Programming Java

STEP IT ACADEMY

Lesson 5

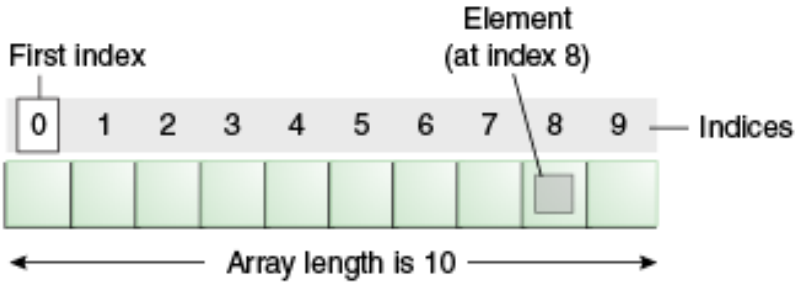
Arrays and Strings

Contents

1. Arrays

In Java, array is a finite set of elements of the same type. Java array is an object, so a variable that refers to an array is a reference type variable.

One-Dimensional Arrays



Syntax of the array reference declaration

```
type[] identifier;
or
type identifier[];
```

Example of declaring a reference to an array:

```
int[] mas; // an array type variable named mas is declared
```

or

```
int mas[];
```

Array object is created using the new operator.
The syntax of creating an array of elements:

```
new int[<number of elements>];
```

For example:

```
int[] mas; // declaration of array variable
mas = new int[3]; // creating an array
```

or

```
int[] mas = new int[3]; // an array of three elements
                        // is created and a reference
                        // to an object is stored
                        // in a variable named mas.
```

> ***NOTE!!!*** *Immediately after the creation of an array, its elements are initialized with the default values: 0 for all numeric types, a character with Unicode of 0 for char type, false for boolean and null for reference types.*

Array elements are accessed by an index. Indexing of array elements begins with 0.

For example:

```
int[] mas = new int[3];
System.out.println(mas[0]);
```

Outcome: 0

Values of array elements can be specified at creation of an array. This process is called initialization of an array.

For example:

```
int[] mas = new int[]{3, 2, 1}; // an array of three
    // elements will be created
```

```
    // first element will have the value of 3,
    // second and third elements will have the values
    // of 2 and 1, respectively.
```

> ***Note!!!*** *You cannot initialize an array and specify its size at creation simultaneously since its size is defined by the number of elements in the initialization block.*

For example:

```
int[] mas = new int[5]{3, 2, 1}; // compilation error!!!
```

There is a simplified array initialization record for arrays.

For example:

```
int[] mas = {3, 2, 1};
```

Array has a **length** property that stores the length of an array (the number of elements specified at creation of an array); this property cannot be modified.

For example:

```
int[] mas = {3, 2, 1};
mas.length = 5;         // compilation error
System.out.println(mas.length);
mas = {1, 2, 3, 4, 5}; // compilation error, can be used
                        // at declaration only.
```

Multidimensional Arrays

There are no multidimensional arrays in Java. Nested arrays are used in Java for simulating the operation with data in n-dimensional space.

Additional square brackets are added at declaration to create a multidimensional array.

Example of creating a two-dimensional array:

```
int[][] mas = new int[5][5]; // an array of 5 elements
// is created, in which each of these elements
// is an array of 5 elements.
```

Example of creating a three-dimensional array:

```
int[][][] mas = new int[5][3][3];
```

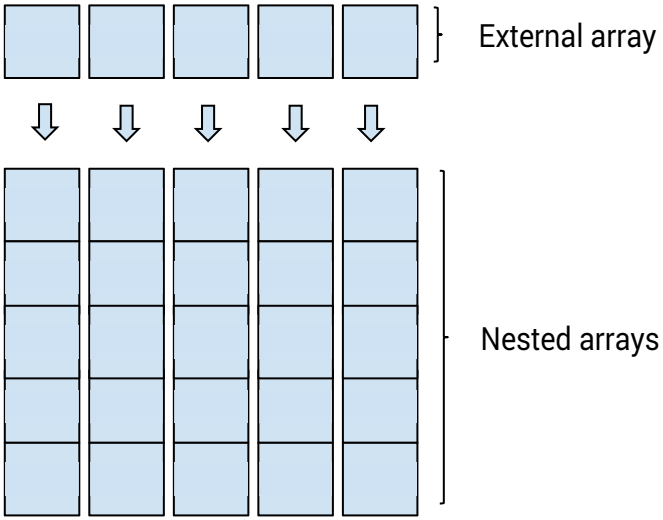


Figure. The structure of a two-dimensional array in Java

Multiple indexes can be used for accessing a multidimensional array.

Example of writing a value in a square array:*

```
int[][] mas = new int[3][3]; // an array of five
                             // elements is created
mas[2][1] = 3; // writing a value in an array
```

The syntax of multidimensional arrays provides lazy initialization.

For example:

```
int[][] mas = new int[3][]; // a one-dimensional
// array is created containing three elements,
// the values of which are null
mas[0] = new int[4];
mas[1] = new int[]{3, 3};
mas[2][2] = 7; // runtime error
```

The Use of the for-each Loop for Working with Arrays

Example of array value output:

```
int [] numbers = new int[] { 3, 2, 1 };
for (int number : numbers)
{
    System.out.print(number);
}
```

* Square array is a two-dimensional array, in which the number of elements in the nested arrays is equal to the number of elements in a source array.

For example:

```
int [] numbers = new int[] { 3, 2, 1 };
for (int number : numbers)
{
    number = 3; // the number variable can only be
    // used for reading the values of array elements
}
System.out.print(Arrays.toString(numbers));
```

Outcome: [3, 2, 1]

The java.lang.System.arraycopy() Method

The **arraycopy** method copies the values of one array to another one. This method is native* (written in C++), and because of this, it copies the values of one array to another one faster than any code written in Java. It is used in collections, for example, in java.util.ArrayList.

Syntax:

```
System.arraycopy(src, srcPos,
    dest, destPos, length)
```

where src is a source array, dest is a destination array, length is a number of copied elements from the src array, srcPos is an index in the src array, from which the copying begins, destPos is an index in the dest array, to which the elements will begin to copy.

For example:

```
int[] src = new int[]{1, 2, 3, 4, 5};
int[] dest = new int[]{5, 4, 3, 2, 1};
int length = 3;
```

* Native method has an implementation in a language other than Java.

```
int srcIndex = 1;
int destIndex= 2;
System.arraycopy(src, srcIndex, dest, destIndex, lenght);
System.out.println(Arrays.toString(dest));
```

Outcome: [5, 4, 2, 3, 4]

Note: *An exception will be thrown if there is no specified number of elements in the source array or destination array.*

The java.util.Arrays Class

The Arrays class contains static methods for working with arrays.

- toString method converts the contents of an array to a string. If an array contains objects as elements, then each toString() method will be called for each element, and the resulting strings will be concatenated*.

Example of output of the contents of an array to the console:

```
int [] mas = new int[]{1,2,3};
String content = Arrays.toString(mas);
System.out.println(content );
```

In terms of Java, an array is an object of special class. Just as all classes in Java, it is derived from the Object class, but the toString() method is not overridden in it, so when passing it to the System.out.println method, we see a standard implementation of the toString() method. Therefore, it is necessary to use the Arrays.deepToString() method for displaying multidimensional arrays in the console.

* Concatenation is a combination of multiple strings into a single string.

An example of output of the contents of a multidimensional array to the console:

```
String [][] objects = new String[3][3];
String content = Arrays.deepToString(objects);
System.out.println(content );
```

- fill method populates an array with the identical values.

For example:

```
boolean[] test1 = new boolean[3];
Arrays.fill(test1, true);
```

Example of populating array elements in the range of 2 to 5 with the value of 2:

```
int[] mas = new int[10];
int startIndex = 1;
int endIndex = 4;
Arrays.fill(mas, startIndex, endIndex, 2);
```

- sort method sorts the values of an array in an ascending order.

For example:

```
int [] mas = new int[] {3, 1, 4, 6, 2};
Arrays.sort(mas);
System.out.println(Arrays.toString(mas));
```

Outcome: [1, 2, 3, 4, 6]

If an array stores elements of reference types, it is possible to specify sorting rules in the sort method using the objects of a class that implements the Comporator interface.

Example of sorting in descending order:

```
Integer [] mas = new Integer[] {3, 1, 4, 6, 2};
Arrays.sort(mas, Collections.reverseOrder());
System.out.println(Arrays.toString(mas));
```

Outcome: [6, 4, 3, 2, 1]

- equals compares the contents of two arrays (elementwise), and returns true if these arrays have the same number of elements, and all the elements are identical; otherwise it returns false.

For example:

```
int [] mas1 = {1,2,3};
int [] mas2 = {1,2,3};
boolean isEqual = Arrays.equals(mas1, mas2);
System.out.println(isEqual);
```

- binarySearch returns an index, by which an element was found using binary search.

For example:

```
int index = Arrays.binarySearch(mas, 3);
```

- deepHashCode returns a hash code calculated on the basis of deep analysis of array elements. It can be used for arrays storing reference types.
- asList returns an immutable list (adapter object), converting array elements to list elements.

2. Strings

The String class is the most used class in Java, which is used for storing a set (array) of characters. The state of the String class object cannot be modified after it was created (class objects are immutable). Following from the class declaration: public final class String it is forbidden to inherit the subclasses from the String class.

The String class implements three interfaces: Serializable, Comparable<String>, CharSequence. The String class is essentially a wrapper over an array of characters (char [] value).

Internal arrangement of the String class:

```
public final class String{
private final char value[]; // an array of characters
private final int offset; // offset from the beginning
                          // of an array
private final int count; // number of characters in a line
private int hash; // hash code of a line
}
```

In Java, a string object can be created using the new operator of a string literal (characters limited by double quotes).

Example with a string literal:

```
String name = "John";
name = ""; // empty line
```

Example with the new operator:

```
String name = new String("John");
name = new String(); // empty line
```

When a string literal is used in the program for the first time, a new String class object is created, and a reference to it is assigned in a variable. A string object is created in a special place in memory called the string pool*. When reusing a literal, a new object is not created, and a reference to the previously created object is assigned in a variable.

For example:

```
String name1 = "John"; // an object of a new string
// is created and placed in a pool of strings.
String name2 = "John"; // a reference
// to the previously created object is assigned
// in a variable.
```

A new object is always created when creating a string object using the **new** operator.

Static Methods of the String Class

- valueOf returns a string representation of the primitive type values.

For example:

```
String value = String.valueOf(3);
value = String.valueOf(0.5);
value = String.valueOf(true);
```

- format method returns a string, in which the format specifiers** are replaced with the values of parameters in a method.

* String pool is located in a place of memory called Permanent Generation. It is necessary for reducing the number of identical objects.
** The sequence of characters begins with %

Syntax:

```
String.valueOf(<template strings>,
    <substitutional values>);
```

For example:

```
String result = String.valueOf("Once upon a time there
                                were %d silly goats", 2);
```

The following format specifiers are used in formatting:

Format specifier	Performed formatting
%a	Hexadecimal floating-point value
%b	Boolean value of an argument
%c	Character representation of an argument
%d	Decimal integer value of an argument
%h	Hash code of an argument
%e	Exponential representation of an argument
%f	Decimal floating-point value
%g	Selects a shorter representation of the two: **%e** or **%f**
%o	Octal integer value of an argument
%n	Inserts a character of a new string
%s	String representation of an argument
%t	Time and date
%x	Hexadecimal integer value of an argument
%%	Inserts a character **%**

The number of format specifiers must correspond to the number of parameters in a method. The type of values must correspond to the format specifier.

Escape Sequences

Screening with a slash is used for some characters that cannot be inserted in a string (for example, double quotes).

For example:

```
String text = "\"A character preceeded by the double
            backslash (\\) is an escape sequence.\"";
```

Escape sequence	Description
\t	Tabulation character
\b	Backspace
\n	New line
\r	Carriage return
\f	Form feed
\'	Single quote
\"	Double quote
\\	Backslash

Instance Methods

All the methods of the String class do not modify the line, in which they are called, but instead they return a reference to a new String class object.

Immutability of a string object is determined by:

- ***Language design.*** Strings are created in a special place in memory in a Java heap known as String Intern pool. When you create a new line using a string literal (not with a constructor or some library functions), a check is performed on whether an object of such a string already exists in a string pool. If it does, then a reference is returned to an already existing object; otherwise, a new string object is

created in the pool, and a reference to it is returned. If a string could be modified, the operation of a string pool would be impossible.

- ***Security.*** Strings are widely used as parameters for opening files and connecting to a database, connecting to network resources, as passwords, etc. If a string could be modified in all these cases, then it would cause serious threats to the security of the program and data.
- ***Efficiency.*** Hash code of a string is often used in Java as a key, for example, in the implementation of the java.util. HashMap associative array. Immutability of a string guarantees that its hash code will always be the same, which allows calculating the hash code once at creation of a string, which significantly speeds up the work with string if used frequently.
- charAt returns a character from a string for an index.
- concat returns concatenation (combination of two strings into one).

For example:

```
String firstName= "John";
String lastName= "Bull";
// similarly
String fullname = fullname = cat.concat(name);
//firstName+ " " + lastName;
System.out.println(fullname);
```

- length returns the number of characters in a string.

- isEmpty returns true if a string does not contain characters; otherwise returns false. Works faster than the previous one.
- charAt returns a character from a string for an index.

For example:

```
String testString = "test";
char myChar = testString.charAt(3);
System.out.println(myChar);
```

Outcome: t

- contains returns true if a string contains at least one match with the compared string.

For example:

```
String testString = "testing";
boolean test = testString.contains("st"));
System.out.println(test);
```

Outcome: true

- startsWith returns true if a string begins with the searched character or string.

For example:

```
String str1 = "Star Wars";
boolean test = str1.startsWith("Star");
System.out.println(test);
```

Outcome: true

- startsWith returns true if a string ends with the searched character or string.

For example:

```
String str1 = "java.exe";
boolean test = str1.endsWith(".exe");
System.out.println(test);
```

Outcome: true

- trim returns a string with the removed beginning and end spaces.
- toLowerCase returns the string, in which all uppercase characters of the initial string are replaced with lowercase.
- toLowerCase returns the string, in which all lowercase characters of the initial string are replaced with uppercase.
- indexOf returns the index of character, from which the first match with the searched string or character is found. The search starts from the beginning of a string. Returns -1 if there is no match.

For example:

```
String text = "defensibility";
int index = text.indexOf('e');
System.out.println(index);
```

Outcome: 1

The indexOf method allows starting the search not from the beginning but from a specific character.

For example:

```
String text = "defensibility";
int index = text.indexOf('e');
index = text.indexOf('e', index + 1);
System.out.println(index);
```

Outcome: 1

- lastindexOf returns an index of the character, from which the first match with the searched string or character is found. The search starts from the end of a string. Returns -1 if there is no match.

For example:

```
int index = "readme.txt".lastIndexOf(".");
System.out.println(index);
```

Outcome: 6

- substring returns a part of a string from the initial string.

For example:

```
// cuts out a string starting from the sixth
// character including to the end of a string.
String world = "Hello World".substring(6);
System.out.println(world);
```

Outcome: World

For example:

```
String world = "Hello World".substring(1, 4);
System.out.println(world);
```

Outcome: ell

Note: *The substring method returns a new line, but an array of characters is taken from a source string. The only things that change are the offset index from the beginning of an array and the number of elements from the offset index (count), due to which, the garbage collector will not be able to clear the memory from the unused part of a string.*

- replace returns a string, replacing a character or a set of characters in the initial string with another character or a set of characters.
- getBytes returns a string as an array of bytes.

The StringBuffer Class

As mentioned earlier, internal data of the **String** class object cannot be altered after it is created, so the **StringBuffer** class is used for working with strings as a soft structure. Unlike strings, the **StringBuffer** class objects cannot be created using string literals, but only with the **new** operator.

For example:

```
StringBuffer sb = "test"; // compilation error
StringBuffer sb = new StringBuffer("test");
```

StringBuffer Class Methods

- append adds a new string to the StringBuffer class object, and returns a reference to this object.

For example:

```
// creating an object
StringBuffer sb = new StringBuffer("test");
```

```
sb.append('-').append("test"); // adding values
                               // chain-wise
sb.append(true); // singular addition
sb.append(1);
System.out.println(sb);
```

Outcome: test-testtrue1

> ***Note:*** *String concatenation in a loop of the String class objects using the + operator may lead to performance problems due to the fact that new objects for strings are constantly created. In such situation, it is preferable to use an object of the StringBuffer or StringBuilder class.*

- insert inserts a string or a character to the StringBuffer class object.

For example:

```
StringBuffer sb = new StringBuffer("I java!");
sb.insert(2, "love "); // 2 is an index of a character,
                       //following which
                       // a string will be inserted
System.out.println(sb.toString());
```

Outcome: I love java!

- reverse reverses the order of characters.

For example:

```
StringBuffer sb = new StringBuffer("palindrome");
sb.reverse();
System.out.println(sb);
```

Outcome: emordnilap

- delete deletes a part of a string starting from the specified index.

For example:

```
StringBuffer phrases = new StringBuffer("I do not
                                        like java!");
phrases.delete(2, 7);
System.out.println(phrases);
```

- deleteCharAt deletes a character from a string at a specified index.
- replace replaces a substring in a string.

StringBuilder and its Differences from the StringBuffer Class

The StringBuilder class has the same purpose and methods as the StringBuffer class. Unlike StringBuffer, the StringBuilder class lacks data synchronization for multistream programming, so it is recommended to use it in a unistream program.

The toString() Method

All classes in Java are implicitly derived from the Object class. The Object class has the toString() method that returns string representation of the object's state. In the StringBuilder and StringBuffer classes, this method returns an object of the String class, which contains a string stored in an object of the StringBuilder or StringBuffer class at the moment of the method call.

The StringTokinazer Class

The StringTokinazer class is used for splitting a string into parts (tokens). A regular expression passed by the second parameter to the class constructor is used as a separator.

Here is an example of splitting strings into words using space character as a separator:

```
String s = "Java is the best programming language.";
StringTokenizer st = new StringTokenizer(s);
while (st.hasMoreTokens())
{
    System.out.println(st.nextToken());
}
```

Here is an example of splitting strings into words using comma as a separator:

```
String s = "Java,is,the,best,programming,language.";
StringTokenizer st = new StringTokenizer(s, ",");
while (st.hasMoreTokens())
{
    System.out.println(st.nextToken());
```

The Features of String Concatenation with Other Types

The strings obtained as a result of concatenation of string literals and variables are not included in the string pool.

For example:

```
String t = "t";
String cat1 = "ca" + t;
```

```
String cat2 = "cat";
System.out.println(cat1 == cat2);
```

Outcome: false

In case of string concatenation with the + operator and other primitive data types, all the subsequent types are casted into string representation. This may cause the result different from what is expected.

For example:

```
String text = "sum =" + 1 + 2; // 1 and 2 are converted
                               // to a string
System.out.println(text );
```

Outcome: sum =12

Addition of parentheses in the expression changes the order of execution. The operation of addition of literals will be executed before casting to a string.

Example with the braces:

```
String text = "sum =" + (1 + 2);
System.out.println(text );
```

Outcome: sum = 3

For example:

```
String text = 1 + 2 + " = 3";
System.out.println(text );
```

Outcome: 3 = 3

Example with a combination of operators:

```
String text = "mul =" + 3 * 2;
System.out.println(text );
```

Outcome: mul = 6

String Comparison

When applying comparison operator (==) to the reference type variables, the **references** to an object stored in variables are compared, therefore, if we create two strings with the same content using the **new** operator, then the comparison of these strings with the == operator will return **false**. Therefore, the **equals** method that compares strings for equality should be used for comparing the identity of different strings.

Example of comparing strings initialized with the literals:

```
String text1 = "java";
String text2 = "java"; // a reference to an object from
                       // the string pool is assigned
System.out.println(text1 == text2);
System.out.println(text1.equals(text2));
```

Outcome: true, true

Example of comparing the strings created with the use of the new operator:

```
String text1 = new String("java");
String text2 = new String("java");
System.out.println(text1 == text2);
System.out.println(text1.equals(text2));
```

Outcome: false, true

A string created using concatenation or the **new** operator can be put in the string pool in software using the intern() method. The intern() method checks whether this string is in the string pool, and places it in the string pool if it is not there. Returns a reference to a string from the string pool.

For example:

```
String text1 = new String("java");
text1 = text1.intern();
text2 = text2.intern();
String text2 = new String("java");
System.out.println(text1 == text2);
System.out.println(text1.equals(text2));
```

Outcome: true, true

***Note!!!** The equals method is not overridden in the StringBuilder and StringBuffer classes, and according to the standard implementation, it compares references to the object in the Object class. In this regard, if it is necessary to compare two objects of the StringBuilder or StringBuffer classes, both objects should be casted to a string using the toString method.*

For example:

```
StringBuffer sb1 = new StringBuffer("abc");
StringBuffer sb2 = new StringBuffer("abc");
System.out.println(sb1.equals(sb2)); // false
```

Outcome: false

Regular Expressions

*Some people, when confronted with a problem, think: "I know, I'll use regular expressions." Now they have two problems.**

Regular expressions** is a formal language used for searching and manipulating text substrings. Metacharacters constitute the basis of a language. Regular string is a sample or a template that defines the rules of searching substrings in the original string.

Regular expressions are a powerful and flexible tool for working with text, which significantly reduce the amount of source code, but the use of regular expressions often complicates the understanding and makes the code harder to read. Therefore, it is recommended to consider the necessity of using regular expressions very carefully.

Most characters in regular expressions represent themselves, except for metacharacters.

Here are the main metacharacters of regular expressions:

Metacharacter	Purpose
^	Beginning of a checked string.
$	End of a checked string.
.	Shorthand for a character class that matches any character.
|	Subexpressions combined in this way are called alternatives.
?	Preceding character is optional.
+	Indicates one or more instances of the immediately preceding element.

* This quote belongs to Jamie Zawinski (JWZ), one of the first developers of Netscape.

** Regular expressions are often called regex of regexp in the programming jargon.

Metacha-racter	Purpose
*	Any number of instances of an element (including zero).
\\d	Digital character.
\\D	Non-digital character.
\\s	Space.
\\S	Any character except for space.
\\W	Any character other than an alphanumeric character or underscore.
\\w	Alphanumeric character or underscore.

The String class has several methods that use regular expressions:

- matches method returns true if the initial string corresponds to a regular expression, otherwise returns false.

This example checks whether the string meets the rules for e-mail address:

```
String email = "unguryan@itstep.org";
// regular expression for checking
// the correctness of the email address
String regular = "[a-zA-Z]{1}[a-zA-Z\\d\\u002E\\u005F]+
                  @([a-zA-Z]+\\u002E){1,2}
                 ((net)|(com)|(org))";
System.out.println(email.matches(regular));
```

Outcome: true

- replaceAll returns the string, in which all the substrings that satisfy the condition of a regular expression are replaced by a string from the second parameter.

For example:

```
String text= "I love coffee!";
text = text.replaceAll("[Cc]offee", "java");
```

- split returns an array of strings, splitting it into parts with a regular expression used as a separator.

Java has the java.util.regex package intended for working with regular expressions.

For example:

```
String url= "http://mystat.itstep.org/";
// regular expression for checking
// the correctness of a URL
String regular = "/^(?:([a-z]+):(?:([a-z]*):)?\/\/)?
                 (?:([^:@]*)(?::([^:@]*))?@)?((?:[a-z0-
                 9_-]+\.)+[a-z]{2,}|localhost|(?:
                 (?:[01]?\d\d?|2[0-4]\d|25[0-5])\.)
                 {3}(?:(?:[01]?\d\d?|2[0-4]\d|25[0-5])))
                 (?::(\d+))?(?:([^:\?\#]+))?(?:\?
                 ([^\#]+))?(?:\#([^\s]+))?$/i";
Pattern p = Pattern.compile(regular);// creates
                                     // a pattern object
Matcher m = p.matcher(url); // creates a matcher object
System.out.println(m.matches());
```

Outcome: true

Lesson 5
Arrays and Strings