DOI: 10.5281/zenodo.3592042

IMPLEMENTATIONS OF CLASSES AND OBJECTS IN APPLICATIONS FOR ECONOMIC ORGANIZATIONS

Emilia VASILE, PhD Professor
Athenaeum University, Bucharest, Romania
rector@univath.ro

Dănuţ-Octavian SIMION, PhD Associate Professor
Athenaeum University, Bucharest, Romania
danut_so@yahoo.com

***Abstract**: The paper presents the implementations of classes and objects in applications for economic organizations build on Java technologies. The IT systems are informations system that allows the collection, transmission, storage, data processing and dissemination of information thus obtained by using the means of information technology and specialized personnel in the automatic processing of data. The computer system comprises all internal and external, formal or informal information used within the company as well as the data that were the basis of their obtaining, the software required to process data and disseminate information within the organization, the procedures and techniques for obtaining based on primary data and disseminating information, the hardware platform necessary for data processing and dissipation of information and personnel specialized in data collection, transmission, storage and processing. The computer system is structured to meet the needs of different user groups such as management factors at the level of strategic, tactical and operational management, the personnel involved in the process of data collection and processing and the personnel involved in the process of scientific research and the design of new products and manufacturing technologies. An economic application that uses implementations of classes and objects offers more flexibility and a good encapsulation of the main components that are specific to IT systems.*

Keywords: *Java objects, business classes, IT systems, programing logic, informational support, application components, business interactions*
JEL Classification: *C23, C26, C38, C55, C81, C87*

1. Introduction

A class designates a category of objects and acts as a blueprint for creating such objects. A class models an abstraction by defining the properties and behaviors for the objects that represent the abstraction. An object presents the properties and behaviors defined by its class. The properties of an object of a class are also called attributes and are defined by fields in Java. A field in a class definition is a variable that can store a value that represents a particular property. The behaviors of an object of a class are also known as operations and are defined using methods in Java. The domains and methods in a class definition are collectively named members. One of the fundamental ways of representing complexity is abstractization. An abstraction denotes the essential properties and behaviors of an object that differentiates it from other objects. The essence of OOP is the modeling of abstractions, using classes and objects. The difficult part in this approach is finding the correct abstractions (Prabhu, 2019; Umair, 2019).

Each object is a data structure, associated with a set of methods. From the point of view of traditional procedural programming, the methods are functions or procedures. The invocation (appeal) of a method of an object is considered in the POO as the transmission of a message to that object, indicating the operation to be performed and the parameters required for this purpose. Objects with the same data structure and methods are a class. The class is an extension of the data type concept. The characteristics of objects and classes are: identity, encapsulation, aggregation, classification, inheritance and polymorphism (Monus, 2019; Pankaj, 2019).

API is the interface through which an application program accesses the operating system of the computer and other services offered by the computer or the network. Because Java is a POO language, various services are viewed as objects belonging to classes. In this situation, each Java API contains the description of the classes in a certain category of services: for communication with the operator, for the incoming / outgoing operations, for network communication, etc. The description is made from the point of view of the application programmer, that is, for each class, only the purpose of the respective class is presented, as well as the public data and methods of the class, without giving indications on how the class is implemented. Thus the principle of encapsulation is respected. The programmer can use the respective classes in the applications he develops, because he knows all the data and methods that are accessible to them, their significance and the way they are used.

2. The Java classes and objects for business components

According to the principles of Object Oriented Programming (OOP), each object is the instance of a class. However, there may be several instances of the same class. According to the principle of identity each object must have a unique reference, by which it is distinguished from the other objects. In Java, object references are values of special variables, called reference variables.

The declaration of the reference variables is made in the form:

```
<class><variablename>[=<initialization>][<variablename>[=<ini-
tialization>]] *;
```

where
<class> - the name of the class of objects referred to by the respective variables;
<variablename> - an identifier;
<initialization> - an expression that assigns to the variable a reference to an object in class <class>.

This can be:

- another variable, which already has as value a reference to an object of class <class> or of a descending class;
- a string literal, if <class> is String;
- an expression by which a new instance of the <class> class or of a descendant class is constructed and which has as value a reference to the newly constructed instance.

As an example for the first two ways of initializing some reference variables we can give the following statement:

```
String str1 = "an example of a string", str2 = str1, str3;
```

The String class, which we will study in detail later, is the string class. In the above statement it is stated that str1, str2 and str3 are variables referring to objects in the String class. The variable str1 receives as a starting value a reference to the string “an example of a string”; str2 is initialized with the same reference as str1; Finally, the str3 variable is not initialized. The program from the InitSir.java file gives examples of initializations and assignments of references to objects in the String class by applying these procedures.

According to the OOP principles, the class incorporates a data structure and a set of methods. The data is placed in fields. Each field has a name and a value that belongs to a certain type of data. At the conceptual level, the fields can contain data of primitive types or objects. In the case of data of primitive types, the field even contains its value, while, in the case of objects, the field actually contains a reference to the respective object. However, in order not to complicate the exposition unnecessarily, we will consider that the name of the field is, at the same time, the name given to the primitive value or to the object that it contains. One can easily observe the similarity between the field concept and the variable one (Chhajer, 2019; Loganathan, 2019).

In Java, the concept of variable is used for data in methods, while for data contained in classes or objects the field name is used. This name is also used in other programming languages for data contained in structures. In a program you can use several objects in the same class. In principle, the values contained in the fields of these objects are different, so each object has its own state. Consequently, in the memory “Java virtual machine” there is a data area reserved for each object, which contains the data fields of that object. There is, however, a category of fields, called static fields, whose values are unique to all objects in that class. It is considered that these fields do not belong to the objects, but to the class itself. In memory there is, for each class, a single memory area that contains its static fields. Static fields are thus considered to be class variables, while ordinary (non-static) fields are considered instance variables (Prabhu, 2019; Pankaj, 2019).

Methods are functions or procedures. The function returns a value, while the procedure does not return a value, but is used only to obtain a side effect. But there are functions that have side effects. For this reason, in Java, the procedure is considered to be a function, which returns the void value. Each method has a form signature:

<type> <name> ([<formal parameter list>])

where:

<type> - the name of a primitive data type or class, representing the type of the returned value; if the method does not return a value, the type is void;

<name> - is the name of the method and is an identifier;

```
<formal_parameter_list>::= <parameter_statement> [, <parameter_
statement>] * <parameter_parameter>::= <parameter_type>
[<parameter_name>]
```

According to this specification, the list of formal parameters, if there are such parameters, contains one or more parameter statements, separated by commas. Each parameter declaration contains its type and, optionally, its name. The parameter type can be a primitive type or a class name. Formal parameters are those that appear in the specification of a method and in its signature, as opposed to the actual parameters, which appear when the respective method is used.

In the specifications of the Java API documentation, for each method, its signature is preceded by one or more modifiers, of the form:

```
<modifier>::= <modifier_of_access> | static | end | abstract

<modifier_of_access>::= public | private | protected
```

If the access modifier is public, the respective method can be accessed from any other class or from any object. If the access is protected, the method can be used only in the descending classes from the one in which it was defined. The methods with private access can only be used in the class in which they were defined, being "visible" from other classes. For this reason, such methods are not given in the API, depending on the implementation of the respective class. Here are two examples of such specifications:

The static modifier has the meaning that the respective method belongs to the class and not the instance, so it is a static method. Such a method can only change the static fields of the respective class, not the fields of the instances (Monus, 2019; Umair, 2019).

The final modifier has the meaning that the respective method can no longer be redefined in the derived classes, so it is a final method. The abstract modifier indicates that this is an abstract method, for which it is specified, for the time being only the signature and the meaning, but which must be defined in the derived classes. Classes that contain abstract methods are called abstract classes. Examples:

```
public static double atan2(double a, double b)
public boolean equals(Object obj)
public final void wait(long timeout)
```

The first of these methods belongs to the Math class and calculates the tangent arc in the ratio a / b; both the arguments and the returned value are of double type. The atan2 () method is static. The second method belongs to the Object class and compares between two objects, returning a Boolean value (true or false). Unlike the first two methods, which are functions themselves, the third is a procedure, because it returns the void value. The method is final, so it cannot be redefined in the derived classes. It also belongs to the Object class. The formal parameters of the methods in the examples above could be specified only by their types, as follows:

```
public static double atan2(double, double)
public boolean equals(Object)
public final void wait(long)
```

Specifying the names of the formal parameters can be useful only to make the description of their meaning clearer.

In order for a program to use the values contained in the fields of the classes or instances (objects) it is necessary to use references to them. The reference to a field of an object is made in the form

```
<refer_to_object>.<fieldname>
```

For example, if the variable w has the value of reference to an object and this object contains the alpha field, the reference to the alpha field is made in the form w.alpha. Consider now that the alpha field itself is an object, which contains a field called beta. The reference to this field will be made in the form w.alpha.beta so it is indicated the beta field, contained in the object in the alpha field, which - in turn - is contained in the object referred to by the variable w. Such names, preceded by a point and another name, are called qualified names in the Java language.

The reference to a static field (of a class) is made in the form:

```
<class_name>.<fieldname>
```

For example, the reference to the static field PI in the Math class is made in the form of the qualified name Math.PI.

The reference to a method of an object is made in the form:

```
<refer_to_object>.<invoke_method>
```

in which the reference to the object is made as in the case of using a field of the respective object, and the invocation (appeal) of the method is made in the form:

```
<method_name> (<list_of_effective_parameters>)
```

The actual parameters (also called current parameters) are those whose values are substituted for the formal parameters when invoking the respective method. Each effective parameter is an expression of the same type with the formal parameter that it substitutes. For example, if the variable z is a reference value to an object that contains the method

```
public int omega(String str, int b)
```

the invocation of this method can be done in form

```
z.omega("a string", 723)
```

Instead of the literal "a string" you could put any expression that has the value of an object of the String class, and instead of the literal 723 you could put any expression that has the value of a number of type int, or that can be converted into this type. .

The reference to a static method (of a class) is made in the form

```
<class_name>. <method_invocation>
```

in which the invocation of the method is done as in the case of an ordinary (non-static) method. For example, invoking the atan2 () method in the Math class can be done by the expression Math.atan2 (x + 3, 2 * y-1) in which x and y are numeric variables.

Both static fields and static methods, instead of the method name can be used in the reference expressions and the name of an instance of the respective class. For example, if m1 is a reference to an object in the Math class, then the expressions in the examples above can be written in the form m1.PI and m1.atan2 (x + 3, 2 * y-1).

3. Development of class conversions and objects in Java

When assigning values to reference variables and other operations, conversions from one class to another are sometimes required. In Java, a variable referring to objects in a class can be assigned as reference values to objects in that class or any of its descendant classes, without the need for explicit conversion.

Instead, in order to move from the superclass (or from an ascendancy) to the class, explicit cast conversion is required, as in the case of conversions for primitive data types. The cast operator is formed, in this case, from the bum of the class to which the conversion is made, between round brackets. Thus, since the Object class is a superclass of the String class, the instructions are valid:

```
Object ob;
String str1=”a string”, str2, str3;
str2=str1;
ob=str1;
str3=(String)ob;
```

The assignment of str2 = str1 is possible, since it is made between variables of the same class. The assignment ob = str1 does not require explicit conversion, since it is done from class to superclass. Instead, assigning str3 = (String) ob required explicit conversion by cast, because it is done from the superclass to the class. By this, the programmer has assumed the responsibility that the variable ob has as a value a reference to an object in the String class.

When assigning values to reference variables and other operations, conversions from one class to another are sometimes required. In Java, a variable referring to objects in a class can be assigned as reference values to objects in that class or any of its descendant classes, without the need for explicit conversion. Instead, in order to move from the superclass (or from an ascendancy) to the class, explicit cast conversion is required, as in the case of conversions for primitive data types. The cast operator is formed, in this case, from the bum of the class to which the conversion is made, between round brackets. Thus, since the Object class is a superclass of the String class, the instructions are valid:

```
Object ob;
String str1=”a string”, str2, str3;
str2=str1;
ob=str1;
str3=(String)ob;
```

The assignment of str2 = str1 is possible, since it is made between variables of the same class. The assignment ob = str1 does not require explicit conversion, since it is done from class to superclass. Instead, assigning str3 = (String) ob required explicit conversion by cast, because it is done from the

superclass to the class. By this, the programmer „assumed responsibility" as the variable ob effectively has as value a reference to an object of the class String.

Each object class is provided with one or more special methods, called constructors, which have the role of building in the memory of the Java virtual machine a new instance of the respective class. These builders bear the name of the class to which they belong. In the documentation of the respective class, all the manufacturers are indicated, specifying the number of parameters, as well as the type and significance of each parameter. For example, in the case of the String class, several builders are specified in the documentation, of which we mention here the following two:

String () - constructor without parameters, which constructs an empty string (which does not contain any characters);

`String (String str)` - which constructs a new object in the String class, but with a content string identical to the one contained in the object with the reference str received as a parameter.

In order to execute a certain constructor, the new operator must be applied, in the form of the expression

```
new <constructor>
```

which has as a side effect the construction in the memory of the Java virtual machine of a new object, resulting from the application of the invoked constructor, and as a value of the expression a reference to the respective object is obtained. For example, the expression

new String („new string") it has the effect that in memory an object is constructed that contains the string „new string", and as a value of the expression a reference to this object is obtained (Monus 2019; Loganathan 2019).

If, for some reason (for example, lack of memory space), the object could not be built, the new operator returns the null value, that is, a „to nothing" reference.

The reference returned by the new operator can be used to initialize a reference variable, or to assign a value to such a variable. Let, for example, be the instructions:

```
String a = "a string", b = new String (a), c, d, e;

 c = new String (b);

 d = b;

 e = c;
```

three variables are declared with reference to string a, b, which are given values as follows: a is initialized using directly the literal „a string"; b is initialized with the reference returned by the expression new String (a), so with a reference to a new object in the String class, having the same content as that of reference a; Finally, the variable c is assigned the value returned by the expression new String (c), that is, a reference to a newly constructed object, containing the same string as object b. According to the principle of identity, these objects will have different references, even if they have identical contents. Instead, by assigning instructions d = b; and e = c new objects are not constructed, but are assigned to variables d and e as reference values to already existing objects.

The variables a, b, c, d, e contain references to objects in the String class. The references are here conventionally represented by arrows. In fact, they are the addresses of the memory areas occupied by the respective objects. Although the three objects contained the same, they each have their own identity, so they occupy different areas of memory and have different references (addresses) (Chhajer 2019; Umair 2019).

Let's now look at what happens if the variable b is given as a reference value to another object in the String class, using for this purpose, for example, one of the instructions:

```
b = "high sir"; or

b = new String ("alt sir");
```

The value of variable d remained the previous one (indicates the object to which it indicates above and b), instead variable b now contains a reference to the newly formed object, with the content of „another string".

If an object is no longer needed, it can be destroyed, ie removed from memory. In Java virtual machine, there is a garbage collector that automatically frees up the memory space occupied by objects to which there is no reference. As a result, the programmer is no longer able to explicitly foresee the destruction of objects and, therefore, the classes no longer contain destructors, as in other POO languages.

Example of a situation in which some objects remain without references:

```
e = b;

a = "Our Sir";

c = new String (a);
```

The following transformations were made:

- the reference e has been assigned the same value as the reference b;
- an object with „New Sir" content was built. and the variable received as reference value to this object;
- an object with the same content as the indicated one was constructed, and the variable c now indicates this new object.

As a result, two of the „one string" objects originally built remained without reference. There is no way in the program to use these objects, so they must be eliminated. This role is fulfilled by the waste collector. However, the programmer does not have the opportunity to decide when the actual elimination of these objects will occur, the decision pertaining only to the waste collector.

4. Conclusions

In object-oriented programming, the close connection between data and the operations performed on them also extends to the data structures. Each objects is a data structure, associated with a collection of methods (functions or procedures) through which these data are manipulated. Each object is characterized by its state and behavior. The state of the object is characterized by the values that have, at a given moment, the data contained by it. If one or more such values are changed, the state of the object is changed. (Chhajer, 2019; Pankaj, 2019). From the point of view of procedural programming, the method can be a function or a procedure. The difference between them is that the function is invoked / called to obtain its value, also called inverted value, while the procedure does not return a value, but is executed to obtain a side effect, for example reading data from the keyboard, displaying some data on the screen, etc. Object-oriented programming supports the implementation of computer-oriented modeling techniques in the real world, which - in turn - are a component of object-oriented analysis (Prabhu, 2019; Loganathan, 2019). The objects and the implementations of those are the the best chice in developing enterprise economical applications. In this way the economical entitites are represented with their properties and their methods that define the economical actions and the business flows in which are involved.

References

Chhajer, B. (2019). Object Oriented Programming Concept with an Example "Is a" and "Has a" relations, [online] *Java Sprint.* Available at: www.javasprint.com/java_training_tutorial_blog/object_oriented_programming_oops

Loganathan, G. (2019). Object Oriented Programming, [online] *JavaHelps.* Available at: www.javahelps.com/2015/01/object-oriented-programming.

Monus, A. (2019). OOP Concepts in Java with examples, [online] *JournalDev.* Available at: www.journaldev.com/12496/oops-concepts-java-example/.

Pankaj (2019). OOPS Concepts in Java – OOPS Concepts Example, [online] *JournalDev.* Available at: www.journaldev.com/12496/oops-concepts-java-example.

Prabhu, R. (2019). Object Oriented Programming (OOPs) Concept in Java, [online] *GeeksforGeeks.* Available at: www.geeksforgeeks.org/object-oriented-programming-oops-concept-in-java.

Umair, M. (2019). A Systematic Approach to Write Better Code With OOP Concepts, [online] *DZone.* Available at: dzone.com/articles/object-oriented-programming-concepts-with-a-system.