Laboratory 1

Utilities and the execution environment

1. Regulation of discipline

- Attendance at laboratory hours is mandatory.
- Students are obliged to respect the schedule of the laboratory, the groups / semigroups of which they belong to.
- The recovery of the laboratory hours is done according to the university regulation.
- Following the laboratory activities, each student will be evaluated. The minimum passing grade is 5. Students with an unfinished situation or with a grade below 5 will not be able to take the discipline exam.

2. Laboratory objectives

- presentation of the obligations incumbent upon the students in order to be able to take the discipline examination;
- presentation of the tools required for the laboratory hours of the Real Time Systems discipline;
- brief recapitulation of some basic notions of *Java* programming language.

3. Tools needed for the laboratory

In the first part of the lab, Java applications will be developed using Java 2 SE 1.6 (J2SE). In order to be able to write the first Java application, the first step is to download and install the Java development kit. The steps to be followed are as follows:

- Select the desired platform from the page http://www.java.com/en/download/manual_v6 .jsp, according to the operating system it will have to run.
- After downloading the Java Development Kit (JDK), the installation is done by simply executing the executable file.
- In addition to the J2SE components, it is useful to download and unzip to the jdk1.6.x_<version>/docs directory, the *html* documentation corresponding to the J2SE platform are installed.

Theoretically after installing *J2SE*, Java applications can be built, compiled and ran without the need for any additional tools. For this reason, we have to do the following steps:

STEP 1. Edit the source file and save it with the .java extension (using Notepad, for example).

STEP 2. Run the command prompt on Windows, compile the source file using the javac.exe utility. The compilation will generate the Java executable file (byte code) with the extension: .class.

STEP 3. Execute the application using the java.exe program.

Developing complex *Java* applications, which usually contain multiple source files, can become very difficult when using a simple text editor. It is recommended that the programmer choose and install in addition to the J2SE platform, an application that will help in applications development. Such an application is called IDE (Integrated Development Environment). Currently, there is a wide variety of development environments in the market for programmer to choose from. Some examples are:

- *Borland Jbuilder* www.borland.com/jbuilder ;
- *Java CodeGuide* www.omnicore.com ;

- *JCreator* www.jcreator.com ;
- *NetBeand* www.netbeans.org ;
- *Eclipse* www.eclipse.org ;
- *IntelliJ IDEA* www.jetbrains.com ;
- etc.

The integrated Eclipse development environment will be used within the laboratory. It is free and can be downloaded from the http://www.eclipse.org/downloads/index.php page (downloading the Eclipse IDE for Java Developers version is recommended). The installation consists of unzipping the Eclipse environment in the desired location, then running the eclipse.exe program.

It is recommended that the J2SE be installed before installing the Eclipse environment.

Observation:

Installing additional tools is needed to develop real-time Java applications, this will be presented in Laboratory 8.

5. Recapitulation of the main notions of Java programming language

Java is a high-level programming language, originally developed by JavaSoft, a company under the Sun Microsystems company (later acquired by Oracle). Among the main features of the language, the following are worth mentioning:

- **simplicity**, eliminates operator overloading, multiple inheritance and all "facilities" that can cause writing a confusing code;
- **robustness**, eliminates the frequent sources of errors that occur in programming by eliminating pointers, memory automatic management and eliminating memory cracks through a procedure to automatically free memory from objects that are no longer used;
- **completely object oriented**, complete eliminates the procedural programming style;
- **ease** in network programming;
- **security**, providing strict security mechanisms for programs implemented through: dynamic verification of code for detecting dangerous sequences, imposing strict rules for running programs launched on remote computers, etc.
- it is **architecturally neutral**;
- **portability**, in other words *Java* is a language independent of the work platform, the same application running, without any modification, on different systems such as Windows, UNIX or Macintosh;
- **compiled** and **interpreted**;
- ensures **high performance**;
- allows programming with **threads of execution** (*multithreaded*).

5.1 Java: Complied and Interpreted Language

Depending on how the programs are executed, the programming languages are divided into two categories:

- **interpreted**: the instructions are read line by line by a program called interpreter and translated into machine instructions; advantage: simplicity; disadvantages: low execution speed
- **compiled**: the source code of the programs is transformed by the compiler into a code that can be executed directly by the processor; advantage: fast execution; disadvantage: lack of portability, the code compiled in a low level format can only be run on the platform where it was compiled.

The compilation of a *Java* program results in a set of special files called *byte code*. These files are not executable on any operating system. To be able to run it requires a Java *Virtual Machine* that will interpret the byte code and execute instructions that are specific to the operating system that it is running on. This ensures platform independence and portability of *Java* applications.

5.2 About the Java Virtual Machine

A standalone application, written in *Java*, is compiled and runs on a so-called *Java Virtual Machine*. This makes it possible for *Java* applications to run on different platforms (Sun, MacOS, Win32, and Linux) without having to recompile these applications for each of them separately. Thus Java applications are platform independent.

The practical implementation of the platform independence concept was achieved by using a virtual computer that actually runs *Java* compiled applications. Following the compilation of *Java* source files will not generate executable code for a particular platform, but will generate an intermediate code called *byte code*. This byte code is similar to the assembly language but cannot be run directly on any operating system.

To run a *Java executable code* (so-called *byte code*), you need a special program that will interpret this byte code and execute operating system-specific instructions. This program that interprets the byte code is called *Java Virtual Machine*. *Java Virtual Machine* is an abstract computer. Like real computers, it has a set of instructions, a set of registers and uses different memory zones.

The main components of JVM are:

- *Class Loader Subsystem* - this module is responsible for loading classes into memory.
- *Execution Engine* - the mechanism responsible for executing the instructions in the classes uploaded to the *JVM*.
- *Runtime Data Area* - the zone where necessary components are stored during the running of a program. In this zone, the loaded classes, the built objects, the information about the methods call, the parameters of the variables, the values returned by the methods, the data about the execution threads are stored.

5.3 Freeing the memory of useless objects

All objects that are created within the virtual machine during program execution are placed in a memory zone called Heap. Removing them from memory is done automatically by a component of the *JVM* called the *Garbage Collector* (*GC*).

The *GC* mechanism of freeing the memory from objects that are no longer useful offers two important advantages. First of all, the programmer does not have to explicitly take care of releasing the useless objects. Due to implementation errors that lead to memory overload situations, the programmer may lose many hours to find and repair the defect. The second advantage is that the *GC* ensures the integrity of the programs - it does not allow the accidental deletion of objects or memory zones that would lead to the loss of the integrity of the program or the virtual machine.

The additional processor's load given by the *GC* component is a disadvantage of this approach to automatic memory release. This problem is solved to a large extent by developing efficient algorithms that are implemented at the *GC* level.

It is important to remember that the time when an object is to be released from memory cannot be predicted in advance. The programmer does not have to base his program architecture on the basis of certain assumptions where a particular object will be released from memory.

Observation:

Although the *GC* mechanism ensures releasing memory from objects that are no longer referenced to, there is a danger that a poorly thought-out and implemented program may have problems allocating memory and reaching the state of memory overload, leading to program interruption. This can happen because *GC* deletes from the Heap zone only those objects that are not referenced by any variables.

Although it is not possible to know in advance exactly when an object is released from memory, the objects to be deleted are announced by calling the completed method ().The finalize () method is found in any Java object (being inherited from the Object base class) and is called by the GC when the object is to be deleted.

In code 1 Sequence presents a program that demonstrates the release of each object from memory is called the *finalize*() method.

Observation:

Getting into the action of *GC* and release the memory from objects (so implicitly calling the *finalize*() method) is not guaranteed by Java Virtual Machine. As a result, all releasing resource operations that are needed to be performed within an application (for example, closing network connections, files, database connections, etc.) must be performed by the programmer manually by building some methods that are explicitly called at the right time.

Code Sequence 1

```
public class GarbageCollectorTest {
        static int removedObjects;
        public static class Car {
                String name;
                Car(String name) {
                        this.name = name;
                }
                public void finalize() {
                        removedObjects++;
                        System.out.println("The car " + name
                              + " is removed. Number of removed cars is "
                              + removedObjects);
                    }
          }
          public static void main(String[] args) {
                    Car myC = null;
                    for (int i = 1; i < 10; i++) {
                    myC = new Car(" Car " + i);
                    myC.finalize();
                    }
          }
}
```

5.4 The lexical structure of the language

5.4.1 The character set

Java language works natively using the *Unicode* character set. This is an international standard that replaces the old *ASCII* character set and uses 2-byte character representation, which means that 65536 characters can be represented, unlike ASCII, where it was possible to represent 256 characters. The first 256 *Unicode* characters correspond to those in *ASCII*, the reference to the others being made by \ **uxxxx**, where **xxxx** represents the character code.

Example:

- \ u0030 - \ u0039: ISO-Latin digits 0 - 9
- \ u0660 - \ u0669: Arabic-Indic digits 0 - 9
- \ u4e00 - \ u9fff: letters in the Han alphabet (Chinese, Japanese, Korean)

5.4.2 Keywords

The reserved words in Java are the same in C++, with a few exceptions.

5.4.3 Identifiers

There are unlimited sequences of Unicode letters and digits, starting with a letter. Identifiers are not allowed to be identical to other reserved words.

5.4.4 Literals (constants)

- *Integer literals*

3 numbering bases are accepted: base 10, base 16 (starting with 0x characters) and base 8 (starting with digit 0) and can be of two types:

- normal, (represented by 4 bytes - 32 bits);
- long (8 bytes - 64 bits): ends with the character L (or l).

- *Floating point literals*:

For a literal to be considered a floating it must have at least one decimal point, be in exponential notation, or have the suffix F or f for normal values (represented by 32 bits), respectively D or d for double values (represented 64 bits).

- *Logical literals*:

- *true*: Boolean value of truth;
- *false*: Boolean value of false.

Unlike C ++, integer literals 1 and 0 do not have the role of true and false.

- *Character literals*:

A character literal is used to express the characters of the *Unicode* code.

The representation is done either using a letter or an *escape* sequence written between apostrophes.

The *escape* sequences allows the representation of characters that have no graphic representation and the representation of special characters such as "*backslash*", the *apostrophe* character, etc. Predefined *escape* sequences in Java are presented in Table 1.

Table 1. Predefined *escape* sequences

Code	***escape* sequences**	**Character**
\u0008	'\b'	*Backspace*(BS)
\u0009	'\t'	Tab horizontal (HT)
\u000a	'\n'	*linefeed* (LF) – new line
\u000c	'\f'	*form feed* (FF)
\u000d	'\r'	*Carriage control* (CR)
\u0022	'\"'	*quotation mark*
\u0027	'\''	*Single quote*
\u005c	'\\'	*Backslash*

- *String literals*:

A literal string consists of zero or more characters between quotation marks. The characters that make up the character string can be graphic characters or escape sequences like those defined in character literals. The empty string is "". If the string is too long it can be written as a smaller string concatenation. The string concatenation is done with the operator "+", for example: "alpha" + "beta" + "gamma". Any string is actually an instance of the *String* class, defined in the *java.lang* package. *String* objects are immutable (any change in the string causes the creation of a new object) [ECK, 2006].

5.4.5 Separators

A separator is a character that indicates the end of one lexical unit and the beginning of another. In *Java* the separators are as follows: () {} []; ,. The instructions of a program are separated by a semicolon.

5.4.6 Operators

- attribution: =
- mathematical operators: +, -, *, /,%

There are operators for self-incrementing and self-decrementing (post and pre).

Example:

x++, ++x, n--, --n.

Observation:

The evaluation of the logical expressions is done by the *short-circuit* method, the evaluation stops when the truth value of the expression is surely determined.

- logical operators: &&(and), ||(or), !(not) ;
- relational operators: <, <=, >, <=, ==, != ;
- biţ opeartors: & (and), |(or), ^(xor), ~(not) ;
- translation operators <<, >>, >>> (*shift* to the right without a sign) ;
- *if-else* operators*: logical expression ? val_pt_true : val_pt_false;*
- , (*comma*) operator: is used for sequential evaluation of operations: *int x=0, y=1, z=2;*
- "+"operator: for string concatenation, for example:
  ```
  String s="abcd";
  int x=100;
  System.out.println(s + " - " + x);
  ```
- conversion operators *(cast) : (data_type),* example:
  ```
  int i = 200;
  long l = (long)i; //widening conversion
  long l2 = (long)200;
  int i2 = (int)l2; //narrowing conversion
  ```

5.4.7 Comments

There are three types of comments in Java:

- multi-line comments, closed between / * and * /.
- multi-line comments regarding documentation, closed between / ** and * /. The text between the two sequences is automatically moved to the application documentation by the automatic documentation generator *javadoc*.
- single line comments starting with //.

Observation:

1. we cannot write comments inside other comments.
2. we cannot enter comments within character literals or strings.
3. the sequences / * and * / may appear on a line after the sequence // but they lose their meaning; the same is true of the // sequence in comments starting with / * or / **.

5.5 Data types

Java data types fall into two categories: ***primitive types*** and ***reference types***.

Java starts from the premise that "everything is an object". So data types should actually be defined by classes and all variables should actually store instances of these classes (objects). In principle this is true, however, for the ease of programming, there are also the ***primitive types*** of data, which are the usual ones:

- arithmetic
  - integers: *byte* (1 byte), *short* (2), *int* (4), *long* (8)
  - real: *float* (4 bytes), *double* (8)
- character: *char* (2 bytes)
- logical: *boolean* (*true* and *false*)

In other languages the format and size of the primitive data types used in a program may depend on the platform where program runs. In *Java* this is no longer valid, any dependency on a specific platform being eliminated.

Vectors, classes, and interfaces are ***reference types***. The value of a variable of this type is, in contrast to the primitive types, a reference (memory address) to the value or set of values represented by the respective variable.

There are three types of *C* data that are not supported in *Java*. These are:

pointer, struct and *union*. The *pointers* were eliminated because they were a constant source of errors, their place being taken by reference type, and *struct* and *union* are no longer valid as long as the composite data types are formed in *Java* through classes.

5.6 Variables

Variables can be either a primitive type of a date or a reference to an object. The code convention of variables in *Java* is given by the following criteria:

1. the *final variables* (*constants*) are written in capital letters;
2. the *normal variables* are written as follows: the first letter is small; if the variable name consists of several lexical atoms, then the first letters of the other atoms are capitalized, for example:

```
final double PI = 3.14;
int value = 100;
long numericElement = 12345678L;
String myFavoriteBeverage = "water";
```

Depending on where the variables are declared, they are divided into the following

categories:

1. *member variables*, declared within a class, visible to all methods of the respective class and to other classes depending on their access level (see "Declaring member variables");
2. *local variables*, declared in a method or a block of code, visible only in the respective method / block;
3. *methods' parameters*, visible only in the respective method;
4. *exception handling parameters*.

Observation:

The variables declared in a *for* cycle control remain local to the cycle body.

Unlike C ++ it is not allowed to hide a variable:

```
int x = 12;
{
        int x = 96; // illegal
}
```

5.7 Execution control

The *Java* instructions for execution control are similar to those in C (see Table 2).

In *Java* there is no ***goto*** instruction. However, you can define labels of the form *name_lable* used in expressions like: *break name_label* or *continue name_label* (see Code Sequence 2).

Table 2. Execution control in *Java*

Decision Instructions	

If-else	*if* (exp_booleană) { /*...*/}
	if (exp_booleană) { /*...*/} *else* { /*...*/}

switch-case	*switch* (variabilă) {
	case val1 : /* ... */ *break;*
	case val2 : /* ... */ *break;*
	/*...*/
	default : /*...*/
	}

Jump instructions	

for	for (initialization; exp_boolean; step_ iteration)
	Ex*: for(int i=0, j=100 ; i<100 && j>0; i++, j--)*
	{/ ... /*}*
	Note: Both the initialization and the iteration step can have several comma separated instructions.

while	*while (exp_booleană) {*
	/... */*
	}

do-while	*do {*
	/... */*
	}
	while (exp_boolean) ;

Instructions for handling exceptions	
try-catch-finally, throw	(see “Exception Handlers”)
Other instructions	
break	Forcedly leaves the body current iteration
continue	finish the current iteration
return	*return* [valoare]; terminate a method
Label	defines a label

Code Sequence 2

```
int  i = 0;
label: while  (i < 10) {
        System.out.println("i=" + i);
                int  j = 0;
                while  (j < 10) {
              j++;
                              if  (j == 5)
                                      continue  label;
                              if  (j == 7)
                                       break  eticheta;
                   System.out.println("j=" + j);
        }
        i++;
}
```

5.8 Vectors

The vector declaration is made as follows:

Type [] vectorName; or

TypeVector name [];

Example:

```
int[] integers;
String addresses[];
```

The instantiation is done through the *new* operator and has the effect of allocating the memory for the vector, more precisely specifying the maximum number of elements that the vector will have:

nameVector = new Type[dimension];

Example:

```
int  v[] = new  int[10]; // allocates space for 10 integers
String[] adrese = new  String[100];
```

After declaring a vector, it can be initialized, meaning that its elements can receive some initial values, obviously if this is the case. In case the instantiation is missing, the memory allocation being automatically made according to the number of elements that the vector is initialized with.

Example:

```
String colors[] = {"Red", "Yellow", "Green"};
int  []factorial = {1, 1, 2, 6, 24, 120};
```

The first index of a vector is *0*, so the positions of a vector with *n* elements will be between *0* and *n-1*.

Constructors such as *Type VectorName* [*dimension*] are not allowed, as memory is only allocated through the *new* operator.

In *Java* multidimensional paintings are actually vector of vectors.

Example:

```
int m[][]; // declaration of a matrix
m = new int[5][10]; // with 5 lines, 10 columns
int l[] = m[1]; // m[1] is a vector with 10 elements
```

Example:

```
int [ ]a = new int[5];
int la = a.length; //has the value 5
int m[][] = new int[5][10];
int lm = m[0].length; //has the value 10
```

Copying a vector to another vector is done using the *System.arraycopy*() method:

Example:

```
int x[ ] = {1, 2, 3, 4};
int y[ ] = new int[4];
System.arraycopy(x,0,y,0,x.length);
```

The implementation of vectors with a variable number of elements is provided by the *Vector* class in the *java.util* package. A *Vector* object contains only *Object* type elements.

5.9 Strings of characters

A string can be represented by a vector consisting of *char* elements, a *String* type object, a *StringBuffer* or a *StringBuilder* object.

If a string is constant then it will be declared as *String*, otherwise it will be declared with *StringBuffer*.

String concatenation is done by using the "+" operator.

In *Java*, the "+" concatenation operator is extremely flexible in the sense that it allows the concatenation of strings with objects of any type that have a string representation.

5.10 Using command line arguments

A *Java* application can receive any command line arguments at launch time. These arguments are useful to allow the user to specify various options related to the operation of the application or to provide certain initial data to the program.

Programs that use command line arguments are not 100% pure *Java* because some operating systems such as *Mac OS* do not normally have command line.

The command line arguments are introduced when launching an application, being specified by the name of the application and separated by space. For example, suppose the *Sort* application lexicographically sorts the lines of a file and receives as an argument the name of the file to sort. To order the file "*persons.txt*" the launch of the application will be done as follows:

```
java Sort persons.txt
```

Therefore, the general format for launching an application that receives arguments from the command line is:

```
java ApplicationName [arg1 arg2 . . . argn]
```

If there are several arguments, they must be separated by spaces, and if one of the arguments contains spaces, then it must be placed between quotation marks.

6. Knowledge verification

1. There are two complex numbers: 2+5i and 4-i. To create an application that calculates the summation and the product of the two numbers, to verify the result.
2. Two 3X3 arrays are given,
   - first matrix: R1 = [2 3 1], R2 = [7 1 6], R3 = [9 2 4],
   - the second matrix: R1 = [8 5 3], R2 = [3 9 2], R3 = [2 7 3],

   Calculate the sum and the product of the two matrices. Check the result.
3. What is a class and what is it made of?
4. What is a constructor?
5. How many types of access modifiers are there in Java?