How to improve understanding of OOP constructs

Rudolf Pecinovský
University of Economics Prague
Department of Information Technologies
Churchill sq. 4, 130 00 Prague 3
Czech Republic
Phone: +420 603 330 090
Email: rudolf@pecinovsky.cz

Jiří Kofránek
Charles University in Prague
Laboratory of Biocybernetics
Department of Pathophysiology
U nemocnice 5, 128 52 Prague 2
Czech Republic
Phone: +420 777 686 868
Email: kofranek@gmail.cz

Abstract—**Most textbooks and courses explain 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 school through to university and requalification courses for professional programmers shows that many students have a difficulty with this traditional approach. In this paper we show that a modified approach according to Architecture First methodology leads to a better understanding of the basic OO constructs.**

I. Introduction

OBJECT Oriented Programming (OOP) is a fundamental paradigm of modern programming languages. Over the last 10 years we have been teaching OOP at computer clubs for students from primary schools as well as at 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 have experience with teaching a range of students from complete beginners to students with advanced knowledge of programming obtained from textbooks or other courses.

Both beginners and advanced programmers experience problems with mastering certain object oriented constructs. We have succeeded in modifying the explanation of OO constructs so that beginners improve their understanding and advanced programmers learn to avoid poor programming habits acquired as a result of incorrect understanding of OOP.

A. Summary of perceived problems

Almost all textbooks explain the basic object oriented constructs in a way that is more or less borrowed from older C++ textbooks. However, such an 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 new, but using a slightly modified explanation, we can explain the secondary constructs as natural extension of the 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.

This paper describes the outcome of research that has been accomplished as part of research program funded by Grant Agency of the Czech Republic Grant No. GACR P403-10-0092 and by the grant FR—TI3/869 and by the sponsorship of the ICZ a. s. company.

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 is an object,
- the difference between objects and classes,
- the concept of the `interface`,
- constructors and construction of objects,
- the keyword this,
- 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 textbooks [1], [2], [4], [5], [6], [8], [9], [10], [11], [15], [16], [22], [23], [24], [26], [27] and [28]. We can divide these textbooks into three groups:

- [2], [9], [10] and [24] – 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.
- [1], [5], [8], [11], [22], [28] – mainly teaching the language with its APIs. Teaching the art of programming is secondary.
- Remaining texts claim 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 compared it 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.

II. Everything is an object

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

- most of them explain it using ***real*** world examples,
- 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 *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.). If they meet such objects in a program for the first time, it will take some time until they accept the fact that abstract ideas can be also represented by objects.

A. Recommendation

We have discovered that it is useful to explain to students at the very beginning that in object oriented programming we treat as an object everything that can be expressed by a noun, including the abstract terms mentioned above. 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 here 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 from the very beginning of the course. Among suitable candidates for these *abstract* objects are, for example characteristics of graphical objects such as colors or directions.

III. 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 a class serves as a blueprint or a template for the objects that the program uses. Some authors note that we can look at a 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.

A. Recommendation

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 classes should also be treated as objects). However, classes are special objects with special properties:

- They are the only objects that can create new objects called instances of their parent 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 a class. A new object can be created only by its parent class.
- Classes define two types of attributes (fields):
  - First type of attributes is marked with the modifier `static`. We can interpret static attributes as attributes that do not move and stay "statically" in their class.
  - The attributes that are not marked with the modifier `static`, declare attributes for instances of the class. Each created instance takes its own copy of these attributes.
- Similarly, classes define two types of methods:
  - Methods marked with the modifier `static` belong to the class and can use directly only members (attributes and methods) of their class. Members of instances and other classes should be qualified by their owner.
  - Other methods belong to instances. They have (similarly to constructors) a hidden constant parameter `this`, which is initialized by a reference to their instance. Although the parameter `this` is not included in the list of parameters, it can be used in the body of a method.

    These instance methods can use the members of their instance and its class directly. (We observe that the class behaves as if it were a proper parent and allows all of its instances to use its (i.e. static) attributes and methods.) Other members should be qualified by their owner.

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

This explanation 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 class/object. The only difference is that classes are shown in the class diagram while instances are shown in the object bench. Thus students find this explanation consistent with their experience.

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

- Students have no problems with understanding the difference between class and instance 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 it helps them later to understand better the principles of inheritance.

IV. Constructor and parameter `this`

Another topic whose understanding and use sometimes causes problems are constructors and the keyword `this`. Almost all textbooks follow the original description published in [25], which says: *“Constructor is identified by having the same name as its class.”* The text does not differentiate if the constructor is a method.

The above mentioned textbooks differ in the explanation of what is a constructor. [1], [9] and [10] 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].

Most authors define a constructor as a special kind of a method having the same name as its class. However, most of them do not explain why the reflection, exceptions and almost all debuggers use for constructor the name `<init>`. They either ignore it (e.g. [4], on page 927) or they do not discuss the difference between this name and the name introduced at the beginning of the course (e.g. [8]).

Taking a constructor as a method with the name of 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.

A. Recommendation

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

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

The latter explanation is closer to the actual implementation. It appears that it is more efficient to explain constructors in this way and explain them as methods with special properties:

- In Java the internal name of constructors is `<init>`. However, this name violates the rules for identifiers and therefore Java authors decided to declare constructors in source code as “empty-string” methods, or more precisely as methods, whose names are empty strings.
- A 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 priory***, the language syntax theoretically does not need the statement `return this;` In fact, we do not write it, it is inserted by the compiler on our behalf to prevent mistakes.

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

- Construction of a new object proceeds in two steps:
  - First, 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 the creation of the object (e.g. the address of the VMT). Additionally, the allocated memory is filled with zeroes and/or compile-time constants.
  - Second, the “empty-string” method (constructor) is invoked with argument `this` pointing to the allocated memory and possibly also with other arguments. The constructors’ task is initializing this memory so that it correctly represents the object.

  We can outline the described behavior by writing the statement in two lines (here, due to narrow columns, they are four):

  ```
  new ClassName
  //Invoking new - 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 the very first statement in its body. Nothing may precede it, not even an opening brace.
- If a constructor delegates its responsibility for initializing the object to another constructor, it should qualify this invocation by `this` as we are used to doing with normal methods. However, in this case we do not write the ***dot***. So instead writing
  ```
  this.(/*parameters*/);
  ```
  we write only
  ```
  this(/*parameters*/);
  ```

When we explain constructors in this way, students more easily understand the `this()` statement as a means 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 students sometimes complain that the object is not created by the constructor but by the `new` “operator”. Here 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 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.

V. 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 to objects and classes.

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

For now we ignore that if we want 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`). In addition, we should not only teach how to implement an `interface`, but also how to incorporate it in our design. These problems are discussed in [17], [18] and [19] during explanation of the *Design Patterns First* methodology (predecessor of the *Architecture First* methodology). From the textbooks mentioned at the beginning of this paper only [9] and [27] first explain the `interface` and later the inheritance.

When explaining inheritance all the mentioned textbooks explain that a child should represent a special kind of its parent. However, they 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 additional code. Unfortunately, the majority of programming textbooks do not present bad examples of inheritance usage at all. This would warn the reader against a bad design early.

After such an explanation the students often remember only that inheritance serves primarily for reusing code and they also use it only for this purpose.

A. Three kinds of inheritance

At the beginning of explaining class inheritance we should introduce the three kinds of inheritance ([14]):

- ***Inheritance of interface*** (in [14] ***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; all that matters is that it has the right behavior so that it can be substituted.
- ***Inheritance of implementation*** (in [14] ***subclassing***) – it is an implementation mechanism for sharing 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 overridden code and/or new members violate the parent's contract.
- ***Natively understood inheritance*** (in [14] the "***is-a" relationship***) talks about our assumption that one kind of object is a special case of another. Here an inconsistence may appear 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.

B. Recommendation

There are two recommendations:

1) Postpone the explanation of class inheritance as late as possible

The reason for postponing this explanation is to offer enough time for exercising usage of interfaces. Students should learn not only how to implement a given interface, but they also should master how to recognize situations, where incorporating an interface in their design is useful.

At this point it is useful to introduce the ***Decorator*** design pattern and prepare at least one project, where using this pattern is more useful than the frequently (and improperly) used inheritance of classes. There are two reasons why to introduce this pattern:

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

If our lessons follow the *Architecture First* methodology ([17], [18] and [19]), 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.

2) Explain class inheritance as an automated implementation of the Decorator design pattern

As the next step we inform students that in addition to the inheritance of `interface` (languages construct) there is also 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 (child) acquires both the implementation and the interface from the decorated object (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 `super`.

For the decorated "`super`" object we introduce the term ***parent subobject***. In contrast to the standard decorator a 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 to use the inherited members. So the child constructor has two options:

- to delegate its responsibility for initialization to one of its peers by the statement `this()` or

- to start 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 to create the parent object in our classes explicitly, because the compiler implicitly 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 following these rules is much more 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. 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 students that the three kinds of inheritance must not interfere. 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.

VI. Design of algorithms

In the OOP era of the present times, it tends to be forgotten sometimes that the design of rather complex algorithms may be ahead of us at the end of object analysis. Several graphical languages are available for their representation. The most often used are flowcharts, UML activity diagrams and Nassi-Schneiderman diagrams. However all of them have some drawbacks. Flowcharts and UML activity diagrams do not force using of structured algorithmic constructs. Nassi-Schneiderman diagrams use oblique lines, which degenerate the space for conditions of condition statements.

Our experiences proved as the most effective graphical tool ***kopenograms*** ([12], [13]) – a handy tool for clear graphical representation of the structure of algorithms. They have found long-term application particularly in teaching programming classes. These are a convenient supplement of UML diagrams used to represent algorithmic structures.

A. Recommendation

Use kopenograms as the graphical tool by explanation of complicated algorithms and algorithmic constructs.

VII. Results after application of the presented suggestions

In the first semester all students at Department of Information Technologies in University of Economics have mandatory lessons on *Fundamentals of programming*. The students' average results are not excellent because most of them tend to study IT management and they take the mandatory programming as an inevitable duty. Therefore they do not want to task their mind with an intensive thinking about this topic and they try to find out simple and straightforward guidelines for solving their assignments.

Tables 1 and 2 show how the students in groups of the first author were successful in past years. The first table shows all students, who began to study in the given year. Table 2 does not involve students, who found out the school too difficult and left it. So the results in the second table are a little higher, however we have started to watch this statistic since 2008.

Row headers show start time of a lesson of a particular group. In 2005, when the first author started to teach at the faculty (before he had presented mainly to professional programmers), he had only one group. In following years he presented to three to four groups. After the first year we became frightened of the low success and lowered our demands, however later on we realized it was not the right way and we started to improve our methodology.

As we have said the students' results are not excellent – it oscillated around 50 %. However, after introducing of the described modifications the results significantly increased without reducing the demands. It increased to 75 % and if we do not involve the students, who left the school, it increased up to 83 %. In the next year this level was kept. In addition we observed that these groups' students obtained better skills than graduates of other groups. However this phenomenon was not statistically processed.

VIII. 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 these 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.

Further, it recommends 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 constructs more logical.

In the next chapter it concentrates on inheritance. It suggests postponing the explanation of class inheritance after the explanation of `interface`, and simultaneously preceding it by the explanation of the *Decorator* design pattern. The *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 ensuring the correct inheritance of the contract is the programmer's responsibility.

Finally the paper shows that by incorporating the suggestions into the explanation the students' results in the observed groups significantly increased. This statistic corroborates my previous feeling from the courses, where the methodology was tested.

TABLE 1:
AVERAGE RESULTS – ALL STUDENTS

	2005	2006	2007	2008	2009	2010	2011
07:30	53%	61%	26%	53%	20%	80%	100%
09:15		65%	45%	65%	47%	70%	64%
11:00		68%	47%	28%	50%	84%	80%
12:45			58%		61%	68%	57%
Average	53%	65%	44%	49%	48%	75%	76%

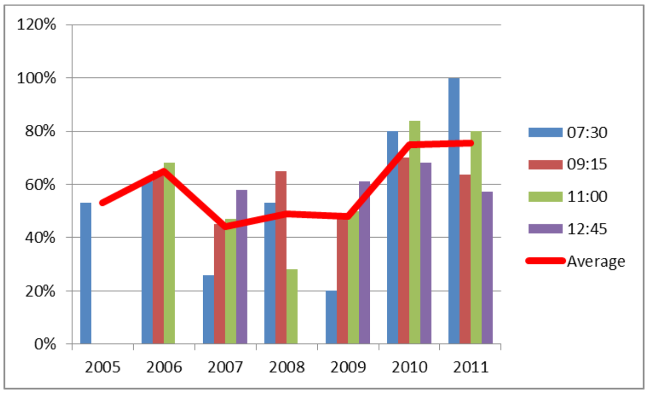

Fig. 1: Average results – all students

TABLE 2:
AVERAGE RESULTS WITHOUT STUDENTS LEFT THE SCHOOL

	2008	2009	2010	2010
07:30	56%	25%	86%	100%
09:15	72%	47%	88%	64%
11:00	29%	50%	84%	89%
12:45		65%	76%	57%
Average	53%	50%	83%	78%

ACKNOWLEDGMENT

This paper describes the outcome of research that has been accomplished as part of research program funded by Grant Agency of the Czech Republic Grant No. GACR P403-10-0092 and by the grant FR—TI3/869. Authors also thank for significant sponsorship the ICZ a. s. company.

REFERENCES

[1] Arnold K., Gosling J., and Holmes D. 2005. *The Java™ Programming Language, Fourth Edition*. Addison Wesley Professional. ISBN 0-321-34980-6.
[2] Barnes D. & Kölling M. 2004. *Objects First With Java: A Practical Introduction Using BlueJ (2nd Edition)*. Prentice Hall. ISBN 978-0-131-24933-2.
[3] 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.
[4] Deitel H. M. & Deitel P. J. 2007. *Java How to Program, 7th Edition*. Prentice Hall, ISBN 978-0-132-22220-4.
[5] Eckel B. 2007. *Thinking in Java (3rd Edition)*. Prentice Hall, ISBN 978-0-131-00287-2-6.
[6] Fain Y. 2004. *Java Programming for Kids, Parents and Grandparents*. Smart Data Processing. ISBN 0-9718439-5-3. DOI= http://www.csd.abdn.ac.uk/~tnorman/teaching/CS1014/information/JavaKid8x11.pdf
[7] Gosling J., Joy B., Steele G. & Bracha G. 2005. *Java™ Language Specification, Third Edition*. Addison Wesley. ISBN 978-0-321-24678-3. DOI= http://java.sun.com/docs/books/jls/download/langspec-3.0.pdf
[8] Horstman C. S. & Cornell G. 2007. *Core Java™, Volume I – Fundamentals (8th Edition)*. Prentice Hall PTR. ISBN 978-0-132-35476-9.
[9] Horstman C. S. 2007a. *Big Java (3rd Edition)*. John Wiley and Sons. ISBN: 978-0-470-10554-2.
[10] Horstman C. S. 2007b. *Java Concepts for Java 5 and 6*. John Wiley and Sons. ISBN 978-0-470-10555-9.
[11] Horton I. 2002. *Beginning Java 2*. Wrox. ISBN 978-0-76454-365-4.
[12] Kofránek J., Pecinovský R., Novák P. 2012. "Kopenograms – Graphical Language for Structured Algorithms." *Proceedings of the 2012 International Conference on Foundation of Computer Science. WorldComp 2012* Las Vegas. CSREA Press. ISBN 1-60132-211-9.
[13] *Kopenogram web page*. http://www.kopenogram.org.
[14] Lalond W. & Pugh J. 1991. *Subclassing ≠ Subtyping ≠ IsA*. Journal of Object-Oriented Progrtamming. Vol. 3, No. 5.
[15] Liang Y. D. 2006. *Introduction to Java Programming: Comprehensive Version (6th Edition)*. Prentice Hall. ISBN 978-0-132-22158-0.
[16] Morelli R. & Walde R. 2006. *Java, Java, Java, Object-Oriented Problem Solving (3rd Edition)*. Prentice Hall, ISBN 978-0-131-47434-5.
[17] 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. DOI = http://edu.pecinovsky.cz/papers/
[18] 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. ISBN: 978-1-59593-610-3. DOI = http://edu.pecinovsky.cz/papers/
[19] Pecinovský R. 2009a. "Early Introduction of Inheritance Considered Harmful." *Proceedings of Objects 2009*, Hradec Králové. ISBN 978-80-7435-009-2. DOI = http://edu.pecinovsky.cz/papers/
[20] Pecinovský R. 2009b. "Using the methodology Design Patterns First by prototype testing with a user." *Proceedings of IMEM 2009*, Spišská Kapitula. DOI = http://edu.pecinovsky.cz/papers/
[21] Pecinovský R. 2010. *Learn Object Oriented Thinking and Programming*. – Translation of the original book: *OOP – Naučte se myslet a programovat objektově*. Computer Press 2010. ISBN 978-80-251-2126-9. English version to be published.
[22] Schildt H. 2004. *Java: The Complete Reference, J2SE 5 Edition*. McGraw-Hill Osborne Media. ISBN 978-0-07-223073-4.
[23] Schildt H. 2005. *Java: A Beginner's Guide, Third Edition*. McGraw-Hill Osborne Media. ISBN 0-07-223189-0.
[24] Sierra K. & Bartes B. 2009. *Head First Java, 2nd Edition*. O'Reilly Media. ISBN 978-0-596-00920-5.
[25] Strustrup B. 1991. *The C++ Programming Language, 2nd Edition*. Addison-Wesley Publishing Company. ISBN 0-201-53992-6.
[26] Winder R. & Graham R. 2006. *Developing Java Software, 3rd edition*. John Willey & Sons, Ltd. ISBN 0-470-09025-1.
[27] Zakhour S., Hommel S., Royal J., Rabinovitch I., Risser T. & Hoeber M. 2006. *The Java Tutorial: A Short Course on the Basics, 4th Edition*. Prentice Hall PTR. ISBN 978-0-321-33420-6.
[28] Zukowski J. 2002. *Mastering Java 2*. Sybex. ISBN 978-0-7821-4022-4.