(5) Java Introduction: Arrays and Strings

Nico Ludwig (@ersatzteilchen)

TOC

Initial Words

Yes, my slides are heavy.

I do so, because I want people to go through the slides at their own pace w/o having to watch an accompanying video.

On each slide you'll find the crucial information. In the notes to each slide you'll find more details and related information, which would be part of the talk I gave.

Have fun!

Once again the Problem of Code Repetition

- Let's consider following code to read three numbers from the console and output their sum:

```
// <Program.java>
import java.util.Scanner;

public class Program {
        static int promptAndReadNumber() {
            System.out.println("Please enter a number:");
            System.out.println("The number should be greater than ten:");
            Scanner inputScanner = new Scanner(System.in);
            return inputScanner.nextInt();
        }
}
```

```
// Reading three numbers from the console:

int a, b, c;
a = Program.promptAndReadNumber();
b = Program.promptAndReadNumber();
c = Program.promptAndReadNumber();
System.out.println("The sum: "+(a + b + c)+"!");
```

- But what to do, if we need to sum up more than three numbers? A piece of cake! Just add another prompt and variable:

```
// Reading four numbers from the console:

int a, b, c, d;
a = Program.promptAndReadNumber();
b = Program.promptAndReadNumber();
c = Program.promptAndReadNumber();
d = Program.promptAndReadNumber();
System.out.println("The sum: "+(a + b + c + d)+"!");
```

- … and, what to do if want to sum ten numbers? Add six more prompts and six more variables?
  - We already heard about the principle of DRY! What if we apply a loop to solve this problem?
  - Let's do that, it should solve our problem.

Reducing Code Repetition with Arrays – Part 1

- All right, give loops a chance, we'll use a for loop! But … it doesn't compute, we can not formulate the required code:

```
// Reading four numbers from the console:

int a, b, c, d;
for (int i = 0; i < 4; ++i) {
    a??? Huh? = promptAndReadNumber();
}
```

- The loop allows to formulate the repeating prompt, but we can not assign to the four variables to sum up!
- The basic problem are the variables, more exactly, the need to assign four individual values to variables.
- We can solve this problem by using a variable, which can store multiple values at once, keeping a list of variables.
- In Java, variables holding multiple values are called arrays. Let's rewrite the code reading four values from the console:

```
// Reading four numbers from the console:

int a, b, c, d;
a = Program.promptAndReadNumber();
b = Program.promptAndReadNumber();
c = Program.promptAndReadNumber();
d = Program.promptAndReadNumber();
System.out.println("The sum: "+(a + b + c + d)+"!");
```



```
// Reading four numbers from the console using an array:

int[] numbers = new int[4];
numbers[0] = Program.promptAndReadNumber();
numbers[1] = Program.promptAndReadNumber();
numbers[2] = Program.promptAndReadNumber();
numbers[3] = Program.promptAndReadNumber();
System.out.println("The sum: "+(numbers[0] + numbers[1] + numbers[2] + numbers[3])+"!");
```

Reducing Code Repetition with Arrays – Part 2

- Let's review the example using the array:

```
// Reading four numbers from the console using an array:

int[] numbers = new int[4];
numbers[0] = Program.promptAndReadNumber();
numbers[1] = Program.promptAndReadNumber();
numbers[2] = Program.promptAndReadNumber();
numbers[3] = Program.promptAndReadNumber();
System.out.println("The sum: "+(numbers[0] + numbers[1] + numbers[2] + numbers[3])+"!");
```

  - The new aspect in this code is, that it stores values in the array *numbers* and not in individual variables (e.g. *a*, *b*, *c* and *d*).

- With arrays, we can regard the DRY principle: we use a for loop to read multiple values from the console and summing them up.

```
// Reading four numbers from the console and sum them up with a loop:

int[] numbers = new int[4];
int sum = 0;
for (int i = 0; i < 4; ++i) {
        numbers[i] = Program.promptAndReadNumber();
        sum += numbers[i];
}
System.out.println("The sum: " + sum + "!");
```

  - Just the value 4 (which is used twice here) controls how many values are asked from the user and be summed up.

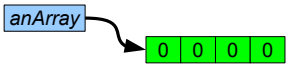
- Of course, this information about arrays is overwhelming, so let us discuss the details about arrays now.

Introduction to Arrays – Part 1

- What is an array?
  - In brief: arrays are like lists of values, an array is a container: it stores a bucket of values.
  - An array variable represents multiple variables kept under one symbol held in one object in memory!

- Let's start having a first glimpse on the definition of an array variable:

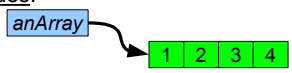
```
int[] anArray = new int[4];
```



  - This statement creates an array, *anArray*, of 4 elements and each element has the value 0.
  - The count of elements in the array is specified in the brackets.
  - => *anArray* actually represents 4 variables, which are called components (Java specification) or elements, they all have the value 0.

- Because *anArray* has only 0-initialized values, we should give the elements some values:

```
for (int i = 0; i < 4; ++i) {
        anArray[i] = i + 1;
}
```



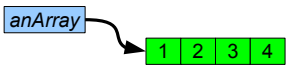
  - This time we use the bracket syntax as an operator to write values into each individual element in *anArray* via their indexes.
  - We use a loop to generate index numbers to access each element in *anArray* exactly. for loops are excellent to work with arrays.
  - Notice, that the indexes are incremented from 0 to 3 (*i* < 4), because counting up the indexes starts at index 0 (not 1).
  - The individual 4 variables aggregated in *anArray* are accessible as *anArray[0]*, *anArray[1]*, *anArray[2]* and *anArray[3]*.
  - After the loop is done the elements have these values: *anArray[0]* = 1, *anArray[1]* = 2, *anArray[2]* = 3 and *anArray[3]* = 4.

Introduction to Arrays – Part 2

```
int[] anArray = new int[4];
```

- The length of an array must be specified or be "concludable", when the array is created. The length is of type int.
- The length of an array is of type int. => Arrays are datatypes, which need other data of type int: the length.

```
for (int i = 0; i < 4; ++i) {
        anArray[i] = i + 1;
}
```



- Because an array is like a list, we need two sorts of data to use an array: the array object and the position in the array.
- The position in an array is also of type int. => Arrays are datatypes, which use other data of type int: the position.
  - We use the counter variable *i* as "position-int" for the []-operator to access each individual element on its position *i* in the array.
  - The value we use as "position-int" for an array is called index.
- Each element can be modified/assigned to with the []-operator and an index just by using assignment operations.
  - The []-operator is usually called index-operator, element-access-operator or array-subscript-operator.

Introduction to Arrays – Part 3

- Now we can use kind of the same for loop we used to write the elements of *anArray* to read the elements of *anArray*:

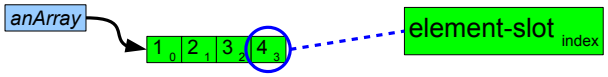
Good to know
Actually, the length of an array, i.e. the count of elements, and the index are of type int, this leads to the fact, that an array cannot have more than 2^{31} - 1 elements and has 2^{31} - 2 as greatest index.

```
// Set the elements' values:
for (int i = 0; i < 4; ++i) {
        anArray[i] = i + 1;
}
// Read the elements of the array:
for (int i = 0; i < 4; ++i) {
        System.out.println(anArray[i]);
}
```

```
Terminal
NicosMBP:src nico$ java Program
1
2
3
4
NicosMBP:src nico$
```

  - We also use the bracket syntax as an operator to get the value of each individual element in *anArray* via its index.
  - Once again a for loop is excellent to generate index numbers to write each element in *anArray*.
  - The individual 4 variables aggregated in *anArray* are accessible as *anArray[0]*, *anArray[1]*, *anArray[2]* and *anArray[3]*.

- To make the association between an individual element and its index more clear, we use a more precise illustration:

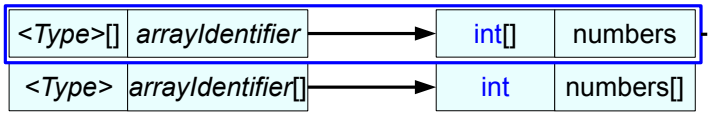


  - The indexes of the elements are notated as subscript numbers in the "element-slot boxes".

- Terminology alert for German programmers: Stick to calling an array array, not "Feld", even if German literature does!
  - A "Feld" is a field of a UDT! This is an example where translation is really inappropriate, leading to ridiculous misunderstandings.

Declaration, Creation and Initialization of Arrays – Part 1

- For the array declaration, Java provides two syntaxes, the array brackets can be put on the type or on the identifier:
  - Remember: the identifier is the name of the variable.

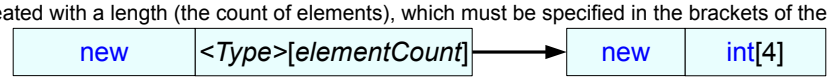


In this course we'll stick to this syntax!

Notice
Arrays can be created from any type!

- Apart from declaration, Java provides many ways to actually create arrays.
  - Creation of arrays means, to reserve a portion of memory for the array.
  - Creation is also called allocation, e.g. allocation of (the memory of) an array, or "instantiation", e.g. "instantiation of an array".

- Arrays can be created with a length (the count of elements), which must be specified in the brackets of the array creation syntax:



  - After an array has been created, it must be filled with values, yes we'll do so with a for loop:

```
int[] numbers = new int[4];
for (int i = 0; i < 4; ++i) {
        numbers[i] = i + 1;
}
```

  - We use the counter variable *i* as index for the []-operator to access each individual element.
  - Each element can be modified/assigned to with the []-operator and an index just by using assignment operations.

Declaration, Creation and Initialization of Arrays – Part 2

- A remarkable fact about Java's arrays is, that the count of elements can be a variable value, it needs not to be a constant:

```
int count = printPrompt("How many elements should the array contain?");
int[] numbers = new int[count];
for (int i = 0; i < count; ++i) {
        numbers[i] = i + 1;
}
```

  - The significant point in this example is, that *count* is a variable value, which is specified by the user at run time.
  - This means, that we create an array with a dynamic count of elements, i.e. a dynamic portion of memory allocated at run time.
  - Remember: In former examples we used the constant 4 for the array length, which was specified as compile time.

- Another remarkable fact about this code: we also used the variable *count* to formulate the loop's termination expression.
  - This is a very important way of programming, we will discuss in short.

- When using array declarators, the length of the array is specified with the brackets to allocate memory.
  - The specification of array length is of type int. – We can conclude: the maximum size of arrays is *Integer.MAX_VALUE*!

- Empty arrays have just no elements at all:

```
// Creating empty arrays:
int[] empty = {};
int[] alsoEmpty = new int[0] ;
```


- If a "data container" should store more than *Integer.MAX_VALUE* elements, another type must be used, which uses keys of a type different from int. – Such "data containers" are called associative collections in Java.

Declaration, Creation, Initialization of Arrays – Part 3

- If we already know the values of the array when it is created, we can initialize arrays with array initializers.

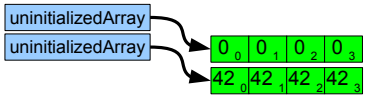
```
// Create and fill an array the "traditional" way:
int[] myArray = new int[3];
myArray[0] = 232;
myArray[1] = 6789;
myArray[2] = 3
```

```
// Creates an int-array with array initializer:
int[] myArray2 = {232, 6789, 3};
```

```
// Creates an int-array with array initializer's alternative syntax:
int[] myArray3 = new int[]{232, 6789, 3};
```

- Finally, arrays can also be created/initialized with special methods, collected in the class *java.util.Arrays*.
  - E.g. we can use the method *java.util.Arrays.fill()* to set all elements of an array to a specific value without using a loop:

```
int[] uninitializedArray = new int[4];
java.util.Arrays.fill(uninitializedArray, 42);
```



- *java.util.Arrays* is a so called companion class, i.e. it is a companion of array objects.
  - *java.util.Arrays* has various other valuable methods worth having a look at, e.g. for copying, searching, sorting and comparison.

```
// Sorting an int[] using java.util.Arrays.sort()
int[] numbers = { 8, 12, 3, 36, 2, 6, 77, 8, 9 };
java.util.Arrays.sort(numbers);
// numbers = [2, 3, 6, 8, 8, 9, 12, 36, 77]
```

```
// Comparing two int[]s using java.util.Arrays.equals()
int[] numbers1 = { 8, 12, 3, 36, 2, 6, 77, 8, 9 };
int[] numbers2 = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
boolean areEqual = java.util.Arrays.equals(numbers1, numbers2);
// areEqual = false
```

12

- In Java arrays can only be created on the heap, the creation syntax does not matter!
- In Java it is not guaranteed that (one dimensional) arrays reside in a contiguous block of memory.

Accessing and Modification of Array Elements

- Accessing and modification of array elements is done with the []-operator, we have already used it in many loops:

```
int[] numbers = new int[4];
for (int i = 0; i < 4; ++i) {
        numbers[i] = i + 1;                    // modify/write value from numbers at index i
        System.out.println(numbers[i]);    // access/read value from numbers at index i
}
```

  - Each element is addressed via its index in the array. Syntactically, the index is the argument, which is passed to the []-operator.
  - Because the array's length is of type int, the index must also be of type int.

- The array-indexes are 0-based. – So the indexes' range is [0, length[. The first valid index is 0 and the last is length - 1!
  - This is why indexes are incremented starting from 0 upwards to i < length in for loops.

```
for (int i = 0; i < 4; ++i) {
        System.out.println(numbers[i]);
}
```

Good to know
Most programming languages or "computer-oriented" notations use 0-based indexes.
Notable exceptions are early BASIC-dialects and Cascading Style Sheets (*:nth-child()*).

- After an array was created and its elements filled with values, we can rewrite the values of those elements at any time.

```
// rewrite all elements in numbers2:
int[] numbers2 = {232, 6789, 3};
for (int i = 0; i < 3; ++i) {
        numbers2[i] = i * i;
}
```

  - The notable fact here: array elements are not in any kind "read only".

Array Length – Part 1

- Let's review this example:

```
int[] numbers = {12, 13, 14};
for (int i = 0; i < 3; ++i) {
        System.out.println(numbers[i]);
}
```

  - Sure, this code will output the three ints 12, 13 and 14 to the console. In the next step, we append the int 15 to the initializer list:

```
int[] numbers = {12, 13, 14, 15};
for (int i = 0; i < 3; ++i) {
        System.out.println(numbers[i]);
}
```

```
Terminal
NicosMBP:src nico$ java Program
12
13
14
NicosMBP:src nico$
```

  - But only three ints 12, 13 and 14 are written to console! – Right, we forgot to tell the loop to iterate the 4th element (3rd index)!

```
int[] numbers = {12, 13, 14, 15};
for (int i = 0; i < 4; ++i) {
        System.out.println(numbers[i]);
}
```

```
Terminal
NicosMBP:src nico$ java Program
12
13
14
15
NicosMBP:src nico$
```

  - Now we will remove the int 12 from *numbers* and write its elements to console:

```
int[] numbers = {13, 14, 15};
for (int i = 0; i < 4; ++i) {
        System.out.println(numbers[i]);
}
```

```
Terminal
NicosMBP:src nico$ java Program
12
13
14
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 3
      at Program.main(Program.java:21)
NicosMBP:src nico$
```

  - What happened now? – A run time error appeared!

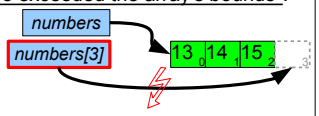
Array Length – Part 2

- So, what happened here?

```
int[] numbers = {13, 14, 15};
for (int i = 0; i < 4; ++i) {
        System.out.println(numbers[i]);
}
```

```
Terminal
NicosMBP:src nico$ java Program
12
13
14
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 3
      at Program.main(Program.java:21)
NicosMBP:src nico$
```

  - On a closer look, we spot the problem: we tried to access *numbers[3]*, i.e. the 4th element, but only three elements are in *numbers*.
  - As a Java programmer we say, that "we've exceeded the array's bounds":



Good to know
In most cases an arrays' bounds are exceeded by exactly one index-position too small/large. This is usually called (the famous) off-by-one error/bug. The problem is also called fencepost-problem, because it is tricky to get the count of fenceposts you need: you have to build a fence of 100m and need a fencepost every 10m
– how many fenceposts do you need? 9, 10 or 11?

- If we excess an array's bound, the Java VM will throw an *ArrayIndexOutOfBoundsException*.
  - *Exceptions* generally indicate run time errors, which terminate the program if we don't do anything about it.
    - We've already dealt with *Exceptions*, e.g. with the *ArithmeticException*, which is thrown, if a division by zero was performed.
  - That isn't as bad as it seems:
    - In other languages array excess can lead to accessing invalid memory, **the mayor gate for security leaks in software**!
    - In languages, which operate close to memory, such as C/C++, array excess is extremely (!) dangerous and usually leads to a disaster.
  - So, in a sense, termination of the program is the better way to deal with array excess!

- It should be said that bounds checked arrays are a milestone in stable programming compared to C/C++, where, e.g. modification of regions exceeding an arrays bounds is undefined. Where we had to debug for hours in C/C++ only to find the bug in the code, spotting and correcting such an error in Java is a piece of cake due to *Exceptions*!

Array Length – Part 3

- All right, the question is, what can we do about this? Answer: we've to avoid array excess by defensive programming!

- But, what was the problem exactly? What do we have to get right?
  - The problem is, that the array variable and the bounds used for the iteration are separately managed in our code.
  - If data is managed separately, which has indeed a logical connection, things will fail, when only a single piece of data is changed.
  - E.g. we have removed one element from numbers, but we kept the iteration from 0 to the old count of elements 4 (3rd index):

```
int[] numbers = {13, 14, 15};
for (int i = 0; i < 4; ++i) {
        System.out.println(numbers[i]);
}
```

- The good news is, that Java's arrays, know and expose their length!
  - Each array has a so called field, *length*, which exposes its length. *length* can be used to write stable array iteration with loops:

```
int[] numbers = {13, 14, 15};
for (int i = 0; i < numbers.length; ++i) { // excellent!
        System.out.println(numbers[i]);
}
```

  - What we have done here was an important step: we replaced the magic number 4 by *numbers.length* to stabilize code!
  - Because we know that an array's bounds are [0, length[, and that we can access an array's length we can write stable loops!
  - In C/C++ the inability of getting or tracking the length of a arrays is a very big disadvantage!

Array Length – Part 4

- On a first glimpse, Java arrays exposing their length sounds not so spectacular, but it is an invaluable feature!

- Real life case: getting the first and last element of an array:

```
int[] numbers = {1, 2, 3, 4};
int firstElementsValue = numbers[0];
int lastElementsValue = numbers[numbers.length - 1]; // numbers.length = 4
```

- Real life case: length of passed array.
  - Like any other type, arrays can be a parameter type in methods.
  - With array params we've exactly a case, in which we don't know the length of arguments in advance:

```
static void printArray(int[] numbers) {
        for (int i = 0; i < numbers.length; ++i) {
                System.out.println(numbers[i]);
        }
}
```

- Real life case: length of returned array.
  - Like any other type, arrays can be a return type of methods.
  - With returned arrays we've another case, in which callers don't know the length of the returned array in advance:

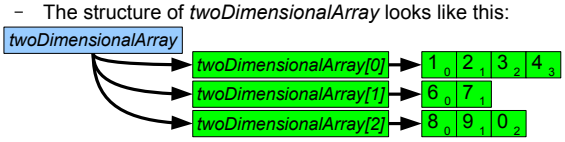
```
int[] numbers = getNumbers();
for (int i = 0; i < numbers.length; ++i) {
        System.out.println(numbers[i]);
}
```

Multidimensional (md) Arrays – Part 1

- Java allows arrays to have other arrays as elements, so that we have arrays of arrays.
  - The depth of cascading of arrays in arrays is called dimension of the array. The arrays we have seen are one-dimensional arrays.
  - Now, we'll create an array, which contains other arrays, i.e. a two-dimensional array:

```
int[][] twoDimensionalArray = new int[3][]; // First dimension: three sub-arrays
// Set the values of the sub-arrays in the second dimension:
twoDimensionalArray[0] = new int[] {1, 2, 3, 4};      // -> Second dimension's' first sub-array with the elements 1, 2, 3 and 4
twoDimensionalArray[1] = new int[] {6, 7};            // -> Second dimension's second sub-arrays with the elements 6 and 7
twoDimensionalArray[2] = new int[] {8, 9, 0};         // -> Second dimension's third sub-arrays with the elements 8, 9 and 0
```

  - The structure of *twoDimensionalArray* looks like this:



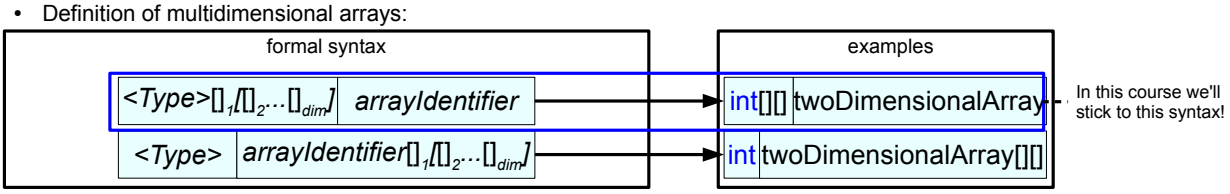
- We can iterate and print all values of *twoDimensionalArray* to the console.
  - We do so by cascading two loops and iterate the array, the sub-arrays and their elements:

```
for (int i = 0; i < twoDimensionalArray.length; ++i) { // loops the first dimension array
      System.out.println("Subarray "+i);
      for (int j = 0; j < twoDimensionalArray[i].length; ++j) { // loops the second dimension's sub-arrays
      System.out.println(" Value: "+twoDimensionalArray[i][j]); // multiple application operator []
      }
}
```

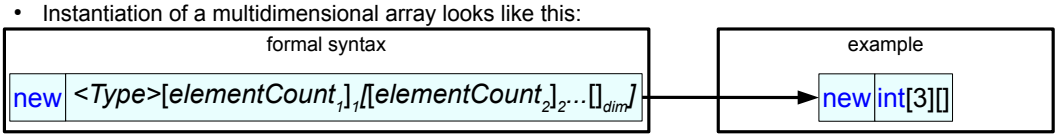
```
Terminal
NicosMBP:src nico$ java Program
Subarray 0
 Value: 1
 Value: 2
 Value: 3
 Value: 4
Subarray 1
 Value: 6
 Value: 7
Subarray 2
 Value: 8
 Value: 9
 Value: 0
NicosMBP:src nico$
```

Multidimensional (md) Arrays – Part 2

- Definition of multidimensional arrays:



- Instantiation of a multidimensional array looks like this:



- Additionally, we can use multidimensional array initializers to create md arrays and providing initial values.

```
// filling an md array with multiple steps:
int[][] twoDimensionalArray = new int[3][];
twoDimensionalArray[0] = new int[] {1, 2, 3, 4};
twoDimensionalArray[1] = new int[] {6, 7};
twoDimensionalArray[2] = new int[] {8, 9, 0};
```

```
// multidimensional array-initializer:
int[][] twoDimensionalArray = {{1, 2, 3, 4}, {6, 7}, {8, 9, 0}};
```

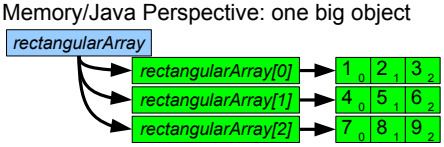
Multidimensional (md) Arrays – Part 3

- In practice, esp. md arrays, in which all sub-arrays have the same length are important. They're called rectangular arrays.

- Rectangular arrays are programmatically equivalent to mathematical matrices:

```
int[][] rectangularArray = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}; // Represents a 3x3 matrix.
```

Memory/Java Perspective: one big object



Mathematical Perspective: a 3x3 matrix

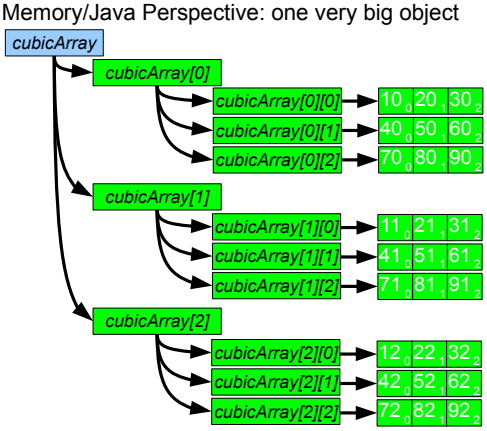
$$rectangularArray = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

Multidimensional (md) Arrays – Part 4

- Rectangular arrays can have any dimension, e.g. they could represent three dimensional cubic matrices:

```
int[][][] cubicArray = {
        {{10, 20, 30}, {40, 50, 60}, {70, 80, 90}},
        {{11, 21, 31}, {41, 51, 61}, {71, 81, 91}},
        {{12, 22, 32}, {42, 52, 62}, {72, 82, 92}}
}; // Represents a 3x3x3 matrix.
```

Memory/Java Perspective: one very big object



Mathematical Perspective a 3x3x3 matrix

$$cubicArray = \begin{pmatrix} \begin{pmatrix} 10 & 20 & 30 \end{pmatrix} & \begin{pmatrix} 40 & 50 & 60 \end{pmatrix} & \begin{pmatrix} 70 & 80 & 90 \end{pmatrix} \\ \begin{pmatrix} 11 & 21 & 31 \end{pmatrix} & \begin{pmatrix} 41 & 51 & 61 \end{pmatrix} & \begin{pmatrix} 71 & 81 & 91 \end{pmatrix} \\ \begin{pmatrix} 12 & 22 & 32 \end{pmatrix} & \begin{pmatrix} 42 & 52 & 62 \end{pmatrix} & \begin{pmatrix} 72 & 82 & 92 \end{pmatrix} \end{pmatrix}$$

Multidimensional (md) Arrays – Part 5

- Multidimensional arrays can basically be processed like one dimensional arrays!
  - Because md arrays are just arrays of arrays, the application of the []-operator on an md array just addresses another array.

```
int[][] matrix = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};

int[] row1 = matrix[0];
// row1 = {1, 2, 3}
```

- In order to loop over, e.g. two dimensions of an md array, we need two cascaded loops to "drill down":

```
for (int i = 0; i < matrix.length; ++i) { // loops the rows
    int[] row = matrix[i];
    for (int j = 0; j < row; ++j) { // loops the columns
        System.out.println("Value: "+row[j]);
    }
}
```



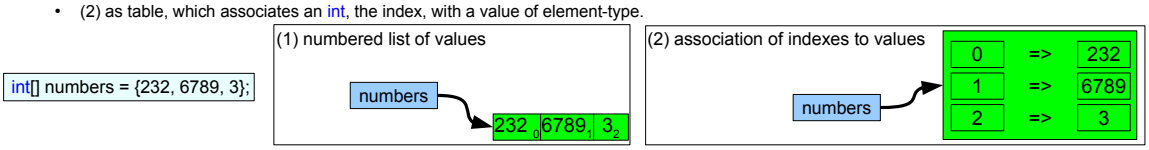
```
for (int i = 0; i < matrix.length; ++i) { // loops the rows
    for (int j = 0; j < matrix[i].length; ++j) { // loops the columns
        System.out.println("Value: "+matrix[i][j]); // multiple application of []
    }
}
```

  - As can be seen, individual elements of a multidimensional array can be accessed by multiple application of the []-operator!

- Some words on multidimensional arrays:
  - + There are very fast.
  - + They are simple to use.
  - - Above two dimensions the code can become arcane.
  - - The usage of multidimensional arrays can quickly excess the machine's memory.
  - => Multidimensional arrays should be avoided in favor of collections! We'll discuss collections in a future lecture.

Basic Features of Arrays – Summary

- Features:
  - Arrays are big objects, representing lists of individual elements, whereby each element has the same (static) element type.
  - An array is a table or list of elements of the same type.
  - We use other data of type int, the index, to access individual elements in the array with the []-operator.
  - Arrays "know" their length, which can be retrieved via arrays' field *length*.
  - Arrays can be understood
    - (1) as list of values of element-type numbered from 0 to *length* - 1, or
    - (2) as table, which associates an int, the index, with a value of element-type.



- Other features we didn't explicitly mention yet:
  - Each individual element can be []-accessed by its index at any time in any order, we say arrays offer random access.
    - E.g. it is not required to access the element at index 2 only after the elements at 1 and 0 have been accessed.
  - []-access to each element takes the same amount of time, we say arrays allow access to their elements with a constant complexity.
    - E.g. []-accessing the element at index 5 takes the same amount of time as accessing the element at index 2.
  - Arrays can neither grow nor shrink! I.e. we can not remove or add elements and an array's *length* is immutable.

Reviewing the Features of Methods – Returning Arrays from Methods

- An arrays can be used as return types of methods, just as any other type:

```
// Get some numbers from the user:
static int[] getNumbersFromUser() {
        final int nNumbers = 10;
        int[] numbers = new int[nNumbers];
        for (int i = 0; i < nNumbers; ++i) {
                numbers[i] = readNumberFromConsole();
        }
        return numbers;
}

int[] numbers = getNumbersFromUser();
```

Good to know
A variable denoting the count of elements in an array (or any kind of "collection"), is commonly accepted to be a candidate for a variable prefix: 'n'. The idea is, that when ***numbers*** denotes an array of, well, numbers, it seems logical to have another variable denoting count of elements in that array with the count of numbers, or in short ***nNumbers***.

- The ability to return arrays from methods unleashes the power of Java's arrays:
  - Memory management is not our business as Java developers! – The JVM cares for this at run time.
  - The caller does also not need to know the length of the returned array. – The length can be retrieved by reading *numbers.length*.
  - (We defined *nNumbers* as length of the array to be created as compile time constant, but this is not required in Java.)
  - Returning arrays from languages like C/C++ is a much harder task.

- We discuss how memory management works with Java arrays in a future lecture.

Reviewing the Features of Methods – Arrays as Parameters

- In the last lecture we discussed method overloads, for example an overloaded *sum()* method:

```
static int sum(int x, int y) {
    return x + y;
}
static int sum(int x, int y, int z) {
    return x + y + z;
}
static int sum(int w, int x, int y, int z) {
    return w + x + y + z;
}
```

  - So far so good, if we want to sum up more arguments, we can just add overloads with more parameters.
  - But if we use an int[] as single parameter, we can sum up an arbitrary amount of ints, and we can get rid of all other overloads:

```
static int sum(int[] numbers) {
    int sum = 0;
    for (int i = 0; i < numbers.length; ++i) {
        sum += numbers[i];
    }
    return sum;
}
```

  - The only change we have to do on the caller side, is putting the individual summands into an array and pass this array to *sum()*:

```
int result = sum(23, 45, 67);
// result = 135
```

→

```
int[] numbers = new int[]{23, 45, 67}
int result = sum(numbers);
// result = 135
```

→

```
int result = sum(new int[]{23, 45, 67});
// result = 135
```

  - The benefit we have with the int[]-overload, is that we can now sum up any count of summands:

```
int result = sum(new int[]{23, 45});
// result = 68
```

```
int result = sum(new int[]{23, 45, -34, 2, 4562, 12, 60, 75});
// result = 4745
```

    etc.

Review of the main()-Method and Processing of Command Line Arguments

- After we have a better understanding of arrays, we can have another look at the method *main()*:

```
public static void main(String[] args) throws Exception {
	// pass
}
```

  - *main()* accepts a *String[]*! – *args* represents/contains the command line arguments, that are passed when the program is started.

- E.g. we can process the command line arguments via *main()*'s parameter *args*:

```
public static void main(String[] args) throws Exception {
	for (int i = 0; i < args.length; ++i) {
		System.out.println("Argument: "+args[i]);
	}
}
```

  - Now we can start this program with three arguments, which will be printed to the console:

```
Terminal
NicosMBP:src nico$ java Program arg1 arg2 arg3
Argument: arg1
Argument: arg2
Argument: arg3
NicosMBP:src nico$
```

  - We can also pass no command line arguments at all, then nothing will be printed to the console:

```
Terminal
NicosMBP:src nico$ java Program
NicosMBP:src nico$
```

Reviewing the Features of Methods – variable Arity Parameters

- Java offers a syntactic simplification, if an arbitrary amount of arguments of the same type should be passed to a method.
  - To make it work for *sum()*, must be modified somewhat:

```
static int sum(int… numbers){
        int sum = 0;
        for (int i = 0; i < numbers.length; ++i) {
            sum += numbers[i];
        }
        return sum;
}
```

  - Then we can call *sum()* in a way, as if it supports multiple overloads with any number of arguments:

```
int result = sum(new int[]{23, 45});
// result = 68
```
→
```
int result = sum(23, 45);
// result = 68
```

```
int result = sum(new int[]{23, 45, -34, 2, 4562, 12, 60, 75});
// result = 4745
```
→
```
int result = sum(23, 45, -34, 2, 4562, 12, 60, 75);
// result = 4745
```

- The feature we have seen here is the variable arity parameter, *numbers* in this case:

```
static  void  doSomething (<Type>... variableArityParameter) {
    // pass
}
```

  - We just have to add the … separator, the so called ellipsis, behind the type of the parameter to make it a variable arity parameter.
    - … is one token and is not allowed to contain whitespaces! But … can be surrounded with whitespaces.
  - Mind, that the type of this parameter is no array type! Instead a variable list of arguments is automatically aggregated into an array.

Reviewing the Features of Methods – Restrictions of variable Arity Parameters

- A variable arity parameter must be the very last parameter of a method!

```
static int sum(String text, int… numbers) {
        // pass
}
```

```
// Won't compile with various error messages
static int sum(int… numbers, String text) {
        // pass
}
```

Good to know
Java allows to define the method *main()* with a variable arity parameter:

```
public static void main(String... args) throws Exception {
        for (int i = 0; i < args.length; ++i) {
                System.out.println("Argument: "+args[i]);
        }
}
```

That makes sense, because is actually called this way from the command line.

- All arguments to be aggregated into a variable arity parameter must be of the same type.
  - That should be clear, because the arguments are aggregated into an array having equal element-types by definition.

- Methods cannot be overloaded one with a variable arity parameter and one with an array of the same type.

```
static int sum(int[] numbers) {
        // pass
}
// Won't compile: cannot declare both sum(int...) and sum(int[]) in Program
static int sum(int... numbers) {
        // pass
}
```

  - That makes sense, because how should the compiler tell those signatures apart to select the proper overload in, e.g., this case:

```
int result = sum(new int[]{23, 45, -34, 2, 4562, 12, 60, 75});
```

  - If no arguments are passed to variable arity parameters, the parameter represents an empty array.

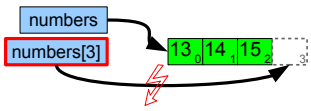
```
int result = sum();
// result = 0
```

Enhanced for Loops, aka for each Loops

- The iteration of arrays is a very common operation we have to do in a program.

- After some practice, esp. the for loop iterating array is simple to use, but it is also simple to introduce nasty bugs.
  - By far the most frequently found bug is excessing an arrays' bounds by accessing a non-existing element using too small/large indexes:

```
int[] numbers = {13, 14, 15};
// oops, off-by-one
for (int i = 0; i < 4; ++i){
        System.out.println(numbers[i]);
}
```




```
Terminal
NicosMBP:src nico$ java Program
12
13
14
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 3
        at Program.main(Program.java:21)
NicosMBP:src nico$
```

  - This kind of error is called off-by-one error, because we are exactly one index-counter off the valid bounds of an array.
  - Programming language designers came to the conclusion, that looping using indexes and lengths is dangerous.
  - Therefor an *ArrayIndexOutOfBoundException* is thrown, which immediately stops the program from execution.

- Java provides a special variant of the for loop, the enhanced for loop, which doesn't use an arrays' indexes or length:

```
int[] numbers = {13, 14, 15};
// oops, off-by-one
for (int i = 0; i < 4; ++i){
        System.out.println(numbers[i]);
}
```

```
int[] numbers = {13, 14, 15};
// the enhanced for loop:
for (int item : numbers) {
        System.out.println(item);
}
```

  - The enhanced for loop are colloquially called for each loops. This term will also used in this course, esp. because of its brevity.

for Loops vs for each Loops

```
int[] numbers = {13, 14, 15};
for (int i = 0; i < numbers.length; ++i) {
        System.out.println(numbers[i]);
}
```



```
int[] numbers = {13, 14, 15};
for (int item : numbers) {
        System.out.println(item);
}
```

- The for each loop's head is simpler, we only have the declaration of the item, then a colon and then the iterated array.

- The core idea of for each is, that it directly retrieves the individual elements of the iterated array:
  - Mind, that item is not an index! It's the element of the current iteration! The idea shows better, when we use an array not holding ints:

```
String[] sentence = {"The", "catcher", "in", "the", "rye"};
// iterates all String-elements in sentence
for (String word : sentence) {
        System.out.println(item);
}
```

Good to know
When using for each loops to iterate arrays, the individual elements gotten for each individual iteration are usually called items. The term item was chosen here, to underscore for each's idea to primarily get the items of an array, rather than using the loop for repetitive control flow.

- Obviously, for each doesn't use indexes and lengths to work, so it is much simpler to use than the for loop.
  - On the other hand for each is also less powerful than the for loop. It is more suitable for array iteration than for control flow.

- Commonalities:
  - for each loops can also be controlled with the keywords break and continue.
  - The for each loop is just a variant of the for loop, it doesn't have a representation different from for loops in flowcharts or NSDs.

ArrayLists

- Arrays are very powerful and important means of programming, not only in Java.
- Objects like arrays, which hold/manage other objects, i.e. its elements are called collections in Java.
- Java's arrays have one relevant downside: after an array was created, we can neither add nor remove elements!
  - It should be said, that this feature of array-immutability, is also the key of arrays' high performance in Java.
- Of course Java provides a solution for this dilemma! We can use another collection type, the class *java.util.ArrayList*.

```
int[] numbers = {13, 14, 15};
for (int i = 0; i < numbers.length; ++i) {
        int number = numbers[i];
        System.out.println(number);
}
```



```
import java.util.ArrayList;

ArrayList<Integer> numbers = new ArrayList<Integer>();
numbers.add(13);
numbers.add(14);
numbers.add(15);
for (int i = 0; i < numbers.size(); ++i) {
        int number = numbers.get(i);
        System.out.println(number);
}
```

- Yes, there are some significant differences comparing code using an int[] and an *ArrayList<Integer>*.

31

- Here we use *ArrayList* in Java, an alternative would be *Vector*, but *Vector* is a (object-) synchronized collection, which is more inefficient than the unsynchronized *ArrayList*. – *ArrayList* should be our default, until synchronization is needed. But if synchronization is needed, it is better to use Java's simple factory *Collections.synchronizedList()* instead of the old-fashioned *Vector*.
  - *Vector*, which was introduced with Java 1, was synchronized from the start, because people wanted to force multithreaded programming from the start and *Vector* should then be a functional default collection for multithreaded programming.

Differences between using Arrays and java.util.ArrayLists (in short ArrayLists) – Part 1

```
int[] numbers = ...
```

```
ArrayList<Integer> numbers = ...
```

- *numbers* is now of type *ArrayList<Integer>* instead of int[], we discuss syntactical specialties of *ArrayList* in short.
  - But, we can directly spot, that the *ArrayList* must be specified with *Integer* in angle brackets instead of the familiar array syntax int[].

```
int[] numbers = {13, 14, 15};
```

```
ArrayList<Integer> numbers = new ArrayList<Integer>();
numbers.add(13);
numbers.add(14);
numbers.add(15);
```

- *ArrayList*s must always be created instantiated with new, they do not support an initializer syntax like array do.
  - However, on the other hand, we do not need to specify a length/count of elements to be stored in the *ArrayList*.
  - We have to add each element we want to store in the *ArrayList* individually, by calling *ArrayList*'s method *add()* for each int value.

```
for (int i = 0; i < numbers.length; ++i) {
```

```
for (int i = 0; i < numbers.size(); ++i) {
```

- We get the current count of elements stored in an *ArrayList* by calling the method *size()* (on int[], we'd to use the field *length*).

```
        int number = numbers[i];
        System.out.println(number);
}
```

```
        int number = numbers.get(i);
        System.out.println(number);
}
```

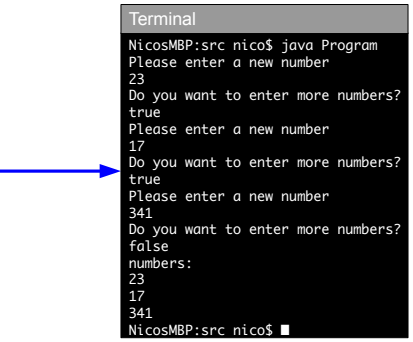
- We can retrieve individual elements by their indexes in an *ArrayList*, when we call *get()* and pass the respective index.
  - The element at that index is returned to the caller of *get()*.
  - Using *get()* almost like using the []-operator with an index to get an element of an array.

Differences between using Arrays and ArrayLists – Part 2

- However, we do not need to specify a length/count of elements to be stored in the *ArrayList* on instantiation/in advance.
  - *ArrayList* automatically adjusts its size, but arrays are fixed in size.
  - Good, if we do not know the possible count of elements to be stored in the array.

- E.g. we can let the *ArrayList* gradually grow: if the user wants to enter more numbers, the *ArrayList* just grows:

```
// The element count doesn't matter, numbers just grows during the iteration dynamically:
ArrayList<Integer> numbers = new ArrayList<Integer>();
boolean addMoreNumbers;
do {
        int enteredNumber = promptAndReadNumber("Please enter a new number");
        numbers.add(enteredNumber);
        addMoreNumbers = promptAndReadBoolean("Do you want to enter more numbers?");
} while (addMoreNumbers);
// Print elements at console:
System.out.println("numbers:");
for (int i = 0; i < numbers.size(); ++i) {
        System.out.println(numbers.get(i));
}
```



Differences between using Arrays and ArrayLists – Part 3

- Besides adding new elements to an *ArrayList*, we can also remove elements at a specific index.
  - Therefore we use *ArrayList*'s method *remove()*, let the user decide:

```
ArrayList<Integer> numbers = new ArrayList<Integer>();
boolean addMoreNumbers;
do {
        int enteredNumber = promptAndReadNumber("Please enter a new number");
        numbers.add(enteredNumber);
        addMoreNumbers = promptAndReadBoolean("Do you want to enter more numbers?");
} while (addMoreNumbers);
// Remove an element at a specific index in numbers:
int indexForRemoval = promptAndReadNumber("Remove an element at index (max.: "+(numbers.size() - 1)+"):");
System.out.println("Element at index "+i+" will be removed.");
numbers.remove(indexForRemoval);
// Print elements at console:
for (int i = 0; i < numbers.size(); ++i) {
        System.out.println(numbers.get(i));
}
```

Terminal

```
NicosMBP:src nico$ java Program
Please enter a new number
23
Do you want to enter more numbers?
true
Please enter a new number
5
Do you want to enter more numbers?
true
Please enter a new number
90
Do you want to enter more numbers?
false
Remove element at index (max.: 2):
Element at index 1 will be removed.
23
90
NicosMBP:src nico$
```

- Apart from *add()*, *remove()* is another method for managing an *ArrayList*s elements.
  - Other notable "management" methods: *clear()* to remove all elements and *set()* to replace an element at a specific index.
  - *ArrayList* provides many other powerful methods, which one usually learns to use and appreciate after a while of practice.

Excessing ArrayList Bounds

- If we excess an *ArrayList*'s bounds, i.e. access elements on indexes beyond its size, we get a run time error:

```
ArrayList<Integer> numbers = new ArrayList<Integer>();
numbers.add(13);
numbers.add(14);
numbers.add(15);
for (int i = 0; i < 4; ++i) {
        System.out.println(numbers.get(i));
}
```

```
Terminal
NicosMBP:src nico$ java Program
13
14
15
Exception in thread "main"java.lang.IndexOutOfBoundsException: Index: 3, Size: 3
      at java.base/java.util.ArrayList.rangeCheck(ArrayList.java:638)
      at java.base/java.util.ArrayList.get(ArrayList.java:414)
      at Program.main(Program.java:28)
NicosMBP:src nico$
```

  - Similar to arrays, an *Exception* is thrown, in this case an *IndexOutOfBoundsException*.

- Off-by-one errors can be avoided by defensive programing.
  - Either by using the *ArrayList*'s *size()* instead of the magic number 4:

```
for (int i = 0; i < numbers.size(); ++i) {
        System.out.println(numbers.get(i));
}
```

  - Or we can use a for each loop with the *ArrayList*:

```
for (int item : numbers) {
        System.out.println(item);
}
```

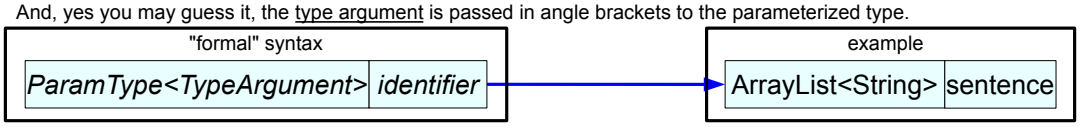
Good to know

Usually for each loops are the preferred way to iterate arrays as well as *ArrayLists*. However, during for each-looping an *ArrayList*, the *ArrayList* must not be modified by changing its element count in any way! E.g. the methods *add()* and *remove()* must not be called from within a for each loop! If we don't obey this rule, we'll get a run time error, more specifically a *ConcurrentModificationException*.

```
for (int item : numbers) {
        if (15 == item) {
        // No!
            numbers.add(16);
        }
}
```

ArrayLists as parameterized Type

- All right, up to now we have managed to ignore the syntactical specialties of *ArrayList*, i.e. the usage of angle brackets.
- *ArrayList* is a so called parameterized type.
- A parameterized type is a type, which is not yet complete. It needs another type as argument to make it complete.
  - And, yes you may guess it, the type argument is passed in angle brackets to the parameterized type.



- There is a difference concerning the types, which are passed as *TypeArgument*: we have to pass reference types.
  - Because primitive types are no reference types, Java provides so called wrapper types, one wrapper type for each primitive type.
  - Here a selection of the most commonly used primitive types associated to their wrapper types:

Primitive Type	Wrapper (reference) Type	Example
int	*Integer*	*ArrayList<Integer>*
double	*Double*	*ArrayList<Double>*
boolean	*Boolean*	*ArrayList<Boolean>*
--	*String*	*ArrayList<String>*

Good to know
The syntax *ArrayList<Integer>* is usually read "*ArrayList* of *Integer*".

  - Notice, that *String* is already a reference type and can be used directly as *TypeArgument*.

Mixed Thoughts about Arrays and ArrayLists

- Arrays need less memory than *ArrayLists* and creation/instantiation of arrays is much faster than that of *ArrayLists*.
  - So arrays are more efficient than *ArrayLists*, this is because Java's arrays have a closer connection to memory than *ArrayLists*.

- Commonalities:
  - Operations, which deal with invalid (too large) indexes lead to run time errors (*Exceptions*).
  - Arrays as well as *ArrayLists* can be iterated with for each loops, which helps avoiding off-by-one errors.
  - Arrays and *ArrayLists* can equally used as parameters and return types.

```
static int[] getNumbersFromUser() {
        final int nNumbers = 10;
        int[] numbers = new int[nNumbers];
        for (int i = 0; i < nNumbers; ++i) {
                numbers[i] = readNumberFromConsole();
        }
        return numbers;
}

int[] numbers = getNumbersFromUser();
```

```
static int sum(int[] numbers) {
        int sum = 0;
        for (int item : numbers) {
            sum += item;
        }
        return sum;
}

int result = sum(numbers);
```

```
static ArrayList<Integer> getNumbersFromUser() {
        final int nNumbers = 10;
        ArrayList<Integer> numbers = new ArrayList<Integer>();
        for (int i = 0; i < nNumbers; ++i) {
                numbers.add(readNumberFromConsole());
        }
        return numbers;
}

ArrayList<Integer> numbers = getNumbersFromUser();
```

```
static int sum(ArrayList<Integer> numbers) {
        int sum = 0;
        for (int item : numbers) {
            sum += item;
        }
        return sum;
}

int result = sum(numbers);
```

Advanced Operations on Arrays with Streams

- Java also provides another means to operate on arrays, *ArrayList*s and other so called "collections": *Streams*
  - With *Streams* we can proverbial streamline code operating on those collections without using any loops:

```
int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
int sum = 0;
for (int i = 0; i < numbers.length; ++i) {
      sum += numbers[i];
}
```

```
int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
int sum = Arrays.stream(numbers).sum();
```

```
int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
int sum = IntStream.of(numbers).sum();
```

- *Streams* abstract collections as a stream of items, to which we can pipeline further operations on this *Stream*:

```
int nNumbers = 5;
int[] numbers = new int[nNumbers];
for (int i = 0; i < nNumbers; ++i) {
      numbers[i] = readNumberFromConsole();
}
```

```
int nNumbers = 5;
int[] numbers
      = IntStream
          .generate(() -> readNumberFromConsole()) // Generate int Stream
          .limit(nNumbers)                         // Get 5 ints from the Stream
          .toArray();                              // Create int[] from 5 ints
```

- Some operations of the companion class *Arrays* can also be expressed with *Streams*:

```
int[] numbers = { 8, 12, 3, 36, 2, 6, 77, 8, 9 };
Arrays.sort(numbers);
```

```
int[] numbers = { 8, 12, 3, 36, 2, 6, 77, 8, 9 };
int[] sortedNumbers = Arrays.stream(numbers).sorted().toArray();
```

  - Methods called on *Streams* often create other *Streams*, on which further methods can be called.
  - This is called a fluent programming style, because method calls are done in chains and data seems to flow through the calls.

- *Streams* are very powerful and we will discuss them in depth in future lectures.

Strings

- Virtually, in computing we only have integral, floaty and textual data.

```
// A String:
String aString = "Frank";
```

- Java represents textual data with the type *String* and individual letters with the type char.
  - *String*s are just strings of chars, hence the name.

- The type *String* is not written in blue color. This is because *String* is no primitive type.
  - Nevertheless, the type *String* can be used mostly like any other primitive type in Java.

- *String*s are generally very important for programming and are esp. very important in Java.
  - An *String[]* is part of the signature of *main()*, the paramount method of a Java program.
  - Each object of reference type offers the method *toString()*, that returns a *String*-representation of an object.
  - *String*s have integrated syntactic support, they are represented by *String* literals.

- On the next slides we will discuss *String* once again and apply the knowledge we have gained meanwhile.

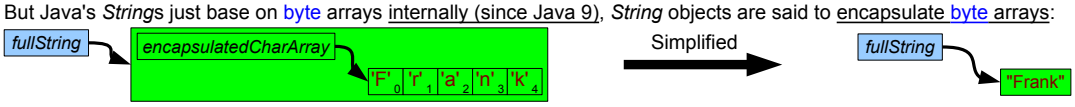
Advanced: Java's Strings vs C/C++ Cstrings

- In opposite to cstrings in C/C++, there is no tight connection between *String* and "char arrays" or even pointers!
  - Cstrings are essentially char[] with a 0-termination as last character, this tricky representation is not used in Java.
  - Nevertheless Java provides *String* methods to interact with a *String*'s contained chars, as if the *String* was an array.
  - In Java we can also create a *String* from a char[].

```
// C++: 0-terminated char arrays _are_ cstrings:
char astring[] = {'F', 'r', 'a', 'n', 'k', 0};
std::cout<<astring<<std::endl;
// >Frank
```

```
// Java: char arrays and Strings have a certain relationship:
char[] someLetters = {'F', 'r', 'a', 'n', 'k'};
String fullString = new String(someLetters);
System.out.println(fullString);
// >Frank
```

  - But Java's *String*s just base on byte arrays internally (since Java 9), *String* objects are said to encapsulate byte arrays:



- Java's *String*s support Unicode from the scratch. No tedious handling like that in C/C++ is required!

- The usage of *String*s in Java is very intuitive and very much simpler than handling cstrings in C/C++.
  - *Java's String*s are always created on the heap, but garbage collections helps avoiding memory leaks.
  - Similar to cstrings in C/C++, Java's *String*s cannot be modified after creation, but new *String*s can be created easily.
  - However, C++' *std::string* is mutable, Java's *String* is not.
  - For Java's internal handling of *String*s a lot of effort was put into optimization, because *String*s are the most often used objects!

String Literals with Escape Sequences – Part 1

```
String hello = "Hello, World!";
```

- *String* literals have to be written as a text enclosed in double quotes!
  - The examples of this course highlight *String* literals in brown color.

- Escape sequences are symbol-sequences in a *String* literal with a special meaning.

- E.g. within a *String* literal we can't use the "-char directly, it must be "escaped".
  - Why is it impossible? Because the compiler has to know the limits of a *String* literal.

```
String text = "Wen"dy"; // Compiler in trouble: is that "Wen" or "Wen"dy"?
```

  - In Java there exists a set of escape sequences, one of them, \"" solves this problem:

```
String text = "Wen\"dy" // OK, just use the escape sequence \". "Wen"dy"
System.out.println(text);
```

- A //-comment within a *String*-literal becomes part of the *String*-literal:

```
String aString[] = "Hello, World! // comment";
```

Good to know
What does "escape" mean, when we talk about strings? An "escape-character" tells the interpreter "Notice, next, there will be a character, that does not belong to the "ordinary character-/typeset", because it is a "control character".

Terminal

```
NicosMBP:src nico$ java Program
Wend"dy
NicosMBP:src nico$
```

String Literals with Escape Sequences – Part 2

- Some "letters" have just no "human readable representation", escape sequences will help us here as well.
  - Such "letters" are often so called control codes.
  - Control codes can usually not be "literally" entered via the keyboard. Let's examine some of them:

```
// Inserting a newline into a string literal (\r\n means "carriage return" and "newline"):
System.out.println("Hello,\r\nWorld"); // \r\n will add a blank line below "Hello," on the console:
// >Hello,
// >World
```

```
// Inserting a tab into a string literal (\t):
System.out.println("a\tb"); // \t will add a tab between a and b on the console:
// >a    b
```

```
// To avoid misinterpretation of \ as a character, it must be escaped as well, so have \\:
System.out.println("Hello\\World"); // \\ will add a backslash between "Hello" and "World":
// >Hello\World
```

Good to know
String literals will carry a lot of backslashes, if there are many characters to escape, e.g. for Windows file paths: "C:\\foo\\bar\\text.txt" or regular expressions: "[^\"]*\"([^\"]*)\"\\s*\\(([^\\(]*)\\)".
This visual effect in the code is called the "leaning toothpick syndrome".

- Future Java versions may provide raw string literals, which allow leaving away a lot of backslashes, improving readability.

Strings have some Aspects of Arrays

- We will now have a closer look on *String*s analyzing some of their methods, which act "array like":

- Read/iterate the chars of a *String*:

```
// Print each letter/char of a String to the console:
String fullString = "Frank";
for (int i = 0; i < fullString.length(); ++i) {
        char letter = fullString.charAt(i);
        System.out.println();
}
```

  - => We use the method *String.length()* like we've used arrays' field length and *String.charAt()* like we've used array's []-operator.

```
// Get a copy of the char array representing the letters of a String and just print/iterate its letters/chars:
char[] letters = fullString.toCharArray(); // Get a copy of fullString's encapsulated char[].
for (char letter : letters) {
        System.out.println(letter);
}
```

  - => We use *String.toCharArray()* to create a copy of the *String*, then we can iterate this char[] with an ordinary for each loop.
    - => We are really handling the *String*'s letters as chars of a char[] in this example.

- However: *String* doesn't provide any methods to modify the encapsulated char[]!
  - There is no method like "setCharAt()".
  - *String.toCharArray()* returns a copy of a *String*'s encapsulated array, which could be changed, but won't change the original *String*.
  - => *String*s are said to be immutable!

String Concatenation – Part 1

- Now we'll discuss another example of *String*'s immutability with *String* concatenation.

- Well, often it is required to compose *String*s by concatenating other *String*s:

```
String allNumbers = null;
// Concatenate multiple Strings to allNumbers:
for (int i = 0; i < 3; ++i) {
    allNumbers += "next number: ";
    allNumbers += promptAndReadNumber();
    allNumbers += ", ";
}
System.out.println(allNumbers);
```

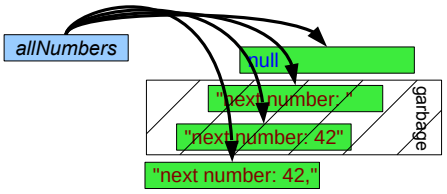
  - From a programmer's perspective, the variable *allNumbers* is modified for multiple times, because += is used.
  - But the *String* value/s is/are never changed!
  - Instead each concatenation operation created a new *String* and overwrites the *String* value referenced by *allNumbers*.
  - The formerly referenced *String* values, will be removed from memory by Java's garbage collector (gc).

- Let's understand what's going on here.

String Concatenation – Part 2

- Let's just analyze the code of only one iteration:

```
String allNumbers = null;                    // 1 String reference of value null
// Concatenate further Strings:
allNumbers += "next number: ";               // 2 (creates new String object)
allNumbers += promptAndReadNumber();         // 3 (creates new String object)
allNumbers += ", ";                          // 4 (creates new String object)
```

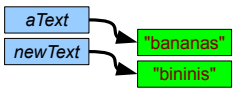


- We analyzed only one iteration but created many *String* objects and created a lot of garbage in the memory.
  - *allNumbers* just references the latest result of the repeated concatenation.
  - This is the prize we have to pay for having immutable *String* objects.

- There exist many *String* methods, which seem to modify Strings, but rather create/return new *String* objects:
  - E.g.: *replace()*, *toLowerCase()*, *toUpperCase()* and *trim()*.
  - This is esp. irritating for beginners:

```
String aText = "bananas";
aText.replace('a', 'i');       // A typical beginner's error.
// aText = "bananas" Oops! replace() didn't modify aText, it had no side effect!

String newText = aText.replace('a', 'i');
// newText = "bininis" Correct! replace() created a new String, which is assigned to newText!
```



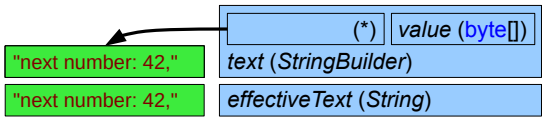
- Is there a better, i.e. more efficient way, to deal with *String*s in this concern? – Yes, with *StringBuilder*s!

- It should be said that Java's gc is highly optimized to care for temporary "particle" objects like those remainders of *String* concatenations.

Strings – StringBuilder

- In Java we can improve that situation significantly with the type *StringBuilder*:

```
StringBuilder text = new StringBuilder();       // 1 (new StringBuilder object)
// Build the text:
text.append("next number: ");                   // 2 (side effect)
text.append(promptAndReadNumber());             // 3 (side effect)
text.append(", ");                              // 4 (side effect)
// Materialize the text into a new String object:
String effectiveText = text.toString();         // new String object
```



- In opposite to *String*, the type *StringBuilder* maintains an internal buffer of characters (a byte[], e.g. named *value*).
  - The effective text, after the text was built (e.g. texts were appended and removed), is retrieved with the method *toString()*.
  - Mind that *StringBuilder*'s methods rather have side-effects on *text* than returning new *StringBuilder* or *String* objects!
  - *toString()* copies the internal buffer into a new *String* object and returns that *String.toString()* "materializes" the internal buffer.
  - I.e. following operations on *text* won't affect the content of the materialized object *effectiveText*!

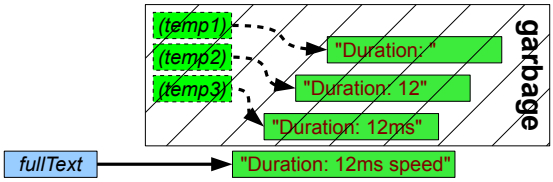
- Besides *StringBuilder* there also exists the type *StringBuffer*.
  - Both types have the same functionality, but *StringBuilder* is faster, because it doesn't synchronize concurrent modification.
  - For the time being we'll follow the advice of the JDK documentation and prefer the type *StringBuilder*.

String Concatenation and Joining

- Let's inspect the situation, when we +-concat multiple *String*s in one expression (without loop and without +=):

```
// Concatenate multiple Strings in one expression:
String testName = "speed";
String fullText = "Duration: " + (3 * 4) + "ms " + testName;
```

- Such an expression potentially creates some temporary *String*s before the effective *String* is created:



Good to know
Since Java 9 the +-operator for concatenation is transformed into invokedynamic calls on one side and static calls using *StringConcatFactory* on the other side. This separation allows to defer optimization to the run time.

- In text processing we often need to jam texts together, which are separated by a specific delimiter.
  - This is a simple task for a *StringBuilder*:

```
String[] texts = {"one", "two", "three"};
StringBuilder sb = new StringBuilder();
for (int i = 0; i < texts.length; ++i) {
      sb.append(texts[i]);
      if (i != texts.length - 1) { // Append last comma, only if the current text was not the last one.
      sb.append(", ");
      }
}
```

  - But it can done very much simpler w/o loops and special handling of last items, when we use the method *String.join()*:
    - *String.join()* accepts a delimiter and a *String[]*.

```
String fullString = String.join(", ", texts);
```



```
// Mixed concatenation and addition with primitive types can lead to
// surprises because of type conversion to String and associativity:
String aText = "Test" + 3 + 5;
// aText = Test 35
// Solution: use parentheses to direct associativity to the addition
String aText2 = "Test" + (3 + 5);
// aText2 = Test 8
```

Put the Elements of an Array into a String

- Sometimes, we just need a "stringified" version of an array, instead of joining data into a single *String*.
  - Alas, arrays, even *String[]* do not have an "intuitive way" to be converted into a single *String*:

```
String[] texts = {"one", "two", "three"};
System.out.println("The texts: "+texts);
// >The texts: [Ljava.lang.String;@6d03e736
```

```
int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
System.out.println("The numbers:"+numbers);
// >The numbers: [I@6d03e736
```

  - Java's *String* representation of arrays is a bit odd and will be discussed in a future lecture, however there are better solutions.

- Constructing a single *String* using a loop and a *StringBuilder* is a performant but "wordy" solution:

```
StringBuilder sb = new StringBuilder();
for (int i = 0; i < texts.length; ++i) {
        sb.append(texts[i]);
        if (i != texts.length - 1) {
            sb.append(" ");
        }
}
System.out.println("The texts: "+sb);
// >The texts: one two three
```

```
StringBuilder sb = new StringBuilder();
for (int i = 0; i < numbers.length; ++i) {
        sb.append(numbers[i]);
        if (i != numbers.length - 1) {.
            sb.append(" ");
        }
}
System.out.println("The numbers: "+sb);
// >The numbers: 1 2 3 4 5 6 7 8 9
```

- Alternatively, we can use the companion class *java.util.Arrays'* method *toString()*:

```
System.out.println("The texts: "+Arrays.toString(texts));
// >The texts: [one, two, three]
```

```
System.out.println("The numbers: "+Arrays.toString(numbers));
// >The numbers: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

  - *java.util.Arrays.toString()* provides many overloads, which accept pretty all kinds of array types like int[] and double[].

String Formatting – Part 1

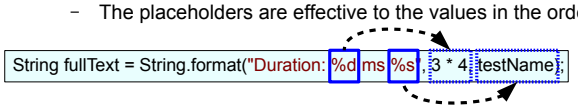
- Besides simple concatenation of *String*s we can also format *String*s easily.
  - Formatting is useful, if we want to guarantee a special and reusable look of a text with minimal effort.
  - *String* formatting is only about formatting of bare text data, it has nothing to do with font styles like "bold" or "underlined"!
- To format *String*s, we will use the method *String.format()*.
  - *String.format()* awaits a *String*, which defines the format of the formatted *String*, and the values to be formatted as *String*:

```
String fullText = "Duration: " + (3 * 4) + "ms " + testName;
// fullText = "Duration: 12ms speed-test"
```

→

```
String fullText = String.format("Duration: %dms %s", 3 * 4, testName);
// fullText = "Duration: 12ms speed-test"
```

- The format-*String* works like a template, which describes the result *String* with placeholders, which are replaced by values.
  - The placeholders in the format *String* template are denoted by the **%**-character and a so called format specifier.
  - The placeholders are effective to the values in the order they are written: **%d** to 3 * 4 and **%s** to *testName*:



Good to know
Java's format Strings correspond to the rules of Unix' *sprintf()*, not the rules of C's *sprintf()*. POSIX-conformant terminals provide the command **printf**.
The general rule for placeholders is
%[argument_index$][flags][width][.precision]conversion

  - Here we have two placeholders **%d** and **%s**, which address the arguments 3 * 4 and *testName* respectively.
  - Conversion **d**: **%d** tells *String.format()* to handle "its" value (3 * 4) as decimal integer and **s**: **%s** converts to a *String* (*testName*).
- *String.format()* can principally deal with an unlimited amount of placeholders and can replace an unlimited amount of values.
  - But, this would mean that *String.format()* deals with an unlimited amount of argument, how can that work?

String Formatting – Part 2

```
String fullText = String.format("Duration: %dms %s", 3 * 4, testName);
```

- Here we call *String.format()* with three arguments, which can be split into the format as first argument and a list of values.
  - The signature of *String.format()* looks like this:

```
public static String format(String format, Object... args);
String fullText = String.format("Duration: %dms %s", 3 * 4, testName);
```

  - As can be seen the parameter *format* stores the format string and the variable arity parameter *args* stores the remaining arguments.

- The Java compiler automatically aggregates all arguments after *format* into an *Object[]* before passing it to *String.format()*.
  - The placeholders are then simply applied on the *format* along the increasing indexes of *args*.
  - *String.format()* also allows explicit index-based access to the replacing values with special format specifiers:

```
String fullText = String.format("Second: %2$d, third: %3$d, first: %1$d", 1, 2, 3);
// fullText = Second: 2, third: 3, first: 1
```

  - Argument indexes like **2$** in the format specifier **%2$**d allow accessing specific values by index + 1 (so the **$**-index is not 0-based!).
  - It allows to break out of the need to write the values in the order of the format-*String*:

```
String fullText = String.format("Second: %2$d, third: %3$d, first: %1$d", 1, 2, 3);
```

  - And we can also replace the very same value for multiple times, when we use index-based format specifiers:

```
String fullText = String.format("Second: %2$d, third: %3$d, second: %2$d", 1, 2, 3);
// fullText = Second: 2, third: 3, second: 2
```

String Formatting – Part 3

- For *String* formatting we can use a lot of conversions in format specifiers. Here we can only discuss some of them.
  - Principally, we could use **%s** always, it just converts its value to a *String*, but more specific conversions allow more control.

- Formatting floating point values with the conversion **f** as in **%f**. Example: output a floating point value rounded to two digits:

```
String fullText = String.format("The result is %.2f", 2.666);
// fullText = "The result is 2.67"
```

  - We can specify a precision of **.2** in **%.2f**, it does the trick and performs the rounding to two digits.

- Formatting date values with the conversion **t** as in **%t**. Example: output the complete date information of "now".

```
String fullText = String.format("very now: %tc", new Date());
// fullText = "very now: Sa Jul 15 09:56:26 MESZ 2017"
```

  - The **t** conversion always needs a suffix. – In this case the suffix **c** in **%tc** means, that the complete date value will be formatted.
  - The suffixes **Y**, **m** and **d** allow us to format the 4-digit year, the two-digit month and the two-digit month's day of a *Date* value:

```
String fullText = String.format("today: %1$tY-%1$tm-%1$td", new Date());
// fullText = "today: 2017-07-15"
```

    - Mind that we had to use index-based access to apply the format specifiers to the same value to be formatted.
  - Format templates provide a shortcut to apply yet another formatting to the last value but on a different position in the template:

```
String fullText = String.format("today: %1$tY-%<tm-%<td", new Date());
// fullText = "today: 2017-07-15"
```

    - As can be seen, we use the **<** character to refer to the 1st value to be formatted for three times in the template.
    - This shortcut seems ridiculous, but it can avoid bugs.

String Formatting – Part 4

- There also exist format specifiers with control function, e.g. **%%** and **%n**:
  - **%%** just outputs the %-character, it works similar to escaping the backslash in *String* literals ("\\"):

```
String fullText = String.format("weight loss: %.2f %%", 5.64);
// fullText = "weight loss: 5.64 %"
```

  - A traditional problem in text processing is, that different OS' use different ways to output newlines. **%n** solves this problem:

```
String fullText = String.format("Your score:%n%d", 8956734);
System.out.println(fullText);
// >Your score:
// >8956734
```

Good to know
When textual line endings come into concern, the world of text processing in split in two. Some OS' use line feed (LF) to encode new lines (e.g. Unix, Android and macOS), others use carriage return (CR) and LF in sequence (= CRLF [kɜˈlɪf]) (e.g. Windows and DOS). The problem is that we as programmers have to encode LF and CRLF with different control codes as escape sequences, LF as "\n" and CRLF as "\r\n". I.e. a program needs to know, under which OS it is running to use the correct escape sequence.

- An overview of format specifiers can be found here: https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Formatter.html

- Because formatting texts and outputting those, e.g. to the console is a common operation, Java provides a shortcut.
  - We can just use the method *System.out.**printf**()*, which works with the same format specifier like *String.format()* does:

```
String fullText = String.format("Your score:%n%d", 8956734);
System.out.println(fullText);
// >Your score:
// >8956734
```

→

```
System.out.printf("Your score:%n%d%n", 8956734);
// >Your score:
// >8956734
```

  - Notice: we had to use another **%n** specifier with *System.out.printf()*, because it doesn't print an extra newline like *System.out.println()* does.

String Formatting – Part 5

- We won't discuss the rules for flags and width in detail, those allow alignment and other fine control of formatting in *String*s:

```
System.out.printf("%(d", -32); // The flag ( on decimal conversions puts negative values into parentheses.
// (32)
```

- Now it is time to discuss how "formal" errors during *String* formatting are handled.
  - If a format template declares more placeholders than values being passed, a *MissingFormatArgumentException* appears at run time:

```
System.out.printf("%d %d %d", 1, 2); // Invalid! java.util.MissingFormatArgumentException: Format specifier '%d'
```

  - If a format template declares more placeholders than values being passed, the surplus values are just ignored:

```
System.out.printf("%d %d %d", 1, 2, 3, 4); // Fine!
// >1 2 3
```

  - If the type of values does not correspond to the format specifier various usually self explaining exception can appear.

```
System.out.printf("%d", "a text"); // Invalid! java.util.IllegalFormatConversionException: d != java.lang.String
```

    - Other *Exceptions*: *UnknownFormatConversionException*, *UnknownFormatFlagsException*, *IllegalFormatWidthException* or *IllegalFormatPrecisionException*

- *String.format()* uses the type *Formatter* internally to perform formatting. Using *Formatter* directly can lead to optimized code.

- Alternative ways to do formatting in Java:
  - *String*s can be formatted applying so called patterns, which can save a lot of code. See *javax.swing.text.MaskFormatter*.
  - To format messages, esp. for GUIs dealing with locales and pluralization, the type *java.text.MessageFormat* can be used together with *java.text.ChoiceFormat*.

Strings – Equality Comparison with equals()

- *Strings* cannot be compared for equality with the == operator! Example of wrong equality comparison:

```
String fstName = "Frank";
String sndName = new String(new char[]{'F', 'r', 'a', 'n', 'k'});
// Semantically wrong! The == operator compares the references of the Strings for
// identity, not the Strings' contents for equality!
if (fstName == sndName) {
      System.out.println("fstName and sndName are equal!");
} else {
      System.out.println("fstName and sndName are not equal!");
}
// >fstName and sndName are not equal! Oops!
```

- The correct way to compare *Strings* for equality is to use another method of *String*: *equals()*:

```
String fstName = "Frank";
String sndName = new String(new char[]{'F', 'r', 'a', 'n', 'k'});
// OK! Use the method equals() to compare Strings for equality!
if (fstName.equals(sndName)) {
      System.out.println("fstName and sndName are equal!");
} else {
      System.out.println("fstName and sndName are not equal!");
}
// >fstName and sndName are equal!
```

Good to know

```
// Because even String literals are genuine String objects,
// we are able to call methods directly on literals!
String fstName = "Frank";
String sndName = "Joe";
boolean result = fstName.equals(sndName);
// result = false

// We can also call equals() directly on the literal "Frank":
boolean result = "Frank".equals(sndName);
// result = false
```

  - As can be seen we call *equals()* on the lhs of the comparison and pass to rhs to *equals()*.

- To compare *Strings* for equality case-insensitively, *String*'s method *equalsIgnoreCase()* can be used.

- Most spoken languages have their own rules for sorting/comparison and the meaning of equality of texts. In such cases the simple operations offered in *String* are not sufficient to handle texts adhering to the rules of specific languages. Java offers so-called *Collator*s, which encapsulate language-specific operations on textual data. For example let's get a *Collator* for the Ukrainian language, then we have to use the *Collator*'s methods to do the comparisons instead of *String*'s methods – we only use *Strings* as a means to transport the textual data to the *Collator*:

```
// Compare Ukrainian texts:
Locale tag = Locale.forLanguageTag("uk-UA");
Collator coll = Collator.getInstance(tag);
boolean equalityComparisonResult = coll.equals(a, b);
int comparisonResult = coll.compare(a, b);
```

Strings – Order Comparison with compareTo()

- *Strings* cannot be order-compared with the operators <, >, <= or >= ! Example of wrong comparison:

```
String fstName = "Alberta";
String sndName = "Caroline";
// Syntactically wrong! The < operator cannot be used to compare Strings!
if (fstName < sndName) { // Invalid! Won't compile!
    System.out.println("fstName is less than sndName!");
} else {
    System.out.println("fstName is greater than or equals sndName!");
}
```

- The correct way to compare *Strings* is to use yet another method of *String*: *compareTo()*:

```
String fstName = "Alberta";
String sndName = "Caroline";
// OK! Use the method compareTo() to compare Strings!
if (0 > fstName.compareTo(sndName)) {
    System.out.println("fstName is less than sndName!");
} else {
    System.out.println("fstName is greater than or equals sndName!");
}
// >fstName is greater than or equals sndName!
```

  - *compareTo()* returns an int, it is 0, if the compared *Strings* are case-sensitively equal.
  - *compareTo()* returns an int, it is less than 0, if lhs is lexicographically less than rhs.
  - *compareTo()* returns an int, it is greater than 0, if lhs is lexicographically greater than rhs.

- To compare *Strings* case-insensitively, *String*'s method *compareToIgnoreCase()* can be used.

- Upper case letters are considered "less than" lower case letters.

Strings – Searching individual chars

- To search an individual char, we can use a loop to iterate over the chars of a *String*:

```
String aString = "bar"; // Search the first 'a' in aString,
char[] chars = aString.toCharArray();      // toCharArray() retrieves a copy of the content of sString as char[].
int indexOfa = -1;                         // Set the initial index to an invalid index of an array!
for (int i = 0; i < chars.length; ++i) {   // Then we iterate aString's content with a normal for loop.
      if (chars[i] == 'a') { // Just check the content of the current char...
            indexOfa = i;
            break;
      }
}
```

- However, the better approach to find a char is *String*'s method *indexOf()*, because using loops is very error prone:

```
String aString = "bar"; // Search the first 'a' in aString,
int indexOfa = aString.indexOf('a'); // Gets the index of the letter 'a'.
// indexOfa = 1
int indexOfz = aString.indexOf('z'); // There is no 'z' in aString, so the indexOfz is -1.
// indexOfz = -1
```

  - Notice, that the returned indexes are 0-based array indexes. I.e. 0 is the index of the first char, additionally -1 is the invalid index.

- Using *indexOf()* counting all occurrences of an individual char is a piece of cake:

```
String aString = "bananas";
int indexOfa = aString.indexOf('a'); // Search the first 'a' in aString.
int occurrencesOfa = 0;
while (-1 != indexOfa) { // While the indexOfa is not -1 (invalid index means "not found"):
      ++occurrencesOfa;                                  // 1. Increment occurrences.
      indexOfa = aString.indexOf('a', indexOfa + 1);     // 2. Search next 'a' and store its index.
}
```

Strings: Substrings and empty Strings

- A substring is, as the name implies, a part/portion of another string. Substrings can be "cut" with the method *substring()*:

```
String aString = "thinking";
String subString = aString.substring(4, 6); // Get the substring from the index 4 to index 6.
// subString = "ki"
```

  - Notice, that the used indexes are 0-based array indexes.
  - Another overload of *substring()* does just accept a begin-index and "cuts" a substring from this index to the *String*'s end.
  - Mind, that a substring is always a new *String* object independent from the original *String* in Java!

- Similar to individual chars we can also search a substring in another string.
  - This time we'll use another overload of the method *indexOf()*, a method, which we already know:

```
String aString = "thinking";
int indexOfki = aString.indexOf("ki"); // Search substring "ki" in aString.
// indexOfki = 4
```

- Sometimes we have to deal with empty *String*s. We can check the length of a *String* or just use the method *isEmpty()*:

```
String content = ""; // An empty string
System.out.println(content.length() == 0);
// >true
System.out.println(content.isEmpty())
// >true
```

- But empty *String*s do not evaluate to false, like in some other languages. We cannot use if directly with a *String*:

```
if (content) { // Invalid! Incompatible types: String cannot be converted to boolean
        // pass
}
```


- In Java the "cutting marks" to get a substring are based on indexes. In opposite to Java, C++' STL's *std::string* uses a begin index and a length to specify a substring's "cutting marks".

Working with individual chars – Part 1

- Apart from whole *String*s, we can also operate on and analyze individual chars.
  - These additional functionality is present as a set of useful methods in the type *Character*.
    - *Character* is the wrapper type for the primitive type char and the companion class for char.
  - As we already know, a *String* can be interpreted as an array of chars. And we can pick and analyze individual chars.

- Java represents chars with Unicode. Unicode is a standardized system, associating each character to a numeric code.
  - A subset of Unicode is the American Standard Code for Information Interchange (ASCII, [æski:]), we'll only discuss ASCII here.
  - The characters and belonging to codes can be taken from so called "ASCII tables".

- ASCII is a 7b code, it encodes at most 128 characters corresponding to the symbols on US typewriters and control codes.
  - ISO/IEC 8859 extended ASCII by one bit which doubles the amount of encodable characters.
  - ISO/IEC 8859 offers 15 sub standards that define 15 different sets of 128 extra letters matching for 15 language facets.
    - E.g. ISO/IEC 8859-5 adds Cyrillic letters and ISO/IEC 8859-8 add Hebrew letters.
    - The first 128 codes (first 7b-covered codes) cover the same character as in ASCII across all ISO/IEC 8859 sub standards.
  - Meanwhile ISO/IEC 8859 was dissolved into ISO UCS which is equivalent to Unicode and is typically encoded as UTF-8.
    - ISO/IEC 8859-1 ("Latin-1") defines the first 256 characters in Unicode.

- ISO/IEC for International Organization for Standardization/International Electrotechnical Commission
- UCS for Universal Coded Character Set

Working with individual chars – Part 2

- Here a part of the ASCII table:

ASCII Code	Symbol	ASCII	Symbol	ASCII	Symbol	ASCII	Symbol	ASCII	Symbol	ASCII	Symbol
…	…	…	**…**	75	**K**	86	**V**	102	**f**	113	**q**
48	**0**	65	**A**	76	**L**	87	**W**	103	**g**	114	**r**
49	**1**	66	**B**	77	**M**	88	**X**	104	**h**	115	**s**
50	**2**	67	**C**	78	**N**	89	**Y**	105	**i**	116	**t**
51	**3**	68	**D**	79	**O**	90	**Z**	106	**j**	117	**u**
52	**4**	69	**E**	80	**P**	…	**…**	107	**k**	118	**v**
53	**5**	70	**F**	81	**Q**	97	**a**	108	**l**	119	**w**
54	**6**	71	**G**	82	**R**	98	**b**	109	**m**	120	**x**
55	**7**	72	**H**	83	**S**	99	**c**	110	**n**	121	**y**
56	**8**	73	**I**	84	**T**	100	**d**	111	**o**	122	**z**
57	**9**	74	**J**	85	**U**	101	**e**	112	**p**	…	…

  - Digit symbols and letter symbols have increasing and adjacent ASCII codes following their lexicographic or numeric order.
    - The ASCII code of 'Q' is a smaller value than the ASCII code of 'S'. This is handy, because Q is lexicographically less then S in a dictionary.
    - The ASCII code of '1' is a smaller value than the ASCII code of '2'. This is handy, because 1 < 2 in the set of integer numbers.
  - Digit symbols and letter symbols have a gap in the ASCII table at]58, 64[.
  - Upper case letter symbols and lower case symbols have a gap in the ASCII table at]91, 96[.
  - Upper case letter symbols have smaller ASCII codes than lower case letter symbols.

Working with individual chars – Part 3

- In Java, we can get the ASCII code of a char (which represents the value of a symbol), by converting a char to an int.
  - We know that symbols with ASCII codes in [48, 57] are digits, so we can check whether a given char represents a digit symbol:

ASCII Code	Symbol
48	0
49	1
50	2
51	3
52	4
53	5
54	6
55	7
56	8
57	9

```
// Checks, whether ch is in the "range" of the ASCII codes of digits:
static boolean isDigit(char ch) {
        int asciiCode = ch; // Assign char to int variable.
        return asciiCode >= 48 && asciiCode <= 57;
}
```

```
// In Java, chars can be automatically converted
// to ints if required. In this case the usage of the
// comparison operators (>=, <=) initiates the conversion.
static boolean isDigit(char ch) {
        // ch will be converted to int implicitly
        return ch >= 48 && ch <= 57;
}
```

```
// Using isDigit() is intuitive:
System.out.println(isDigit('4'));
// >true
System.out.println(isDigit('K'));
// >false
```

Good to know
Methods, which return a logical information (true/false) about a passed argument are often called predicates. Predicates are simple to use, often simple to code and they lead to highly reusable methods.

- Knowing about the ASCII table, we could develop other predicates to analyze chars having codes in a specific range.
  - Besides *isDigit()* we could also program *isUpperCase()* or *isLowerCase()* that way.
  - But programming such methods all over again is of course an undesirable code repetition!
  - Java provides such methods in the type *Character*, which analyze chars in that way. – We don't have to write them ourselves!

```
// Using isDigit() is intuitive:
System.out.println(Character.isDigit('4'));
// >true
System.out.println(Character.isDigit('K'));
// >false
```

```
// Analyzing a char's case:
System.out.println(Character.isUpperCase('T'));
// >true
System.out.println(Character.isLowerCase('p'));
// >true
```

Strings containing Numbers

- Sometimes, textual data contains other data, which can be interpreted as data of primitive type, e.g.:

```
String stringWithNumber = "5297";
```

  - The content of *stringWithNumber* can be interpreted as a number of type int.

- How can we extract the number's value from the *String* as an int? We could scan the *String* and generate the int:

ASCII Code	Symbol
48	0
49	1
50	2
51	3
52	4
53	5
54	6
55	7
56	8
57	9

```
int result = 0;
char[] chars = stringWithNumber.toCharArray();
for (int i = 0; i < chars.length; ++i) {
        // The weight of the letter is its 1-based position in the string
        int digitWeight = chars.length - i - 1;
        // The digitValue is the ASCII code of the char minus 48 (See the ASCII table!).
        int digitValue = chars[i] - 48;
        // The digitValue multiplied with the scale of the digitWeight contributes to the result:
        result += digitValue * Math.pow(10, digitWeight);
}
// result = 5297
```

- But this solution has many downsides:
  - It doesn't work for negative numbers.
  - It only works with decimal numbers and not, e.g., with hexadecimal numbers.
  - The code uses a lot of magic numbers.
  - It will also hurt the DRY principle, because such processing of a *String* to get contained int is required quite often!

Strings Parsing – Part 1

- In the just presented code we read, interpret and process an object (*String*) and convert it into another object (int).
- The operation-chain "read-interprete-calculate-convert" is called parsing among programmers (and grammar-lawyers).
  - Java's type *Integer* provides a method to parse an integer number from a *String* (this line replaces our former code completely):

```
int result = Integer.parseInt(stringWithNumber);
// result = 5297
```

- Similar to *Integer.parseInt()*, we can use *Double.parseDouble()*, *Boolean.parseBoolean()* etc. to parse other values:

```
double doubleResult = Double.parseDouble("74.90");
// doubleResult = 74.90
long longResult = Long.parseLong("3567986");
boolean booleanResult = Boolean.parseBoolean("true");
// booleanResult = true
```

Good to know
Parsing of *String*s containing integral numbers, which exceed the limits of int will lead to overflows (then a *java.lang.NumberFormatException* will be thrown). If we are unsure, if this could be the case, we're better of using the special complex type *java.math.BigInteger* for int parsing:

```
int intValue = Integer.parseInt("9999999999999"); // Will overflow (java.lang.NumberFormatException)
java.math.BigInteger bigIntegerValue = new java.math.BigInteger("9999999999999");
```

We have a similar situation with double values, but double cannot overflow! Instead a double value can reach either the value *Double.NEGATIVE_INFINITY* or *Double.POSITIVE_INFINITY*, when its limits 4.9E-324 (*Double.MIN_VALUE*) or 1.7976931348623157E308 (*Double.MAX_VALUE*) are exceeded. We can avoid this, and deal with values beyond *Double.MIN_VALUE* and *Double.MAX_VALUE* when we use the special type *java.math.BigDecimal* for double parsing:

```
java.math.BigDecimal bigDecimalValue = new java.math.BigDecimal(veryLargeDecimalString);
```


- There exists and overload of *Integer.parseInt()*, which accepts a radix argument. So parsing hexadecimal (radix 16) or octal (radix 8) numbers is possible.

Strings Parsing – Part 2

- Parsing a *String* to a primitive value can be dangerous!
  - The problem: a text which is meant to represent a primitive value (e.g. int) could contain unexpected characters!
  - E.g. assume we try to parse an int from a *String*, but that *String* contains prosaic text instead of an int value:

```
int resultHuh = Integer.parseInt("no number"); // java.lang.NumberFormatException: For input string: "no number"
```

  - In this case, we have to deal with a *NumberFormatException*, which terminates our program immediately.
  - We'll also get a *NumberFormatException*, if we parse an int from a *String* representing a too small or too large int:

```
int tooLarge = Integer.parseInt("9999999999"); // java.lang.NumberFormatException: For input string: "9999999999"
```

  - We'll also get a *NumberFormatException*, if we parse a double from a *String* with from an unexpected language format:

```
double doubleResult = Double.parseDouble("74,90"); // java.lang.NumberFormatException: For input string: "74,90"
```

    - This time time parsing doesn't work, because the character ',' is not a valid decimal separator for Java's default language used for parsing.
    - Instead we have to use the class *NumberFormat* to define a target format for double parsing, which is based on the German language/locale.

```
// Parse a double value with a German format (, as decimal separator and . as thousands grouping separator):
NumberFormat format = NumberFormat.getInstance(Locale.GERMAN);
Number number = format.parse("74,90");
double doubleResult = number.doubleValue(); // OK!
```

- Alas, a Java programmer must exactly know, of which primitive type the content of the *String* is meant to be!
  - Java provides no way to check the content of a *String*, before parsing takes place!

Splitting and tokenizing Strings

- Java provides sophisticated means to parse a *String* using text patterns: splitting, tokenizing and regular expressions.
  - In this lecture, we'll discuss *String* splitting and *String* tokenizing. Regular expressions deserve a lecture of their own.

- *String* splitting is very simple in Java. The idea is to take a *String* and split it into an array of *String* on a certain delimiter.
  - Therefor we can use the method *String.split()*:

```
String sentence = "I say there is no darkness but ignorance";
String[] words = sentence.split(" ");
// words = ["I", "say", "there", is", "no", "darkness", "but", "ignorance"]
```

  - In a sense, *String.split()* is the counterpart of *String.join()*.
  - The argument *String* passed to *split()* can be a simple deliminator or complex regular expression.

- In Java we also have the class *StringTokenizer*, which allows to extract the tokens of a *String* individually:

```
StringTokenizer stringTokenizer = new StringTokenizer(sentence);
while (stringTokenizer.hasMoreTokens()) {
        System.out.println(stringTokenizer.nextToken());
}
```

  - The default configuration of *StringTokenizer* tokenizes the passed *String* on whitespaces.
  - => After Java's official documentation, *StringTokenizer* should no longer be used in favor of *String.split()*.

Thank you!