Rudolf PECINOVSKÝ

Od: International Conference on New Horizons in Education <inteconferences@gmail.com>
Odesláno: pondělí 4. května 2015 15:55
Komu: rudolf@pecinovsky.cz; pavjar@vse.cz
Předmět: INTE 2015 Paper Submission is accepted

International Conference on New Horizons in Education

INTE 2015 Paper Submission is accepted

Dear Colleaugue,

We are pleased to inform you that the Advisory Board of INTE 2015 (International Conference on New Horizons in Education) after peer blind review by 2 reviewers has decided to ACCEPT your paper to be presented at INTE 2015 Conference.

You can download your acceptance letter from our website's accepted papers section:

http://www.int-e.net/accepted_papers

You can complete your registration process by paying the conference fee.

All accepted papers in English will be published in Procedia Social and Behavioral Science and others will be published in Proceding Book with ISSN number which will be prepared by conference organization.

You can check our previous publications here:

http://www.int-e.net/intepubs

Also after registration you will be able to

- get Participation Certificate
- enter to all keynote speeches
- enter presentation sessions
- get lunches throughout the conference
- get conference bag

We wish to thank you for contributing to the success of the conference and we are looking forward to welcoming you in Barcelona, SPAIN.

Best regards

Assoc. Prof. Dr. Ahmet ESKICUMALI
Coordinator of INTE 2015

MODIFIED EXPLANATION OF JAVA OBJECT CONSTRUCTS HELPING WITH THEIR UNDERSTANDING

Rudolf PECINOVSKÝ, Jarmila PAVLÍČKOVÁ
Department of Information Technologies,
University of Economics, Prague,
Czech Republic
rudolf@pecinovsky.cz, pavjar@vse.cz

ABSTRACT
Most textbooks and courses explain the basic object oriented (OO) constructs in a very similar way. Extensive experience with teaching different kinds of courses at various levels, from primary and secondary schools to universities and retraining courses for professional programmers shows that many students have difficulties with this traditional approach to the explanation of basic OO constructs. The paper shows a little modified approach to this explanation that is based on the *Architecture First* methodology and that may lead to a better understanding of these constructs.

KEYWORDS
OOP, Architecture First, teaching, introductory courses

INTRODUCTION
The Object Oriented Programming (OOP) is a fundamental paradigm of modern programming languages. Over the last 15 years we have been teaching OOP at computer clubs to students from primary schools as well as from high schools, grammar schools and universities. At the same time we have been teaching industry-based courses to retrain professionals from structured programming paradigm to OOP and to improve their knowledge and skills. As a result, we gained the experience with teaching a wide range of students from complete beginners to students with advanced knowledge of programming obtained from textbooks or other courses.
In our university we use the Java language in the introductory courses. Therefore we concentrate in this article on Java courses and textbooks, although we encounter similar problems in textbooks and courses dealing with other programming languages. Often we can use the modifications we suggest for teaching Java in these languages, too.
Both beginners and advanced programmers experience the problems with mastering certain object oriented constructs. We have succeeded in modifying the typical explanation of OO constructs so that the beginners improve their understanding and the advanced programmers learn how to avoid poor programming habits acquired as a result of incorrect understanding of OOP.

Summary of perceived problems
Almost all Java textbooks explain the basic object oriented constructs in the way that is more or less borrowed from older C++ textbooks. However, such explanation involves many definitions that are difficult to understand for the beginners. From their point of view these constructs are often inconsistent and confusing.
In addition to these basic constructs several secondary constructs are explained as the new ones, but using a slightly modified explanation, we can explain the secondary constructs as natural extension of those basic ones. Moreover, we can refer to the constructs that students have already mastered. This slight modification can improve the understanding of both sets of constructs. Furthermore, this decreases the number of problems that the students may encounter when using these constructs in their programs.
Our experience has shown that it is useful to explain the following topics in a slightly different way than textbook authors have done so far:

- what the term *object* means,
- the difference between objects and classes,
- the difference between instances and class members,
- the concept of the `interface`,
- the constructors and the construction of objects,
- the keyword `this`,
- the class inheritance.

In the following sections we deal with each of the above mentioned topics and show the modification of explanation that proved useful in our attempts to improve understanding of the subject matter.
We have compared our method with the following textbooks Arnold, Gosling & Holmes (2005), Barnes & Kölling (2011), Briant (2011), Burd (2014), Deitel & Deitel (2011), Eckel (2006), Fain (2004), Horstman (2007), Horstman & Cornell (2012), Horstman (2013), Liang (2014), Morelli & Walde (2006), Savitch (2011),

Schildt (2011), Schildt (2014), Sierra & Bartes (2009), Winder & Graham (2006), Wu (2009), Zakhour, Kannan & Gallardo (2013) and Zukowski (2002). We can divide these textbooks into three groups:

- Barnes & Kölling (2011), Horstman (2007), Horstman (2013), Sierra & Bartes (2009) and Wu (2009) mainly concentrate on explanation of the best programming practices. They intend to teach how to think and how to program in a *true* object oriented way.
- Arnold, Gosling & Holmes (2005), Eckel (2006), Horstman & Cornell (2012), Schildt (2011), Zukowski (2002) – mainly teaching the language with its APIs. Teaching the art of programming is secondary.
- The remaining texts declare that they teach OOP; however the style of explanation and the discussed topics indicate that they belong to the second group.

We note that the objective of this article is not to review the above mentioned textbooks, but to use them as examples of the traditional way of explanation of object oriented constructs and compare them with the proposed approach. We are not going to enumerate the explanations of the topics in the above mentioned textbooks, but we focus on summarizing these approaches.

EVERYTHING IS AN OBJECT

The textbooks explain the term *object* in two ways:

- most of them explain it using *real* world examples,
- the others do not explain it and assume that it is a generally known term which does not need a special explanation.

In both cases students meet similar problems. Their general understanding is that an *object* is something tangible; they have not come across the idea that objects are used also for representing abstract ideas (e.g. beauty, size, direction, connection, interruption, calculation, etc.). None of the analyzed textbooks use such objects in early examples.

If the students meet such objects in a program for the first time, it takes some time until they accept the fact that abstract ideas can be also represented by objects.

We have discovered that it is useful to explain to students at the very beginning that in object oriented programming everything that can be expressed by a noun, including the abstract terms mentioned above is treated as an object. Some students may be confused by it for a while, and they find it difficult to describe an abstract term by means of an object. Therefore, we explain that in programs each object is represented with a set of data items (attributes) that describes the object. From the program's point of view the object is just this set of attributes and it does not matter whether the set represents a *physical* object or an abstract idea.

Most students quickly understand that besides the attributes that characterize cars, chairs, animals or other physical objects they can equally define attributes that characterize colors, directions, beauty, connections and other abstract terms. To facilitate this understanding, we have to use objects of this kind often at the very beginning of the course. Among suitable candidates for these *abstract* objects are, for example, the characteristics of graphical objects such as colors or directions.

CLASSES VERSUS OBJECTS

In most textbooks the class is explained as an abstraction describing some properties of a group of objects, which we call instances of their parent class. Authors often explain that the class serves as a blueprint or a template for the objects that the program uses. Some authors note that we can look at the class as a factory capable of creating objects on demand.

Students sometimes struggle with understanding the difference between classes and objects, especially when we introduce static attributes and methods. Our experience shows that students understand this topic better, when we explain that classes are also objects (we treat everything that can be named by a noun as an object, therefore the classes should also be treated as objects). Students, which have already their first experience with abstract objects, as a direction, color or beauty, have no significant problem with accepting that the classes are also objects.

This concept is (unintentionally) endorsed also by IDE *BlueJ*, which we use in our introductory programming courses. In *BlueJ* we work with classes and objects in a very similar way. Classes as well as their instances are represented by rectangles, whose context menus display all messages that can be sent to the corresponding object (a class or an instance). The only difference is that classes are shown in the class diagram while instances are shown in the object bench. Thus the students find this explanation consistent with their experience.

We explain that several languages (e.g. Smalltalk) classify the classes as standard objects (and therefore they allow e.g. to save these objects in variables), however it leads to a little complicated architecture of the class hierarchy; and therefore the authors of Java and similar languages implement the classes in a different way, which is much understandable for an ordinary programmer. In these languages the classes are special objects with special properties:

- They are the only objects that can create new objects called instances of their mother class. When advanced students complain that other objects can also create new objects, we explain, that these "other objects" can

only return objects that are originally created by certain class. A new object can be created only by its mother class.

- In Java language and similar languages the classes are the only objects that are not instances of any class. We should address the classes in programs directly by their names; however, we cannot assign a class to a variable.
- Classes are represented by a special kind of objects – the class-objects. In Java the class-objects are instances of the `Class` class. When we want to save the class for certain future use, we should save its class-object. However, we should keep in mind that this class-object is not the class itself; it is just the representative of its class.

It is quite astonishing how this small difference in explanation helps to students to understand the term class and how it helps them to solve some more complex problems.

Introduction of classes as a special kind of objects helps also in the explanation of other topics:

- Students have fewer problems with understanding the difference between the class and the instance members (attributes and methods), and they can use both almost from the beginning of the course.
- Students understand more easily the rules for loading a class by a `ClassLoader` and later on it helps them to understand better the principles of inheritance.

INSTANCE MEMBERS VERSUS CLASS MEMBERS

Within typical style of explanation many students have problems with understanding the difference between the static and instance members. The textbooks, which try to explain not only the syntax and libraries, but also the art of programming, therefore place the explanation of static members often far behind the first introduction of objects and their members.

However, when we utilize the basic rule that the classes are similar objects as their instances, the students have no problems to accept that the classes need also their members – attributes and methods. Here *BlueJ*, which inspects both kinds of objects in the similar way, helps again (see figure 1). Thus we can start to use both kinds of members very soon.

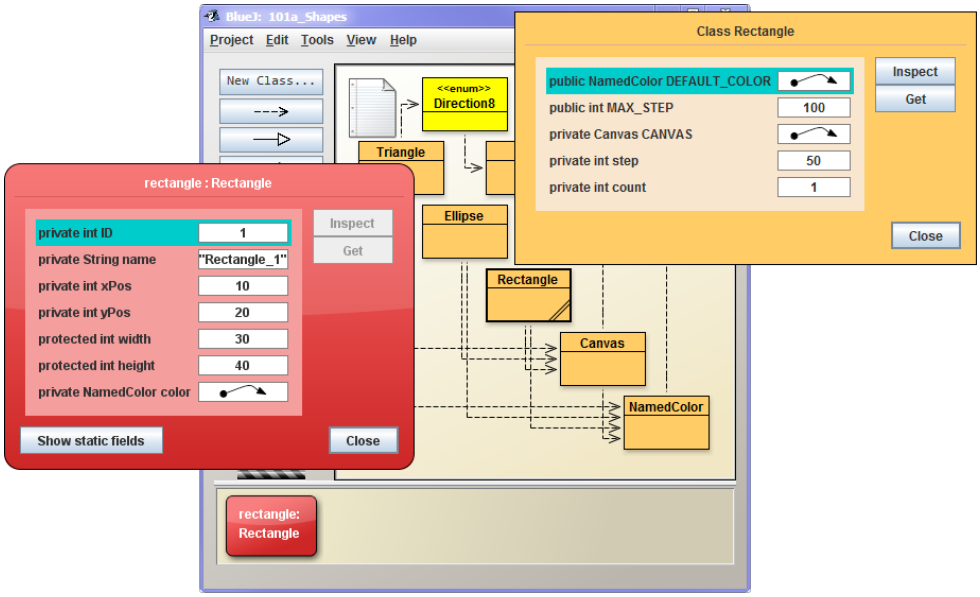

Figure 1: *Inspection of class and its instance*

During our courses, when we leave the interactive mode, where the code is written by the code generator, and start writing the code "by hand" (Pecinovský & Kofránek 2013), it emerged that it is very useful to split up the source code into two parts: the first one engages in the class members and the second one in the instance members.

The standard conventions split up the code into parts engaging in properties (attributes, fields) and behavior (methods) where the class and instance members are often mixed. However, this organization seems to be very confusing for the beginners. The code is much clearer for them, when it is firstly split up into class and instance sections and only then each of these sections is further split up into subparts for fields, constructors, and methods.

INTERFACES AND INTERFACE TYPES

Programming groups' managers often complain that most graduates in software engineering and computer science perfectly know how to implement the given interface; however, they are not able to recognize the situations, where certain interface should be introduced in the developed architecture.

It seems that this is a consequence of too late introduction of this programming construct in the courses. Take into mind several facts:

- The classic book Gamma, Helm, Johnson & Vlissides (2005) together with many others instruct: *"Program to an interface, not an implementation."*
- Java language (and many other languages, too) incorporates the programmatic construct `interface` that can represent the general interface as a kind of the data type in the program.
- The *Early Bird* pedagogical pattern in Bergin (2012) encourages: *"Organize the course so that the most important topics are taught first. Teach the most important material, the "big ideas", first (and often). When this seems impossible, teach the most important material as early as possible."*

Naturally, this leads to a recommendation to incorporate the interface data types and their implementation into the explanation as early as possible. In addition we should explain not only how to implement the interface, but also the reasons why the interfaces are incorporated into the project architecture.

In our courses we explain interfaces together with their purpose and usage in the second lesson, just after the introduction of objects and classes. We also start very soon assigning homework where students should design interfaces of their own to complete the project successfully.

DESIGN PATTERNS EXPLANATION

The next important subjects, we should explain from the very beginning, are the design patterns. (We can again mention the *Early Bird* pedagogical pattern.) We introduce the basic concept in the first lesson and we start using it and teach how (and why) to use it from the second lesson together with the explanation of the interface.

In fact we start to solve our problem with using the *Servant* design pattern and as a consequence the necessity to use (and therefore also to explain) the interfaces.

Starting from the second lesson we introduce the next design pattern (or several design patterns) in almost each lesson, because we need it (them) for solving the current problem.

Thanks to so early introducing and continuous using of design patterns the students accept them as something common and they naturally use them in their programs from the very beginning.

THE KEYWORD/PARAMETER `this`

Another topic whose understanding sometimes causes problems is the keyword `this`. The textbooks differ in its explanation. We can divide these explanations into three groups:

- The explanations from the first group do not mention the meaning of the keyword `this` separately. They only explain what does it mean the expression
  `this.something`
- The explanations from the second group explain that *Java defines* the *keyword `this` that can be used inside any method to refer to the current object* or that it *is useful when you need to refer to the instance of the class from its method* or something similar. Not a mention about the connection of the keyword *`this`* with the method parameters.
- The explanations from the third group explain that beside the explicit parameters there is also the implicit one. Its name is **`this`** and it contains the reference to the invoking object.

Only the third group of explanations describes the real nature of `this` and can fluently continue with explanation that the expression

`instance::method`

determines a lambda expression, of which the first parameter is the method's implicit parameter `this`. The others have to introduce `this` in the role of the first parameter as a new programmatic construction.

CONSTRUCTOR

The next topic causing sometimes problems is the definition and properties of constructors. Almost all textbooks follow the original description published in Strustrup (1991), which says: *"Constructor is identified by having the same name as its class."* The text does not specify if the constructor is or is not a method.

The above mentioned textbooks differ in the explanation of what is a constructor.

- Arnold, Gosling & Holmes (2005) and Horstman (2007, 2013) explain that a constructor is not a method and therefore it may not return anything. Accepting this explanation leads us to assign the responsibility for returning the new object to the `new` "operator"[1].

[1] De iure the `new` is not an operator, however, many teachers and programmers understand it so.

- Others define the constructor as a special kind of a method having the same name as its class. But they still tell nothing about the returning values.

However, nobody explains why the reflection, exceptions and almost all debuggers use the name `<init>` for the constructor. The approaches to this problem can be again divided into three groups:

- The textbooks from the first group do not contain any mention about the possibility to meet the identifier `<init>` by debugging or in stack trace description (these textbooks do not contain any stack trace with this identifier).
- The textbooks from the second group show this identifier in some stack trace description, but they ignore its strangeness and do not contain any mention what the strange identifier is like.
- The textbooks from the third group reveal, that in debuggers and the stack trace descriptions the constructors are named **`<init>`**, but they do not discuss the difference between this name and the name introduced in the first explanation of constructors.

When we look at the constructor syntax we can interpret it in two ways:

- as the method identified by its class name and declaring no return type,
- as the method identified by the empty string and declaring its class as its return type.

Taking the constructor as the method with the name equal to its class and not declaring the type of its return value, we should introduce a new construct `this(…)` serving for invoking another constructor and transfer the responsibility for the initialization of the created object to it.

On the contrary when we use the explanation that the constructor is a method with an empty name, we can explain the construct `this(…)` as a natural extension of the known rules. This explanation is closer to the actual implementation. Therefore it appears that it is more efficient to explain constructors in this way and explain them as the methods with special properties:

- In Java the internal name of constructors is `<init>`. However, this name (deliberately) violates the rules for identifiers and therefore in the source code the constructors are declared as methods, the names of which are empty strings (methods with empty names).
- The constructor must return a reference to the newly created object. This reference is obtained from the hidden parameter `this`, which is initialized by the caller. Because the returned value is known *a priori*, the language syntax theoretically does not need the statement

  ```
  return this;
  ```

  In fact, we may not write it, it is inserted by the compiler on our behalf to prevent mistakes.
- The constructor can be used only for the initialization of the newly allocated memory and therefore it can be invoked only by the `new` "operator" or by another constructor of the same class. In the latter case the invocation statement must be the very first statement of the calling constructor body to ensure that the memory is not yet initialized.

After the above explanation the students understand the explanation of the following syntactic rules much better. We explain:

- Constructing of a new object proceeds in two steps:
  - Firstly, the `new` "operator" is called with a parameter defining the name of the class, whose instance we want to create (the parent class). This parameter determines the size of the memory allocated for the created object and it also specifies other information needed for creating the object (e.g. the reference to the mother class). Additionally, the allocated memory is filled with zeroes and/or compile-time constants.
  - Secondly, the "empty-string" method (constructor) is invoked with the argument `this` pointing to the allocated memory and possibly also with other arguments. The constructors' task is to initialize this memory so that it would correctly represent the object.

  We can outline the described behavior by writing the statement in two lines:

  ```
  new ClassName     //Invoking the new "operator", memory allocation
  (/*parameters*/); //Invoking the contructor
  ```
- As we have noted, the constructor can be used only for initialization of the newly allocated memory. If it is invoked by another constructor, this invocation must be really the first statement in its body. Nothing may precede it, nor an opening brace.
- If the constructor delegates its responsibility for initializing the object to another constructor, it should qualify this invocation by `this` as we are used to do with normal methods. However, in this case we do not write the *dot*. Thus, instead of writing

  ```
  this.(/*parameters*/);
  ```

  (we know, that constructor's source code name is the empty string) we write only

  ```
  this(/*parameters*/);
  ```

When we explain constructors in this way, the students more easily understand the statement `this(…)` as a tool for delegating the responsibility for initializing the object and they also understand what the `<init>` appearing in exception messages or debugger windows means.

This explanation establishes a good basis for the following explanation of static initializers and invocation of `super` constructor. Everything fits logically together.
When the above explanation is used the students sometimes complain that the object is not created by the constructor but by the `new` "operator". We can use the following analogy: "Who makes cups?" They answer: "A potter." Then we explain that the allocated memory serves similarly as potter's clay and that the constructor processes this memory similarly as the potter processes the clay. Using this analogy, we regard the constructor as the author of the created object.

INHERITANCE
The most common problem with teaching inheritance is that it is taught too early. Some textbooks deal with it immediately after the first introduction of objects and classes.
For now we ignore the rule that if we want the students to acquire the knowledge of the OO paradigm well, we should not explain the concept of inheritance until we explain the concept of interface (a general interface as well as the Java construct `interface`). These problems are discussed in Pecinovský, Pavlíčková & Pavlíček (2006), Pecinovský & Pavlíčková (2007) and Pecinovský (2009a) during explaining the *Design Patterns First* methodology, which was the precursor of the *Architecture First* methodology used now. From the textbooks mentioned at the beginning of this paper only Horstman (2007) explains the `interface` before the inheritance.
When explaining the inheritance all the above mentioned textbooks clarify that a child should represent a special kind of its parent. However, particular textbooks do not put the same emphasis on it. Mostly, they mention this rule only at the beginning of the explanation of inheritance and then they show only how we could use the inheritance to avoid writing an additional code. Unfortunately, the majority of programming textbooks do not present bad examples of inheritance usage at all. This would warn the reader of using a bad design in proper time.
After such explanation the students often remember only that the inheritance serves for reusing the code and they also use it only for this purpose.
We discover that the inheritance should be explained in two phases: firstly the inheritance of interfaces, and only much later the inheritance of the implementation.
When touching the inheritance for the first time, we should explain that there are three kinds of inheritance (Lalond & Pugh (1991)):

- **Inheritance of interface** (in Lalond & Pugh (1991) *subtyping*) occurs when a child inherits the entire interface from its parent, i.e. the signature as well as the contract. As a consequence, an instance of a subtype can stand in for an instance of its supertype.
  However, a compiler ensures the inheritance of the signature only. Maintaining the contract is the programmer's job. Subtype implementation details are totally irrelevant for it; all that matters is that it has the right behavior so that the parent can be formally substituted by the child.
- **Inheritance of implementation** (in Lalond & Pugh (1991) *subclassing*) – it is an implementation mechanism for sharing both code and representation. The subclass inherits all the implementation from its superclass (it is the compiler's job). The subclass can change the behavior that does not fit its requirements, and it can also add new members. Here, the danger is that the overriding code and/or the new members violate the parent's contract.
- **Natively understood inheritance** (in Lalond & Pugh (1991) the "*is-a" relationship*) talks about our assumption that one kind of object is a special case of another. An inconsistence may appear at this place when the implementation differs from our inherent assumption. E.g. mathematicians tell us, that a map is a special kind of a set – it is a set of ordered pairs (key, value). However, in the Java standard library the set is implemented as a special kind of a map.

The experience proved that we should postpone the explanation of class inheritance as late as possible. The reason for postponing this explanation is to provide enough time for exercising the usage of interfaces. The students should learn not only how to implement the given interface, but they also should master how to recognize situations, in which incorporating an interface in their design is useful.
At this point it is helpful to introduce the *Decorator* design pattern and to prepare at least one project, where using this pattern is more effective than the frequently (and improperly) used inheritance of classes. There are two reasons why to introduce this pattern:

- The advanced students who mastered inheritance in a previous course (or from a textbook) are provided with situations where the class inheritance is not the best solution. It also helps to improve students' attention to the ongoing explanation.
- We prepare the background for the following explanation of class inheritance.

If our lessons follow the *Design Patterns First* methodology (Pecinovský & Kofránek 2013), an introduction of the *Decorator* design pattern does not present a problem since the students already know several design patterns and they understand their importance.

As the next step we remind to students that in addition to the heretofore used inheritance of `interfaces` (the language construct) there is also the class inheritance. This inheritance combines the inheritance of the parent class interface with the inheritance of the parent implementation. We explain that the inheritance of implementation is internally handled as if the subclass were designed according to the *Decorator* design pattern. In other words, the inheritance of implementation is *de facto* an application of the *Decorator* design pattern in which the decorator (a child) acquires both the implementation and the interface from the decorated object (a parent). The compiler prepares a hidden constant attribute named `super`, in which a reference to the *decorated object* is held. Additionally, the compiler also ensures the automatic delegation of all inherited methods to the `super`.

For the decorated "`super`" object we introduce the term *parent subobject*. In contrast to the standard decorator the constructor of a child does not take its parent (`super`) as a parameter, but it creates the parent subobject by calling a parent's "empty-string" method (a constructor):

```
super (/* parameters */);
```

where, similarly to the statement `this()`, we omit the *dot*.

We explain that the parent subobject must be created before the rest of the child object is initialized to allow the rest using the inherited members. Thus the child constructor has two alternatives:

- delegating its responsibility for initialization to one of its peers by the statement `this()` or
- starting with creating the parent object, i.e. calling its constructor by the statement `super()`.

The only exception is the situation, when we want to call the parameter-less parent constructor – then the compiler is able to insert its call for us behind the scene.

So far we did not create the parent subobject in our classes explicitly, because the compiler implicitly has used the parameter-less parent constructor. We may immediately show, that identical behavior can be obtained by adding the `super();` statement into our original classes.

Our experience shows that the explanation that follows these rules is much comprehensible for the students than using the traditional approach. Especially, the concept of overriding, which was difficult to understand for many students, is now clear and intelligible for most of them. Several programmers attending our retraining courses have commented that thanks to this explanation they finally fully understand the class inheritance.

We should not forget to remind to students that the three kinds of inheritance must not be interfered. They should fit together. In case of class inheritance, the compiler is able to ensure only the inheritance of the implementation and the signature. The inheritance of the contract is the responsibility of the programmer.

RESULTS AFTER APPLICATION OF THE PRESENTED SUGGESTIONS

This methodology has been tested in several student groups whilst in other groups the introductory programming course was running in the classic way. In the succeeding semester the students of these groups jumbled with the students of other groups when enrolling for new subjects. After passing another semester, all students have been asked to fulfill anonymous web questionnaire. 81 students were willing to fill out the questionnaire which was more than half of the questioned persons.

Inside the preliminary questions there was the question asking for the attended course. According replies to this question we can divide the answering students into three groups:

- The students from the first group attended the courses using the above described methodology and continued in programming courses, where they could meet with students from other courses or even cooperate with them. We will mark this group as `1-1`.
- The students from the second group did not attend the above described courses; however, they meet the students from these courses in further semester and cooperated with them. We will mark this group as `0-1`.
- The students from the third group attended the courses using the above described methodology, but did not continue with programming courses and compared their knowledge and skills with their colleagues in their companies. We will mark this group as `1-0`.

The questionnaire contained 12 questions. Three of them touched evaluation of the described methodology. Answers to these questions are in Tables 1 to 3.

The Table 1 contains answers to the question "*Do you think this kind of explanation can help to students to better understand the relation between the developed program and the simulated reality?*"

Table 1:

	1-1	0-1	1-0	Sum
It will harm very much	0%	0%	0%	0%
It will rather harm	0%	0%	14%	4%
It will neither help, nor harm	11%	17%	14%	12%
It will help a bit	62%	67%	45%	58%

It will help very much	17%	17%	23%	19%
Hardly to assess	9%	0%	5%	7%

The Table 2 contains answers to the question "*According to your opinion: Compared to other groups and with regards to the used concept of teaching did the students from the selected groups learn:*"

Table 2:

	1-1	0-1	1-0	Sum
Considerably less	0%	0%	5%	1%
Rather less	9%	17%	14%	11%
Approximately equally	8%	0%	27%	12%
Rather more	32%	33%	14%	27%
Substantially more	38%	33%	5%	28%
Hardly to assess	13%	17%	36%	20%

The Table 3 contains answers to the question "*Compared to students who attended courses of programming in the classic way, is your ability to formalize the handled problem and design the corresponding architecture of the developed program:*"

Table 3:

	1-1	0-1	1-0	Sum
Considerably smaller	2%	0%	9%	4%
Rather smaller	0%	0%	5%	1%
Approximately equal	13%	50%	32%	21%
Rather bigger	49%	33%	27%	42%
Considerably bigger	23%	0%	0%	15%
Hardly to assess	13%	17%	27%	17%

SUMMARY

This paper was written in response to problems that many students have experienced with understanding the object oriented concepts. It shows that by changing the way of explaining certain OO specific constructs we can improve the comprehensibility of these concepts.

It recommends the use of objects that represent abstract concepts from the very beginning of explanation. Subsequently, the class should be explained as a special kind of object with special features – e.g. that it is the only object that can create new objects – its instances. By writing the source code of the class, it is very useful to strongly separate the definitions of the class members and the instance members.

Further, it recommends to explain the concept of interfaces as the kind of data type at the beginning of the course to allow students to learn not only how to implement them, but also how to incorporate them into the design of the new project. The introduction of interfaces allows early incorporation of design patterns into explanation.

Next it suggests revealing that `this` is an implicit parameter of constructors and instance methods to make the follow-up explanations easier and more logical. Similarly it suggests explaining the constructor as a method whose name is an empty string and which can be used only for initializing a newly allocated memory. It shows how this change makes some other constructs more logical.

Special attention is paid to inheritance. It suggests postponing the explanation of class inheritance far after the explanation of `interface`, and simultaneously preceding it by the explanation of the *Decorator* design pattern. The knowledge of *Decorator* design pattern facilitates understanding of the concept of class inheritance. In addition, the paper recommends explaining the three kinds of inheritance and emphasizing that the compiler ensures only the inheritance of signature, while providing the correct inheritance of the contract is the programmer's responsibility.

Finally, the paper shows that students appreciate the modified explanation and most of them feel that this modified methodology mediates them a better knowledge of OOP paradigm and thus also a better base for the program design.

ACKNOWLEDGEMENTS

This paper was processed with contribution of long term institutional support of research activities by Faculty of Informatics and Statistics, University of Economics, Prague.

REFERENCES

Arnold K., Gosling J. & Holmes D. (2005). *The Java™ Programming Language, Fourth Edition*. Addison Wesley Professional. ISBN 0-321-34980-6.

Barnes D. & Kölling M. (2011). *Objects First With Java: A Practical Introduction Using BlueJ (5th Edition).* Prentice Hall. ISBN 978-0-132-49266-9.

Bergin, J. (2012). *Pedagogical Patterns: Advice For Educators*. CreateSpace Independent Publishing Platform. ISBN 1-4791-7182-4.

Briant J. (2011) *Java 7 for Absolute Beginners.* Apress. ISBN 978-1-4302-3687-0

Buchalcevová A. (2008) Buchalcevová, Alena. Where in the curriculum is the right place for teaching agile methods? Proceedings 6th ACIS International Conference on Software Engineering Research, Management & Applications (SERA 2008). Prague : Copyright, 2008, p. 205–209. ISBN 978-0-7695-3302-5.

Burd B. (2014) *Java for Dummies 6th Edition.* John Wiley & Sons, Inc., ISBN 978-1-118-41764-5.

Deitel H. M. & Deitel P. J. (2011). *Java How to Program, 9th Edition.* Prentice Hall, ISBN 978-0-13-257566-3.

Eckel B. (2006). *Thinking in Java (4th Edition).* Prentice Hall, ISBN 0-13-187248-6.

Fain Y. (2004). *Java Programming for Kids, Parents and Grandparents.* Smart Data Processing. ISBN 0-9718439-5-3. Available from: www.csd.abdn.ac.uk/~tnorman/teaching/CS1014/information/JavaKid8x11.pdf

Gamma E., Helm R., Johnson R., Vlissides, J. (2005) *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley. ISBN 0-201-30998-X.

Gosling J., Joy B., Steele G., Bracha G. & Buckley A. (2014). *The Java Language Specification, Java SE 8 Edition (Java Series).* Addison-Wesley Professional. ISBN 978-0-13-390069-9. Available from: docs.oracle.com/javase/specs/jls/se8/jls8.pdf

Horstman C. S. (2007). *Java Concepts for Java 5 and 6.* John Wiley and Sons. ISBN 978-0-470-10555-9.

Horstman C. S. & Cornell G. (2012). *Core Java™, Volume I – Fundamentals (9th Edition).* Prentice Hall PTR. ISBN 978-0-13-708189-9

Horstman C. S. (2013). *Big Java: Early Objects (5th Edition)*. John Wiley and Sons. ISBN: 978-0-470-10554-2.

Horton I. (2002). *Beginning Java 2.* Wrox. ISBN 978-0-76454-365-4.

Lalond W. & Pugh J. (1991). *Subclassing ≠ Subtyping ≠ IsA*. Journal of Object-Oriented Progrtamming. Vol. 3, No. 5.

Liang Y. D. (2014). *Introduction to Java Programming: Comprehensive Version (10th Edition).* Prentice Hall. ISBN 978-0-13-376131-3.

Morelli R. & Walde R. (2006). *Java, Java, Java, Object-Oriented Problem Solving (3rd Edition).* Prentice Hall, ISBN 978-0-131-47434-5.

Pecinovský R. (2009a). *Early Introduction of Inheritance Considered Harmful*. Objekty, Hradec Králové.

Pecinovský R. (2009b). *Using the methodology Design Patterns First by prototype testing with a user.* Proceedings of IMEM, Spišská Kapitula.

Pecinovský R. (2013). *OOP - Learn Object Oriented Thinking & Programming*. Eva & Tomas Bruckner Publishing. ISBN 978-80-904661-9-7. Available from: pub.bruckner.cz/titles/oop

Pecinovský, R. & Kofránek, J. 2013. The Experience with After-School Teaching of Programming for Parents and Their Children. *In Proceeding of the 2013 International Conference on Frontiers in Education: Computer Science and Computer Engineering.* (FECS'13) Las Vegas. http://edu.pecinovsky.cz/papers/2013_WC_Our_Experience_Teaching_After-School_Programming.pdf [Pecinovský and Kofránek 2013]

Pecinovský R. & Pavlíčková J. (2007) *Order of explanation should be Interface – Abstract classes – Overriding.* Proceedings of 12th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'2007). Dundee, ACM Press. Available from: vyuka.pecinovsky.cz

Pecinovský R., Pavlíčková J. & Pavlíček L. (2006). *Let's Modify the Objects-First Approach into Design-Patterns-First*. Proceedings of 11th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'2006). Bologna, ACM Press, ISBN 1-59593-346-8.

Savitch W. (2011) *Java: An Introduction to Problem Solving and Programming (6th Edition).* Addison-Wesley. ISBN 978-0-13-21627-0.

Schildt H. (2011). *Java: The Complete Reference, Eighth Edition*. McGraw-Hill Osborne Media. ISBN 978-0-07-160630-1.

Schildt H. (2014). *Java: A Beginner's Guide, Sixth Edition*. McGraw-Hill Osborne Media. ISBN 0-07-180925-2

Sierra K. & Bartes B. (2009). *Head First Java, 2nd Edition*. O'Reilly Media. ISBN 978-0-596-00920-5.

Strustrup B. (1991). *The C++ Programming Language, 2nd Edition.* Addison-Wesley Publishing Company. ISBN 0-201-53992-6.

Winder R. & Graham R. (2006). *Developing Java Software, 3rd edition*. John Willey & Sons, Ltd. ISBN 0-470-09025-1.

Wu C. T. (2009) *An Introduction to Object-Oriented Programming with Java. Fifth Edition*. McGraw-Hill Science/Engineering/Math. ISBN 978-0-07-352330-9.

Zakhour S., Kannan S. & Gallardo R. (2013). *The Java Tutorial: A Short Course on the Basics (5th Edition) (Java Series)* Prentice Hall PTR. ISBN 978-0-321-33420-6.

Zukowski J. (2002). *Mastering Java 2.* Sybex. ISBN 978-0-7821-4022-4.