = Integer.parseInt ( args [1] );

      int [][] matrix = new int [n] [m];

      for ( int i = 0; i < matrix.length; i++ )
         for ( int j = 0; j < matrix[i].length; j++ )
            matrix[i][j] =10000+ i*100+j;

      for ( int i = 0; i < matrix.length; i++ )  {
         for ( int j = 0; j < matrix[i].length; j++ )
             System.out.print(matrix[i][j] + "␣␣");
         System.out.println();
      }
   }
}
```

```
                         10000 10001 10002 10003 10004
java K4B13E_Matrix 3 5 ⤳ 10100 10101 10102 10103 10104
                         10200 10201 10202 10203 10204
```

The rows of multidimensional arrays can have different lengths:

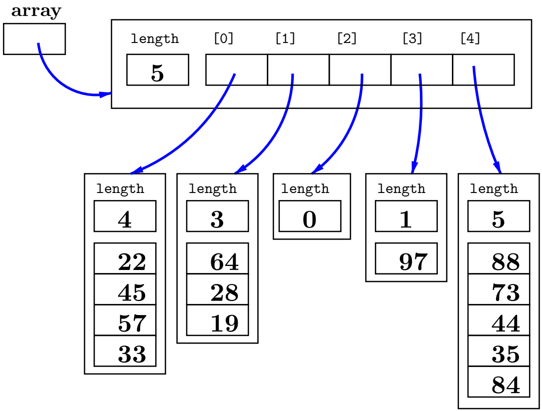
```java
public class K4B14E_ArrayArray {
  public static void main(String[] args) {
    int [] [] array = {
           {22, 45, 57, 33},
           {64, 28, 19},
           {},
           {97},
           {88, 73, 44, 35, 84}
     };

    System.out.println("number of rows "+ array.length);
    for ( int i = 0; i < array.length; i++ ) {
      System.out.print(i + "␣[" + array[i].length + "]␣:␣");
      for ( int j = 0; j < array[i].length; j++ )
        System.out.print(array [i][j] + "␣␣");
      System.out.println();
} } }
```

⤳

```
number of rows 5
0 [4] : 22 45 57 33
1 [3] : 64 28 19
2 [0] :
3 [1] : 97
4 [5] : 88 73 44 35 84
```

```
    int [] [] array = {
            {22, 45, 57, 33},
            {64, 28, 19},
            {},
            {97},
            {88, 73, 44, 35, 84}
```

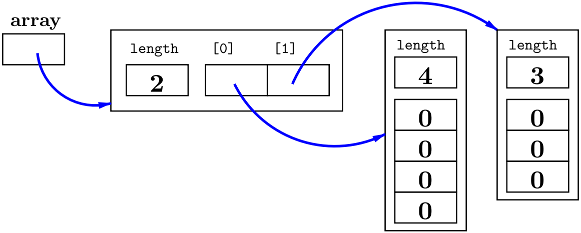


Stepwise memory allocation for multidimensional arrays:

```
  int array[][];
  array = new int[ 2 ][ ];
// memory allocation for the first dimension

  array[ 0 ] = new int[ 4 ];
// memory allocation for subarray 0

  array[ 1 ] = new int[ 3 ];
// memory allocation for subarray 1
```



Options for specifying the number of dimensions:

```
int [] a;                     // ok, one dimension
int b [];                     // ok, one dimension
int [] c, d, e;               // ok, one dimension
int f [][];                   // ok, two dimensions
int g [][][];                 // ok, three dimensions
int [] oneD, twoD [];         // ok
```

`oneD` has one dimension, but **`twoD`** has two dimensions!

Permitted order for partial allocation:

```
int [] [] [] a;               // ok
a= new int[7] [] [];          // ok
a= new int[7] [4] [];         // ok
a= new int[7] [4] [6];        // ok
```

Illegal order for partial allocation:

```
a= new int [] [4] [];         // wrong!
a= new int [] [4] [6];        // wrong!
a= new int [] [] [6];         // wrong!
a= new int [7] [] [6];        // wrong!
```

Excerpt from `java.util.Arrays`:

```
public static void sort ( int[] anArray )
```

- Example call: **`Arrays.sort ( anArray );`**
- sorts **`anArray`** using Quicksort algorithm
- useful for huge arrays
- analogously for **`long`**, **`short`**, **`char`**, **`byte`**, **`boolean`**, **`float`**, **`double`**

```
public static boolean equals (int[] anArray, Object anObject)
```

- Example call: **`Arrays.equals (anArray, anotherArray);`**
- yields true if **`anotherArray`** is array of the same type as **`anArray`**, has the same length and the elements in identical positions have the same values
- analogously for **`long`**, **`short`**, **`char`**, **`byte`**, **`boolean`**, **`float`**, **`double`**

```
public static int binarySearch ( int[] anArray, int value )
```

- Example call:
  `index = Arrays.binarySearch ( anArray, 4711);`
- yields index of the element in **`anArray`**
  which contains the value **`value`**
- since ‘binary search’ is performed, **`anArray`** must be sorted
- if **`value`** is not found, a negative value $\boldsymbol{r}$ is returned;
  $-\boldsymbol{r} - \mathbf{1}$ denotes the position where **`value`** would need to be inserted to keep the array sorted.
- interesting for very huge arrays
- analogously for **`long`**, **`short`**, **`char`**, **`byte`**, **`boolean`**, **`float`**, and **`double`**

from **java.lang.System**:

```
public static void arraycopy (Object src,  int srcPos,
                        Object dest, int destPos, int length)
```

- copies an excerpt of length **length**
  from array **src** (starting at position **srcPos**)
  to the array **dest** (starting at position **destPos**)
- Attention: If the array contains references, this copies only the references, NOT the data!

from **java.lang.Object**:

```
protected Object clone() throws CloneNotSupportedException
```

- Example call (with required type cast):
  int[][] anotherArray = (int[][])
  anArray.clone();
- generates ‘clone’ (full copy) of an array
- Attention: If the array contains references, this copies only the references, NOT the data!

Example program `K4B15E_AsciiArt`

2D graphics made of single characters, with animation, width/height through command-line parameters (here 60x6)

```
X.....X.....X.....X.....X.....X.....X.....X.....X.....X.....
.X.....X.....X.....X.....X.....X.....X.....X.....X.....X....
..X.....X.....X.....X.....X.....X.....X.....X.....X.....X...
...X.....X.....X.....X.....X.....X.....X.....X.....X........
....X.....X.....X.....X.....X.....X.....X.....X.....X.......
.....X.....X.....X.....X.....X.....X.....X.....X.....X......
```

Uses the following initialization method from **`java.util.Arrays`**:

```
public static void fill ( int[] anArray, int value)
```

- Example call **`Arrays.fill (anArray, 42);`**
- sets all elements of **`anArray`** to the value **`value`**
- analogously for **`long`**, **`short`**, **`char`**, **`byte`**, **`boolean`**, **`float`**, **`double`**

```
import java.util.Arrays;
public class K4B15E_AsciiArt {
   public static char [][] image;

   public static void repaint(){
      for (int row=0; row< image.length; row++)
          System.out.println(new String(image[row]));
      if ( ! "".equals(System.console().readLine("\n\n")))
          System.exit(0);
   }

   public static void main(String[] args) {
      int width = Integer.parseInt(args[0]);
      int height  = Integer.parseInt(args[1]);
      image = new char [height][width];

      for (int row=0; row< height; row++)
         Arrays.fill (image[row],’.’);

      int x=0, y=0;
      while (true) {
          image[y][x]=’X’;
          repaint();
          x = (x + 1) % width;
          y = (y + 1) % height;
}  }  }
```

Example program **`K4B16E_Snake`**

animated snake in 2D graphics,
width/height/length through command-line parameters (here 60/6/25)

```
                     =         =
                    = =       = =       W
                   =   =     =   =     =
                  =     =   =     =   =
               = =       = =       = =
                =         =         =
```

Here **`repaint()`** as before, but modified **`main`** method:

```java
   public static void main(String[] args) {
      if (args.length != 3) {
         System.out.println("call: java K4B16E_Snake "
                         + "width height length");
         return;
      }

      int width = Integer.parseInt(args[0]);
      int height  = Integer.parseInt(args[1]);
      int length = Integer.parseInt(args[2]);
      image = new char [height][width];
```

```
  for (int row=0; row< height; row++)
      Arrays.fill (image[row],'␣');

  int xa=0, ya=0, dxa=1, dya=1;
  int xe=0, ye=0, dxe=1, dye=1;
  while (true) {

      image[ya][xa]='W';

      repaint();

      image[ya][xa]='=';
      xa = xa + dxa; if (xa>=width-1 || xa<=0) dxa = -dxa;
      ya = ya + dya; if (ya>=height-1  || ya<=0) dya = -dya;
      if (length > 0) {
        length --;
      } else {
        image[ye][xe]='␣';
        xe = xe + dxe; if (xe>=width-1 || xe<=0) dxe = -dxe;
        ye = ye + dye; if (ye>=height-1  || ye<=0) dye = -dye;
      }
   }
}
```

Excercises on dealing with references on arrays:

(1) How does one compare one-dimensional arrays?

```
import java.util.Arrays;
public class K4B17E_ArrayEq_1_dim {
   public static void main(String[] args) {

      int [] a = new int [4];
      int [] b = new int [4];
      boolean eq;

// wrong:
      eq = (a == b);
      System.out.println("wrong:  identical = " + eq );

// correct:
      eq = Arrays.equals (a, b);
      System.out.println("correct: identical = " + eq );
   }
}
```

(2) How does one compare two-dimensional arrays?

```
import java.util.Arrays;
public class K4B18E_ArrayEq_2_dim {
   public static void main(String[] args) {
      int [][] a = new int [4][5];
      int [][] b = new int [4][5];
      boolean eq;

// wrong:
      eq = Arrays.equals (a, b);
      System.out.println("wrong: identical = " + eq );

// correct:
      eq = true;
      if (a.length != b.length ) {
      eq = false;
      }  else {
        for (int i = 0; i < a.length; i++)
          if ( !Arrays.equals ( a[i], b[i]) ) {
            eq = false;
            break;
          }
      }
      System.out.println("correct: identical = " + eq );
}  }
```

Results of the comparisons:

- **`K4B17E_ArrayEq_1_dim`**:
  The arrays **`A`** and **`B`** have the same content, but different references!

  In corresponding array elements, the same values are stored (here: 0).
- **`K4B18E_ArrayEq_2_dim`**:
  The arrays **`A`** and **`B`** do not have the same content (references!), but the sub-arrays have the same contents

(3) **K4B19E_ArrayTests**:

output method for (small) two-dimensional arrays:

```
...

  static String atos (int[][] array, String arrayName) {
    String output = arrayName + ":\n";
    for (int i = 0; i < array.length; i++) {
      output += "[";
      for (int j = 0; j < array[i].length; j++) {
        output += "␣" + array[i][j] + "␣";
      }
      output += "]\n";
    }
    return output;
  }

...
```

(4) **K4B19E_ArrayTests**:

first attempt to copy a two-dimensional array:

```
...

     int [] [] a = { {22, 45, 57, 33},
                     {64, 28, 19},
                     {},
                     {97},
                     {88, 73, 44, 35, 84} };

     int [] [] b = new int [5][];

     for (int i = 0; i < a.length; i++)
        b [i] = a [i];

     a[1][1] = 0;

     System.out.println( atos(a, "array a") );
     System.out.println( atos(b, "array b") );

...
```

(5) **K4B19E_ArrayTests**: second attempt to copy a two-dimensional array:

```
...

     int [] [] c = { {22, 45, 57, 33},
                     {64, 28, 19},
                     {},
                     {97},
                     {88, 73, 44, 35, 84} };

     int [] [] d = new int [5][];

     for (int i = 0; i < c.length; i++)
        d [i] = (int[]) c[i].clone();

     c[1][1] = 0;

     System.out.println( atos(c, "array c"));
     System.out.println( atos(d, "array d"));

...
```

results of the copy methods:

```
array a:
[ 22  45  57  33 ]
[ 64  0  19 ]
[]
[ 97 ]
[ 88  73  44  35  84 ]
```

```
array b:
[ 22  45  57  33 ]
[ 64  0  19 ]
[]
[ 97 ]
[ 88  73  44  35  84 ]
```

```
array c:
[ 22  45  57  33 ]
[ 64  0  19 ]
[]
[ 97 ]
[ 88  73  44  35  84 ]
```

```
array d:
[ 22  45  57  33 ]
[ 64  28  19 ]
[]
[ 97 ]
[ 88  73  44  35  84 ]
```

4 Non-primitive data types in Java

- Strings
- Arrays
- StringTokenizer
- StringBuffer

java.util.StringTokenizer

- allows to access the individual tokens ((e.g., words or numbers) in a String.
- constructor methods

```
public StringTokenizer (String str)

String st = "This is a String";
StringTokenizer tk1 = new StringTokenizer (st);
// breaks String down into tokens "This", "is", "a", and "String".
//------------------------------------------------------------
public StringTokenizer (String str, String delim)

String st2 = "17,␣25,␣77;␣34␣19";
StringTokenizer tk2 = new StringTokenizer (st2, "␣,;");
// breaks String down in tokens "17", "25", "77", "34", and "19",
// where " ", "," and ";" are recognized as separation characters.
//------------------------------------------------------------
public StringTokenizer (String str, String delim,
                        boolean returnDelims)

StringTokenizer tk3 = new StringTokenizer (st2, "␣,;", true);
// additionally returns separation characters as tokens
```

methods:

```
public int countTokens ( )

int num = tk1.countTokens ();
//  yields the number of tokens in tk1
//————————————————————————————————————————————————————————

public boolean hasMoreTokens ( )

while ( tk1.hasMoreTokens ( ) ) { ... }
// checks if there are more tokens in tk1
//————————————————————————————————————————————————————————

public String nextToken ( )

String str;
str = tk1.nextToken ( );
// yields next token from tk1 and removes it
```

Example

```java
import javax.swing.JOptionPane;
import java.util.*;

/*  computes the sum of the input numbers */
public class K4B20E_TokenSum {
  public static void main(String[] args) {
    int sum = 0;
    String numbers = JOptionPane.showInputDialog
      ("input numbers," + "\n(separated by spaces): ");

    StringTokenizer tokens = new StringTokenizer (numbers);

    while (tokens.hasMoreTokens () )
       sum = sum + Integer.parseInt (tokens.nextToken () );

    JOptionPane.showMessageDialog (null, "Sum: " + sum );
  }
}
```

4 Non-primitive data types in Java

- Strings
- Arrays
- StringTokenizer
- StringBuffer

`java.lang.StringBuffer` as supplement to `String`:

- An object of the class **`String`** is immutable,
  every manipulation of a String creates a new object.
- Objects of the class **`StringBuffer`** are mutable (can be modified).
- **`String`** is efficient when mostly static Strings are processed.
- **`StringBuffer`** is more efficient with highly dynamic Strings.

```
 1 public class K4B21E_StringBufferConstructor {
 2    public static void main( String args[] ) {
 3
 4       StringBuffer buffer1 = new StringBuffer();
 5       StringBuffer buffer2 = new StringBuffer( 10 );
 6       StringBuffer buffer3 = new StringBuffer( "hello" );
 7
 8       System.out.println(
 9               "buffer1␣=␣\"" + buffer1.toString() + "\""
10          +  "\nbuffer2␣=␣\"" + buffer2.toString() + "\""
11          +  "\nbuffer3␣=␣\"" + buffer3.toString() + "\"");
12    }
13 }
```

- line 4: empty StringBuffer with default capacity of 16 characters
- line 5: empty String buffer with capacity of 10 characters
- line 6: StringBuffer with capacity of 21 characters (Contents: **hello**, additional space for 16 characters)
- Method **toString** yields String representation of the StringBuffer

Capacity of the buffer compared to the length of the String:

```
public class K4B22E_StringBufferCapacity {
  public static void main( String args[] ){

    StringBuffer buffer = new StringBuffer( "testing␣a␣buffer" );

    System.out.println
              ("buffer␣=␣␣␣␣␣␣" + buffer.toString()
          + "\nlength␣=␣␣␣␣␣␣" + buffer.length()
          + "\ncapacity␣=␣␣␣␣" + buffer.capacity() );

    buffer.ensureCapacity( 50 );
    System.out.println
          ("\nnew␣capacity␣=␣" + buffer.capacity());

    buffer.setLength( 7 );
    System.out.println
          ("\nnew␣length␣=␣␣␣" + buffer.length()
         + "\nbuffer=␣␣␣␣␣␣␣␣" + buffer.toString()
         + "\ncapacity␣=␣␣␣␣␣" + buffer.capacity());
  }
 }
```

- `StringBuffer("str")`: buffer length: `str` + 16;
- `ensureCapacity(n)`: buffer length: max(n, 2*old value+2)

With **public StringBuffer append(...)** Strings (and values of elementary data types) can be added to a StringBuffer:

```
public class K4B23E_StringBufferAppend {
  public static void main( String args[] ){
     String str = "7␣";
     int intVal = 4;
     double floatVal = 11.0f;
     boolean boolVal = true;
     StringBuffer aBuffer = new StringBuffer( ",␣that's␣" );

     StringBuffer buffer = new StringBuffer();

     buffer.append(str);
     buffer.append("+␣");
     buffer.append(intVal).append("␣=␣");
     System.out.println( buffer.toString()
                       + "\ncapacity:␣" + buffer.capacity() );

     buffer.append(floatVal).append(aBuffer).append(boolVal);
     System.out.println( buffer.toString()
                       + "\ncapacity:␣" + buffer.capacity() );
 }  }
```

append modifies the buffer AND returns a reference to it!

StringBuffers are used automatically to implement concatenation of Strings:

```
 int one = 1;
 String aString = "testing␣...", anotherString;

 anotherString = one + ",␣2,␣3,␣" + aString;
```

is internally evaluated as follows:

```
anotherString = new StringBuffer()
      .append(one)
       .append(",␣2,␣3,␣...")
        .append(aString)
         .toString();
```

More methods (also see the J2SE Documentation)

```java
public StringBuffer insert (int offset, String str)
// inserts a String at position offset;
// overloaded method (also for int, long, char, ...)

public StringBuffer reverse ()
// reverse the contents of the buffer

public StringBuffer delete (int start, int end)
public StringBuffer deleteCharAt (int index)
// deletes a character sequence or a single character

public StringBuffer replace (int start, int end, String str)
// replaces a character sequence by str

public void setCharAt (int index, char ch)
// replaces a character at the given position
```

`java.lang.Character` includes some more methods for String processing (self-explaining...):

```java
public static boolean isLetter (char ch)
public static boolean isDigit (char ch)
public static boolean isLetterOrDigit (char ch)
public static boolean isLowerCase (char ch)
public static boolean isUpperCase (char ch)
public static boolean isSpaceChar (char ch)
public static boolean isWhiteSpace (char ch)

public static String toString (char ch)

public static char toLowerCase (char ch)
public static char toUpperCase (char ch)
```

... and finally a (more clever) palindrome tester

```java
public class K4B24E_Palindrom2 {

// remove white spaces, punctuation, etc.
   static String removeJunk(String str) {
      int len = str.length();
      StringBuffer dest = new StringBuffer(len);
      char c;
      for (int i = 0; i < str.length(); i++) {
         c = str.charAt(i);
         if (Character.isLetterOrDigit(c)) {
            dest.append(c);
         }
      }
      return dest.toString();
   }

// generate a reversed String
  static String reverse(String string) {
      StringBuffer sb = new StringBuffer(string);
         return sb.reverse().toString();
   }
...
...
// Test if palindrome (ignoring upper and lower case)
static boolean isPalindrome(String stringToTest) {
      String workingCopy = removeJunk(stringToTest);
      String reversedCopy = reverse(workingCopy);
      return reversedCopy.equalsIgnoreCase(workingCopy);
   }

   public static void main(String[] args) {
      String testString = System.console().readLine
                                ("input test string: ");
      System.out.println("palindrome: " +
                               isPalindrome(testString) );
   }
}
```