Informatics in Education, 2022, Vol. 21, No. 2, 331–352

DOI: 10.15388/infedu.2022.10

The Application of Higher-Order Cognitive Thinking Skills to Promote Students' Understanding of the Use of *static* in Object-Oriented Programming

Noa RAGONIS[1], Ronit SHMALLO[2]
[1]*Faculty of Education, Department of Computer Science, Beit Berl College*
Beit Berl, Kfar Saba, Israel
[2]*Department of Industrial Engineering and Management, Shamoon College of Engineering*
Ashdod, Israel
e-mail: noarag@beitberl.ac.il, ronits1@sce.ac.il

Received: December 2020

Abstract. Object-oriented programming distinguishes between instance attributes and methods and class attributes and methods, annotated by the *static* modifier. Novices encounter difficulty understanding the means and implications of *static* attributes and methods. The paper has two outcomes: (a) a detailed classification of aspects of understanding *static*, and (b) a collection of questions designed to serve as a learning/practice/diagnostic tool to address those aspects. Providing answers requires learners to apply higher-order cognitive skills and, hence, to advance their understanding of the essential meaning of the concept. Each question is analyzed according to three characteristics: (a) the *static* aspects that the question examines according to a detailed classification the paper provides; (b) identification of the question according: to Bloom's revised taxonomy, to the Structure of Observed Learning Outcome (SOLO) taxonomy; and to the problem-solving keywords used in the question's formulation. Several recommendations for teaching are presented.

Keywords: computer science education, object-oriented programming, static variable/method, SOLO taxonomy, Bloom's taxonomy, higher-order cognitive skills.

1. Introduction

Object-oriented (OO) languages have two main types of attributes and methods – those of instance (an object) and those of class – that are indicated by the *static* modifier. Instance attributes are related to a particular object, and an instance method is invoked on a particular object that was previously created, whereas *static* variables and methods are associated to the class and are generic and independent of any particular instance. Apparently, the *static* aspects are expressed syntactically by including the *static* modifier, but the distinction between the *static* and the non-*static* contexts is

based on the roles of those attributes or methods in the problem solution. The considerations while designing a solution to a problem – whether an attribute characterizes an object of the class or characterizes the class – are not syntactical, but they involve a deep understanding of object-oriented programming (OOP). Understanding the *static* notion involves further aspects such as memory allocation, access rules, program execution, and abstraction.

Students' obstacles in understanding different OOP concepts have been studied extensively (Eckerdal & Thuné, 2005; Kaczmarczyk *et al.*, 2010; Qian & Lehman, 2017; Sorva, 2007, 2008; Xinogalos, 2015), but only a few studies have been concerned about the *static* notion and the challenge to learn and understand it (Chen *et al.*, 2012; Shmallo *et al.*, 2012). None of these studies was devoted directly to the *static* notion or to its various interpretations and implementations. The current study aims at closing this gap by presenting a detailed classification of aspects of using *static* in various contexts, and by developing a collection of questions covering those aspects. The aim is that the collection of questions will serve as a learning/practice/diagnostic tool to be used by educators. Each of the developed questions covers various interpretations and contexts of *static* variables and methods, such as the following: refer to *static* variables in *static* and non-*static* contexts; invoke *static* methods from *static* and non-*static* contexts; examine the validity of references to *static* variables or methods; convert code between the two contexts; and examine the significance of the two contexts referring to aspects of software design.

The paper presents a classification of seven categories that consist of 24 precise aspects of using *static*; and offers a collection of questions to serve as a learning/practice/diagnostic tool. Each question is addressed by (a) the *static* aspects that the question examines according to the detailed classification; (b) an interpretation of the question in relation to Bloom's revised taxonomy of the cognitive processes of learners when solving problems and to the Structure of the Observed Learning Outcome (SOLO) taxonomy; and (c) the pedagogical aspects reflected in the question formulation. The questions were tested with 75 college students on two paths and were improved to account for students' obstacles in understanding the questions. The questions are presented in Java and can be easily transferred to other OOP programming languages, but the expected answers could be different, depending on the particular language rules for the use of *static*.

In what follows we present background on students' conceptions regarding *static* and on the educational learning approaches that led to the development of the questions; the classifications of aspects of understanding *static*; and a detailed presentation of six example questions addressing their goals at the conceptual level, the cognitive level, and the pedagogical level. We also present some implications for teaching, learning, and evaluating.

2. Background

In this section we present the basis for constructing the questions: the essential OOP misconceptions particularly in relation to the *static* notion, the role of *static* variables and

static methods as reflected in the code design, and the cognitive educational approaches that follow the principles under which they were developed.

2.1. *Misconceptions about OOP Concepts*

It is already well known that novices confuse the basic OO notions "class" and "object" and run into obstacles when trying to distinguish between their essence and usage (e.g., Holland *et al.*, 1997; Qian & Lehman, 2017; Shmallo *et al.*, 2012). In particular, they demonstrate difficulties in properly comprehending the notion of an instance (Eckerdal & Thuné, 2005; Xinogalos, 2015). Some view a class as being composed of a collection of sub-components (Thomasson *et al.*, 2006), sub-parts (Teif & Hazzan, 2006), or subsets (Xinogalos, 2015) of objects. Others perceive the terms *class* and *object* as equivalent or view a class as being composed of exactly one object (Eckerdal & Thuné, 2005; Garner *et al.*, 2005; Holland *et al.*, 1997).

Those misconceptions seem to stem from novices' difficulty in differentiating between the declarative phase of a program, where classes are defined, and the execution phase, where objects are created and used (e.g., Ragonis & Ben-Ari, 2005a). Some do not understand the need for the process of instance creation upon execution (Garner *et al.*, 2005; Ragonis & Ben-Ari, 2005b), and some do not understand the mechanism of instance creation and have difficulties realizing their memory allocation (e.g., Kaczmarczyk *et al.*, 2010; Ma *et al.*, 2007; Sorva, 2007). Additional difficulties arise when coping with how to access objects' attributes, believing, for example, that it is possible to access them directly from an outer class, although they were defined with private access (Shmallo *et al.*, 2012). Moreover, students demonstrate difficulties in comprehending the unique role of the "main" class and the main method versus classes that present an entity type (Ragonis & Ben-Ari, 2005b).

Previous research has dealt with the notion of *static* in the context of understanding OOP concepts, but it has not been directly addressed. Research regarding novices' misconceptions found that students confused the properties of *static* data members with constant data members (Chen *et al.*, 2012). Research that addressed novices tended to expand and reduce properties of terms such as *static* and *access* and reported that students believed that a *static* variable may have many occurrences rather than only one, thus expanding its number of occurrences (Shmallo *et al.*, 2012). In research that focused on students' conceptions and misconceptions regarding the *this* reference it was found that only 60% of the students indicated correctly that the *this* reference should not be used in any *static* context, and students showed difficulties when they needed to develop a *static* method replacing an instance method comprising the *this* reference (Ragonis & Shmallo, 2017, 2018; Shmallo & Ragonis, 2020). Conclusions in relation to students' understanding of the *static* notion are limited, since they arose during research on other concepts. As far as we know, no research has previously been dedicated to the *static* notion per se.

2.2. *Static Notion in OOP Languages*

Two main types of variables and methods are used in OO languages, resulting in important differences in program design (Lewis & Loftus, 2009; Olsson, 2020; Oracle, n.d.):

1) Instance variable vs. *static* variable – An instance variable is one per instance of a class (object), while a *static* variable is one per class. Each instance has its own instance variables, while all instances share one copy of a *static* variable.
2) Instance method vs. static method – An instance method characterizes the behavior of a class object, while a static method characterizes the class. An instance method is invoked only by an object identifier and refers to the instance variables, while a static method does not refer to any particular object (even though the static method can be accessed by the class name or by each of its instances).

A *static* variable has the advantage of persisting throughout the life of the program. Therefore, it can be used, for example, to count the number of times that a method has been called. Moreover, *static* methods can be included in (a) a class that defines an object type; (b) a special class designed to serve as a collection of *utils*; and (c) the "main" class, such as the *main* method, the most common example of a *static* method, which is the entry point of any program. This variety adds to students' confusion. Each programming language has its own rules about accessing a variable or method declared as *static* and about the access to variables or methods from a *static* method.

2.3. *Teaching Approaches That Led to Development of the Tool*

Coping with in-depth understanding of the *static* notion requires students to apply higher-order cognitive skills (HOCS). We developed and tested a collection of questions in the framework of HOCS, in relation to Bloom's revised taxonomy and the SOLO taxonomy and in relation to pedagogical problem-solving aspects reflected in the question formulation. Here we elaborate on those approaches.

2.3.1. *Higher-Order Cognitive Skills*

HOCS are skills that go beyond basic comprehension of a problem or concept. Students must acquire HOCS in order to make a decision under various conditions and to apply the knowledge acquired in class in novel and real-life situations (Bagarukayo *et al.*, 2012; Mbarika *et al.*, 2010; Zoller, 2003; Zoller *et al.*, 2002). HOCS promote learning and the development of capabilities such as asking questions, critical thinking, decision making, problem solving, and conceptualization of fundamental concepts (Leou *et al.*, 2006). Improving students' HOCS enhances the learners' abilities to identify, integrate, evaluate, and interrelate concepts within a given problem domain and thereby to make the appropriate decisions to solve a problem (Bagarukayo *et al.*, 2012).

2.3.2. *Bloom's Taxonomy of Educational Objectives*

Bloom's taxonomy (1956) is the most fundamental taxonomy, classifying learning objectives in education into six levels involved in planning students' learning and evaluating their achievements. The revised taxonomy presented by Anderson and Krathwohl (2001) kept the six classifications but updated the lowest and the highest levels and changed the wording to verbs that emphasize the doing. Further, the revision suggested that the upper three levels are not hierarchical but rather express similar cognitive abilities, identified as metacognitive knowledge. We adopted this approach and use the following verbs (in gerund form): level 1 is remembering; level 2 is understanding; level 3 is applying; level 4 is analyzing; level 5 is evaluating; and level 6 is creating. Like Anderson and Krathwohl (2001), we do not consider levels 4–6 hierarchical.

Attempts to apply Bloom's taxonomy in computer science (CS) education have been done earlier. For example, it was applied in CS course design, evaluation, and assessment, including introductory programming exams (Thompson *et al.*, 2008). It was also used in determining three difficulty levels for questions; for examining papers based on the criteria of keywords found in the questions; and for cross analysis of student performance, cognitive skill requirements, and module learning outcomes (Jones *et al.*, 2009).

2.3.3. *Structure of Observed Learning Outcome Taxonomy*

The SOLO taxonomy describes levels of increasing complexity in learners' understanding of subjects or performance tasks (Biggs & Collis, 2014). The taxonomy consists of five levels in the order of students' understanding of a learning task. Level 1 is prestructural (P): the student does not understand the task and refers to irrelevant aspects to solve it. Level 2 is unistructural (U): the student focuses on one relevant aspect to solve the task. Level 3 is multistructural (M): the student focuses on more than one relevant aspect to solve the task but does not integrate those aspects. Level 4 is relational (R): the student integrates several aspects of the task into a coherent structure in order to solve it. Level 5 is extended abstract (EA): the student incorporates the structure in a new one and adds more features, representing a higher and newer state of performance. The taxonomy may be used as an instructional as well as an evolutional tool. We adopted the SOLO taxonomy to classify the levels of increasing complexity in learners' understanding as should be reflected in their answers.

The SOLO taxonomy has been applied in the field of CS education to categorize students' cognitive abilities in various programming tasks, such as code comprehension, code writing and reading, and algorithm design (e.g., Ginat & Menashe, 2015; Izu *et al.*, 2016; Lister *et al.*, 2006; Qahmash *et al.*, 2017).

2.3.4. *Pedagogical Approach of Problem-Solving Questions in Computer Science*

Developing HOCS requires learning assignments that do not just settle on common types of questions, but rather refine concepts with questions that confront conflicts, and in relation to programming, in complex contexts that allow coverage of all aspects of a concept. In the development of our collection of questions and in their interpretation

as presented below, we covered multilayers of the *static* notion and used 11 out of 12 different types of question formulations suggested by Ragonis (2012). In addition, we classified keywords that appear in problem-solving questions into nine categories related to Bloom's cognitive levels 4–6 (Ragonis & Shilo, 2013). The categorization table appears in Appendix A.

3. Aspects of Understanding *static*

We categorized the various facets of the *static* role and usage into seven classifications, consisting of 24 precise aspects. The classification is based on the background presented in section *'Static* Notion in OOP Languages' and on the authors' accumulated experience teaching this complicated concept, which includes accumulating some initial findings from using the collection of questions in classes.

The list is grouped into classifications A–G:

A. Static *variable and* static *method declaration:*
   - A1. A s*tatic* variable is defined using the *static* modifier.
   - A2. A s*tatic* method is defined by the *static* modifier.
   - A3. A s*tatic* method uses parameters for any needed data.

B. Static *variable memory allocation:*
   - B1. A s*tatic* variable is allocated once.
   - B2. Each updating of a *static* variable relates to the same place in memory.
   - B3. A *static* variable has one value at a given state.

C. *Access to* static *variable:*
   - C1. A *static* variable can be accessed by referring to each of the class objects.
   - C2. A *static* variable can be accessed by referring to the class name.
   - C3. An instance method can access a *static* variable.
   - C4. A *static* method can access a *static* variable.

D. *Access to* static *method:*
   - D1. A *static* method can be accessed by an object of the same class.
   - D2. A *static* method can be accessed from outside the class by the class name.
   - D3. A *static* method can be accessed directly from the same class.
   - D4. A *static* method cannot be accessed by an object from another class.
   - D5. A *static* method can be accessed without creation of any object.

E. *Access by* static *method or from* static *context:*
   - E1. A *static* method cannot access an instance attribute.
   - E2. A *static* method cannot access an instance method directly.
   - E3. A class name cannot be used to access an instance method.

F. *Connotation and purpose of* static *variable:*
   - F1. A *static* variable can be used for communication between objects.
   - F2. A *static* variable can be used to count the number of class objects created.
   - F3. A *static* variable can be used to create a unique ID of an object.
   - F4. A *static* variable should be managed consistently according to the code design purpose.

G. *Connotation and purpose of* static *method:*

G1. A *static* method is generic and independent of any instance attributes.

The above classification scheme is the conceptual base that expresses the interpretation of SOLO taxonomy in the context of the current research and enables us to classify the collection of questions. The knowledge and thinking skills students require to answer the questions are reflected in our interpretation of SOLO levels 1–5:

- **Level 1:** Prestructural – Providing an answer requires the student to implement knowledge in relation to *static* in a superficial manner using basic rules, for example, a *static* variable can be accessed by each of the class objects.
- **Level 2:** Unistructural – Providing an answer requires the student to focus on only one relevant aspect of *static*, for example, writing an instruction to call a *static* method from an outer class.
- **Level 3:** Multistructural – Providing an answer requires the student to focus on more than one relevant aspect of *static*, but there is no need to integrate those aspects. An example is implementation of a *static* method that relates to an array of objects that have a *static* variable.
- **Level 4:** Relational – Providing an answer requires the student to integrate several aspects of *static*, for example, using discretion in developing a getter or setter method for a *static* variable.
- **Level 5:** Extended Abstract – Providing an answer requires the student to incorporate knowledge on *static* in a new interpretation representing a higher and newer state of performance. An example is understanding the consequence of converting a *static* method to an instance method, by presenting discretions and implementing the needed changes, or making up a new goal for a *static* variable.

4. Collection of Questions

The collection consists of 22 questions, each focusing on different aspect(s) of the implication and implementation of the *static* notion.

Questions Context. The questions introduced to students relate to a project including three classes:

- Class *Box*: represents a three-dimensional box, with a *static* variable *num*, and a *static* method *whichNum()*
- Class *BoxUtils*: is a collection of *utils* methods that implement three *static* methods
- Class *Test*: has a *main()* method aimed at creating objects and testing the variety of instance methods and *static* methods.

Each question addresses part of the project. The collection of questions appears in Appendix B.

Presentation Structure. The structure of each question consists of the question's target, formulation, and expected answer. Each question is analyzed according to three characteristics:

(a) **The *static* aspects** that the question examines, according to the categories presented in the section "Aspects of Understanding *static*".

(b) **Assignment of the question to level(s) of the two taxonomies:** the revised Bloom taxonomy (Anderson & Krathwohl, 2001), which identifies the learning objective that the student will achieve when solving the question correctly; and the SOLO taxonomy (Biggs & Collis, 2014), which identifies the cognitive abilities learners acquire as learning outcomes for solving future problems. It is important to note that we relate to Bloom's taxonomy as nonhierarchical, and a question can address some learning objectives from different categories. When a question is associated with more than one level, it is displayed in descending order of the level number, from the highest to the lowest.

(c) **Pedagogical aspects that examine cognitive ability** to cope with questions that call for a significant understanding of the concepts. This includes **the question type** (Ragonis, 2012); and **the category of problem-solving keywords** used in the question (Ragonis & Shilo, 2013).

In what follows we present six examples to illustrate the analysis of questions. The full analysis can be found in the complementary materials.

4.1. *Question 2*

Question 2 displays three alternative instructions. Each accesses the *static* method *whichNum()* in class *Box*, which returns the value of the *static* variable *num*.

The question

> Replace the statement in Line 1 at the *main()* method with each of the following statements, and state for each whether it is correct or not. If it is correct, display the output; if not, explain why.
>
> I. System.out.println ("Print:" + boxArr[1].whichNum ());
> II. System.out.println ("Print:" + topBox.whichNum ());
> III. System.out.println ("Print:" + Box.whichNum ());

Expected answer

Instructions I and II are correct, since the *static* method *whichNum()* can be accessed by an instance of the same class. Instruction III is correct, since the reference is by the class name. The output for all instructions will be "Print:4".

The question interpretation

Static aspects	• The *static* method *whichNum()* can be accessed by each of the class objects – *boxArr[1], topBox* (D1) or by the class name *Box* (D2) • The *static* method *whichNum()* can access the *static* variable *num* (C4)
Bloom's taxonomy	Level 3 – Applying

SOLO taxonomy	Level 3 – Multistructural
Type of question	Type 3 – Tracing a given solution Type 4 – Analysis of code execution Type 6 – Examination of the correctness of a given solution
Problem-solving keywords category	Category 2 – Argue and justify Category 3 – Analyze

4.2. *Question 4*

Question 4 is aimed at examining students' conceptions in relation to a *static* variable. It asks what the *static* notion potential and relevant purpose is, beyond the syntactic rules of its usage. The students are expected to offer a context or purpose for including the *static* variable *num* in the class *Box*.

The question

> What do you think the *num* variable stands for and how/why do the *static* properties support it?

Expected answer

A direct and limited answer could be that it is a shared variable of all class objects. A more useful implication is that *num* stands for the number of boxes already instantiated. Students exposed to other applications could suggest them (such as the idea presented in question 21).

The question interpretation

Static aspects	• *num* can be used for communication between the class *Box* objects (F1), or to count the number of boxes already created (F2), or to give a unique identification to each box at the time of creation (F3)
Bloom's taxonomy	Level 5 – Evaluating
SOLO taxonomy	Level 4 – Relational
Type of question	Type 5 – Finding the purpose of a given solution Type 11 – Programming style/aspects questions
Problem-solving keywords category	Category 1 – Address / define criteria Category 7 – Discover

4.3. *Question 8*

In question 8 students must focus on the semantics and syntactical aspects of method declaration as an instance method versus a *static* method.

The question

> The programmer seeks to move the *volume()* method defined in class *BoxUtils* and to define it at class *Box*; should any change be made? If not, explain your position; if yes, write down the method with the required changes and explain the changes.

Expected answer

The changes that should be made are removal of the *static* modifier and removal of the parameters that represent the box dimensions, since they are the box attributes. Access to the attributes can be direct now, but students can keep the getters methods.

The question interpretation

Static aspects	• Since a *static* method is annotated by the *static* modifier, it should be removed and replaced by an instance method (A2) • The *volume()* method has to use parameters, one for each dimension. But, as an instance method, since the box dimensions are part of the object, they should not appear as parameters (A3)
Bloom's taxonomy	Level 5 – Evaluating Level 4 – Analyzing Level 3 – Applying
SOLO taxonomy	Level 4 – Relational
Type of question	Type 11 – Programming style/aspects questions Type 12 – Transformation of a solution
Problem-solving keywords category	Category 2 – Argue and justify Category 3 – Analyze Category 8 – Develop

4.4. *Question 9*

Question 9 is aimed at examining students' OOP design conceptions and asks them to take a stand on whether the method *volume()* should be defined in class *BoxUtils*, as in the presented project, or whether it is better to define it as an instance method.

The question

> Where do you think it is more appropriate to define the method *volume()*, in class *BoxUtils* or in class *Box*? Explain your answer.

Expected answer

Software design considerations should lead the students to define the method *volume()* as an instance method, since it relates to any box, and the box dimensions are box attributes.

The question interpretation

Static aspects	• The method should be declared *static* if it performs a generic function. The *volume()* method is indeed a generic calculation, but the context is boxes. Since there is a particular class that represents a box – class *Box* – it should be defined there (G1)
Bloom's taxonomy	Level 5 – Evaluating Level 4 – Analyzing
SOLO taxonomy	Level 5 – Extended Abstract
Type of question	Type 11 – Programming style/aspects questions

Problem-solving keywords category	Category 1 – Address / define criteria Category 2 – Argue and justify

4.5. *Question 14*

Question 14 aims to follow the getter method for a *static* variable from a *static* method defined in another class. In this case, the only option is to use the class name, since none of the class objects exists in that location.

The question

> Complete the instruction "int n = …" in the method *build()* defined in class *BoxUtils* so *n* will be assigned the value of *num* from class *Box*.

Expected answer

Since there is not any object of class *Box* in the method *build()*, the only option is to use the method *whichNum()* and to access it by the class name *Box*: *Box.whichNum()*.

The question interpretation

Static aspects	• The *static* variable *num* can be assessed by the *static* method *whichNum()* (C4) • A *static* method can be accessed from outside the class by using the class name; hence, using *Box* is needed (D2) • This is the only option, since there is no object of class *Box* that can be used for accessing the method (in contradiction to D1)
Bloom's taxonomy	Level 3 – Applying
SOLO taxonomy	Level 3 – Multistructural
Type of question	Type 7 – Completion of a given solution
Problem-solving keywords category	Category 5 – Complete

4.6. *Question 21*

In question 21 students are expected to complete a software design assignment, in relation to a different usage of a *static* variable: giving each of the class created objects a unique identification.

The question

> The class *Box* should fulfill a new request. Each object has to have an attribute that expresses its unique identification: a serial number, which will start with the string "SN_" following its unique number. For example, the serial number for the first box will be "SN_1", the second will be "SN_2", etc. Make all the needed changes in the class *Box*.

Expected answer

Three main changes have to be made: (a) add a new instance attribute, a serial number, to the class *Box*; (b) build a mechanism that will allow the creation of the needed *String*;

and (c) add an instruction to initial the new attribute in the *Box* constructor. One option for the new code is the following:

```
public class Box
private int width;
private int length;
private int depth;
private String serialNumber = “”;
private static int num = 0;
public Box(int width, int length, int depth)  {
        this.width = width;
        this.length = length;
        this.defth = depth;
        num++;
        this.serialNumber = “SN_” + num;
}
public String getSerialNumber() {
        return serialNumber;
}
```

The question interpretation

Static aspects	• An additional instance attribute should be defined to fulfill the new request. It should not be *static* (in contradiction to F1) • The previously defined *static* variable *num* can serve in the creation of the unique identification for each of the created objects (F1, F3) • Adding the *getSerialNumber()* method, and not the *setSerialNumber()* method, is in line with managing the *serialNumber* variable (F4)
Bloom’s taxonomy	Level 6 – Creating Level 5 – Evaluating Level 4 – Analyzing Level 3 – Applying
SOLO taxonomy	Level 5 – Extended abstract
Type of question	Type 11 – Programming style/aspects questions Type 1 – Development of a solution
Problem-solving keywords category	Category 8 – Develop Category 9 – Integrate

The full collection of 22 questions covers the entire classification consisting of the seven categories A–G and the 24 precise aspects addressed, while each question sharpens a different context. The full classifications are presented in Appendix C. Browsing the full collection is recommended to enable educators and researchers to choose questions according to their purpose.

5. Summary

The *static* notion is one of the OOP foundations. Understanding the concept of *static* requires knowledge of aspects of OOP design, memory allocation, access rules, program execution, and abstraction. The role and usages of *static* variables and *static* methods are not restricted to syntactical distinctions; rather, they involve very basic OOP conceptions of object and class. Previous research has not directly dealt with the *static* notion, but students' obstacles have arisen during research on other OOP conceptions.

The study has two outcomes. The first is a detailed classification of seven categories that consist of 24 precise aspects in relation to the use of *static*. The second is a collection of 22 questions covering all those multifaceted aspects that can serve as a learning/practice/diagnostic tool to address students' difficulties in acquiring the *static* notion, which implies students' HOCS. The classification of the seven categories with 24 precise aspects is based on the literature background, on the authors' accumulated experience teaching the *static* notion, and on accumulating findings from using the collection of questions in classes. The categorization of each question into the (a)–(c) characteristics – static aspects, SOLO and Bloom's taxonomies, and type of question – was firstly done individually by each of the researchers; later, the researchers discussed their categorizations until reaching full agreement.

Developing HOCS requires learning assignments that do not settle on common types of questions; rather, assignments should refine the concepts with questions that confront conflicts. In particular, the programming tasks should offer complex contexts that allow coverage of all aspects of a concept. The collection of questions meets these objectives and covers 11 of the 12 types of question suggested by Ragonis (2012); it also uses various problem-solving keywords of the nine categorized, as suggested by Ragonis & Shilo (2013), which are themselves attributed to Bloom's revised cognitive levels 4–6. We believe that because students cannot be expected to understand intuitively the concept of the current object *this* (Ragonis & Shmallo, 2017, 2018; Shmallo & Ragonis, 2020), the understanding of the *static* notion cannot be accurate without facilitating it using tools that enable meaningful learning.

Analysis of the question collection, as presented in Appendix C, reveals the following:

- In light of the revised Bloom's taxonomy (Anderson & Krathwohl, 2001), the questions reflect the four advanced cognitive skills: 3, applying (expressed in 12 questions); 4, analyzing (expressed in 8 questions); 5, evaluating (expressed in 10 questions); and 6, creating (expressed in 3 questions). However, there is no expression of the two lower levels: 1, remembering, and 2, understanding.
- In our incorporation of SOLO taxonomy (Biggs & Collis, 2014), the three advanced levels are conveyed widely: 3, multistructural (expressed in 7 questions); 4, relational (expressed in 8 questions); and 5, extended abstract (expressed in 6 questions). Level 1, prestructural, does not appear at all; and level 2, Unistructural, is expressed only in one question.

The collection of questions aims to directly expose students' perceptions and misconceptions regarding the concept *static* in different contexts. It is written in Java and can be translated into other OOP languages, but the answers are subject to the rules of the particular programming language used. The questions highlight wide and varied aspects of the use of *static*, frequent usages, and non-frequent usages. Open-ended questions are also used to allow students to express their understanding, including from the perspective of OOP design. We do not recommend using all the questions if the allocated time is limited. The questions are comprehensive and require time for students to cope with them in order to respond properly.

The *static* topic has not been studied in depth in the past; hence, we believe that the current study makes a valuable contribution. The classification of aspects of understanding the *static* notion and the collection of questions can be used as tools for learning, teaching, and research.

References

Anderson, L.W., Krathwohl, D.R. (Eds.) (2001). *A Taxonomy for Learning, Teaching, and Assessing: A Revision of Bloom's Taxonomy of Educational Objectives*. Longman.

Bagarukayo, E., Weide, T., Mbarika, V., Kim, M. (2012). The impact of learning driven constructs on the perceived higher order cognitive skills improvement: Multimedia vs. text. *International Journal of Education and Development using ICT*, 8(2), 120–130.

Biggs, J.B., Collis, K.F. (2014). *Evaluating the Quality of Learning: The SOLO Taxonomy (Structure of the Observed Learning Outcome)*. Academic Press.

Bloom, B.S. (1956). *Taxonomy of Educational Objectives. Handbook 1: Cognitive Domain*. McKay.

Chen, C.-L., Cheng, S.-Y., Lin, J. M.-C. (2012). A study of misconceptions and missing conceptions of novice Java programmers. In: *Proceedings of the International Conference on Frontiers in Education: Computer Science and Computer Engineering (FECS'12)*. Steering Committee of the World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), pp. 84–89.

Eckerdal, A., Thuné, M. (2005). Novice Java programmers' conceptions of "object" and "class", and variation theory. *ACM SIGCSE Bulletin*, 37(3), 89–93.

Garner, S., Haden, P., Robins, A. (2005, January). My program is correct but it doesn't run: A preliminary investigation of novice programmers' problems. In: *Proceedings of the 7th Australasian Conference on Computing Education,* Vol. 42 (pp. 173–180).

Ginat, D., Menashe, E. (2015, February). SOLO taxonomy for assessing novices' algorithmic design. In: *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (pp. 452–457).

Holland, S., Griffiths, R., Woodman, M. (1997, March). Avoiding object misconceptions. In: *Proceedings of the Twenty-eighth SIGCSE Technical Symposium on Computer Science Education* (pp. 131–134).

Izu, C., Weerasinghe, A., Pope, C. (2016, August). A study of code design skills in novice programmers using the SOLO taxonomy. In: *Proceedings of the 2016 ACM Conference on International Computing Education Research* (pp. 251–259).

Jones, K. O., Harland, J., Reid, J. M., Bartlett, R. (2009, October). Relationship between examination questions and Bloom's taxonomy. In: *2009 39th IEEE Frontiers in Education Conference*. IEEE, pp. 1–6.

Kaczmarczyk, L.C., Petrick, E.R., East, J.P., Herman, G.L. (2010). Identifying student misconceptions of programming. In: *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*. ACM, pp. 107–111.

Leou, M., Abder, P., Riordan, M., Zoller, U. (2006). Using 'HOCS-centered learning' as a pathway to promote science teachers' metacognitive development. *Research in Science Education*, 36(1–2), 69–84.

Lewis, J., Loftus, W. (2009). *Java software solutions: Foundations of program design* (8th ed.). Pearson/ Addison-Wesley.

Lister, R., Simon, B., Thompson, E., Whalley, J.L., Prasad, C. (2006). Not seeing the forest for the trees: Novice programmers and the SOLO taxonomy. *ACM SIGCSE Bulletin*, 38(3), 118–122.

Ma, L., Ferguson, J., Roper, M., Wood, M. (2007, March). Investigating the viability of mental models held by novice programmers. In: *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education* (pp. 499–503).

Mbarika, V., Bagarukayo, E., Hingorani, V., Stokes, S., Kourouma, M., Sankar, C. (2010). A multi-experimental study on the use of multimedia instructional materials to teach technical subjects. *Journal of STEM Education: Innovations and Research*, 11(2), 24–37.

Olsson, M. (2020). Static. In: *C# 8 Quick Syntax Reference*. Apress, pp. 85–90.

Oracle (n.d.). Oracle Java documentation – The Java tutorials: Using the *this* keyword. `https://docs.oracle.com/javase/tutorial/java/javaOO/thiskey.html`

Qahmash, A., Joy, M., Boddison, A., Needs, S.E. (2017). Investigating high-achieving students' code-writing abilities through the SOLO taxonomy. In: *Proceedings of the 28th Annual Conference of the Psychology of Programming Interest Group (PPIG 2017)*. Psychology of Programming Interest Group, pp. 17–27.

Qian, Y., Lehman, J. (2017). Students' misconceptions and other difficulties in introductory programming: A literature review. *ACM Transactions on Computing Education (TOCE)*, 18(1), 1–24.

Ragonis, N., Ben-Ari, M. (2005a). A long-term investigation of the comprehension of OOP concepts by novices. *Computer Science Education*, 15(3), 203–221.

Ragonis, N., Ben-Ari, M. (2005b). On understanding the static's and dynamics of object-oriented programs. *ACM SIGCSE Bulletin*, 37(1), 226–230.

Ragonis, N. (2012). Type of questions – The case of computer science. *Olympiads in Informatics*, 6, 115–132.

Ragonis, N., Shilo, G. (2013). What is it we are asking: Interpreting problem-solving questions in computer science and linguistics. In: *Proceeding of the 44th ACM technical symposium on Computer science education* (SIGCSE'13). ACM, New York, NY, USA, pp. 189–194.

Ragonis, N., Shmallo, R. (2017). On the (Mis) understanding of the "this" reference. In: *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education* (SIGCSE'17). ACM, New York, NY, USA, pp. 489–494.

Ragonis, N., Shmallo, R. (2018). A diagnostic tool for assessing students' perceptions and misconceptions regards the current object "this". In: S. Pozdniakov & V. Dagienė (eds), Informatics in Schools – Fundamentals of Computer Science and Software Engineering. ISSEP 2018. *Lecture Notes in Computer Science*, *vol 11169*. Springer, Cham, pp. 84–100.

Shmallo, R., Ragonis, N., Ginat, D. (2012). Fuzzy OOP: Expanded and reduced term inter-pretation. In: *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education* (ITiCSE'12), 309–314.

Shmallo, R., Ragonis, N. (2021). Understanding the "this" reference in object oriented programming: Misconceptions, conceptions, and teaching recommendations. *Education and Information Technologies*, 26(1), 733–762.

Sorva, J. (2007). Students' understandings of storing objects. In *Proceedings of the Seventh Baltic Sea Conference on Computing Education Research* (vol. 88, pp. 127–135). Australian Computer Society, Inc.

Sorva, J. (2008). The same but different students' understandings of primitive and object variables. In *Proceedings of the 8th International Conference on Computing Education Research*. ACM, pp. 5–15.

Teif, M., Hazzan, O. (2006). Partonomy and taxonomy in object-oriented thinking: junior high school students' perceptions of object-oriented basic concepts. In: *Working group reports on ITiCSE on Innovation and technology in computer science education.* pp. 55–60.

Thomasson, B., Ratcliffe, M., Thomas, L. (2006). Identifying novice difficulties in object oriented design. *ACM SIGCSE Bulletin*, 38(3), 28–32.

Thompson, E., Luxton-Reilly, A., Whalley, J.L., Hu, M., Robbins, P. (2008, January). Bloom's taxonomy for CS assessment. In: *Proceedings of the Tenth Conference on Australasian Computing Education*. (vol. 78, pp. 155–161).

Xinogalos, S. (2015). Object-oriented design and programming: An investigation of novices' conceptions on objects and classes. *ACM Transactions on Computing Education (TOCE)*, 15(3), Article 13.

Zoller, U. (2003). HOCS problem solving vs. LOCS exercise solving: What do college science students prefer? In *Science Education Research in the Knowledge-Based Society*. Springer, pp. 201-207.

Zoller, U., Dori, Y., Lubezky, A. (2002). Algorithmic, LOCS and HOCS (chemistry) exam questions: Performance and attitudes of college students. *International Journal of Science Education*, 24(2), 185–203.

N. Ragonis is a senior lecturer. She is the head of the M.Ed. in Integrative STEM Education program and the head of the Computer Science Department, at Beit Berl College. She is also a senior lecturer at the Faculty of Education in Technology and Science, Technion. Ragonis is a graduate of the Faculty of Mathematics and Computer Science at Bar-Ilan University. She received her MSc and PhD degrees at the Weizmann Institute of Science and was a postdoctoral fellow at the Faculty of Education in Technology and Science, Technion. Her research mainly focuses on cognitive aspects of teaching and learning of Computer Science and Computational Thinking, in particular in relation to Logic Programming and Object-Oriented Programming, as well as integrating ICT and innovation in teaching and learning processes. She has published over 60 articles in journals, conferences and chapters in books. She has co-authored the book Guide to Teaching Computer Science (2011; 2014; 2020, Springer) and has authored ten Computer Science high-school textbooks and teachers book guidelines, in relation to OOP, Logic Programming, and Computational Models. Engaged in pre-service and in-service teachers' preparation programs.

R. Shmallo is a senior lecturer in the Department of Industrial Engineering and Management and a member of the Center of the Advancement of Teaching at SCE (Shamoon College of Engineering). She received her PhD in Computer Science Education from Tel-Aviv University in 2013. Her teaching is primarily in computer science programming and analysis and design of information systems using the object-oriented approach. Her research focuses on difficulties encountered by novices in trying to understand the cornerstones of object-oriented programming. Her study involved an examination of a new teaching method that integrates explicit orientation to errors in a way that enables students to strive to learn from those errors in computer science, databases, and other areas.

Appendix A: Problem-Solving Question Categories and Keywords

Table A1

Problem-Solving Question Categories and Keywords (Ragonis & Shilo, 2013)

Category	Keywords	Interpretation
1. Address / define criteria	address / apply / note / mention / specify / indicate / sort / mark	Address and apply different kinds of criteria and attribute the criteria to a list of elements; define criteria
2. Argue and justify	argue / state / assert / determine, followed by justification: explain / argue / prove / justify / demonstrate / illustrate / clarify	State opinion and further establish the claim using any kind of justification
3. Analyze	analyze / examine / investigate / explore	Identify and analyze the meaning or significance of components and factors
4. Compare	compare / classify	Compare different objects/issues by applying principles and observing from different viewpoints; generalize insights
5. Complete	complete / add	Complete or add components to a given structure according to detailed requirements
6. Convert	convert / represent in different forms / modify / adjust / change / transform	Convert a given paragraph/section/clause according to specified, meaningful, qualitative-related instructions (not technical translation)
7. Discover	discover / identify / find out / say what	Discover a phenomenon / indicate an occurrence / find out the purpose / identify components and the relations between them
8. Develop	develop / compose / write / create new elements	Develop a new component / write a new module
9. Integrate	integrate / order / arrange / merge / combine	Integrate some given components into a new structure

Appendix B: The Collection of Questions in Java

The following is a project that includes:

- The class *Box* with a *static* variable *num*, and a *static* method *whichNum()*.
- The class *BoxUtils*, which implements three *static* methods.
- The class *Test* with a main method.

```
public class Box
private int width;
private int length;
private int height;
private static int num = 0;
```

```
public Box(int width, int length, int height)  {
        this.width = width;
        this.length = length;
        this.height = height;
        num++;
}
public int getWidth()
public int getLength()
public int getHieght()
public int getNum()
public void setWidth(int width)
public void setLength(int length)
public void setHeight(int height)
public String toString()
public static int whichNum()  {
        return num;
}
```

public class BoxUtils

```
public static int volume(Box c) {
        return c.getWidth() * c.getLength()* c.getHeight();
}
```

```
public static Box Build(Box c1, Box c2) {
        return new Box(c1.getWidth(), c1.getLength(), c1.getHeight()+c2.getHeight());
}
```

```
public static Box build() {
        int n =  _____;
        return new Box(n, n*2, n*3);
}
```

public class **Test**

```
public static void main(String[] args)   {
        (*)   ……..
                Box[] boxArr = new Box[10];
                for (int n=1; n<4; n++)  {
                        boxArr[n] = new Box(n, n, n);
                }
                Box topBox = new Box(1, 2, 3);
        1)  System.out.println (“Print: “ +  _____);
        2)  _____ ;
        3)  boxArr[4] = _____ ;
        4)  System.out.println (“Print: “ +  _____);
        5)  _____ ;
}
```

Answer the following questions:

1. Replace the statement in Line 1 at the *main()* method with each of the following statements, and state for each whether it is correct or not. If it is correct, display the output; if not, explain why.
   I. System.out.println ("Print: " + boxArr[1].getNum());
   II. System.out.println ("Print: " + boxArr[2].getNum());
   III. System.out.println ("Print: " + topBox.getNum());
   IV. System.out.println ("Print: " + Box.getNum());
2. Replace the statement in Line 1 at the *main()* method with each of the following statements, and state for each whether it is correct or not. If it is correct, display the output; if not, explain why.
   I. System.out.println ("Print:" + boxArr[1].whichNum ());
   II. System.out.println ("Print:" + topBox.whichNum ());
   III. System.out.println ("Print:" + Box.whichNum ());
3. In the *main()* method at the place marked by (*) the programmer added the next instruction: System.out.println ("Print: " + Box.whichNum());
   a. State whether it is correct or not. If it is correct, display the output; if not, explain why.
   b. Instead of *Box*, the programmer wrote *boxArr[0]*; state whether it is correct or not. If it is correct, display the output; if not, explain why.
4. What do you think the *num* variable stands for and how/why do the *static* properties support it?
5. Do you think it is necessary to include the conventional *setNum(int num)* method in the class *Box*? Explain your answer.
6. Do you think that each *static* variable should be initialized by 0 (zero)? Please explain your answer. If you think differently, suggest a scenario where a different initialization of a *static* variable is relevant.
7. Explore the method *volume()* defined in class *BoxUtils*. Replace the statement in line 2 at the *main()* method with each of the following statements, and state for each whether it is correct or not. If it is correct, display the output; if not, explain why.
   I. System.out.println ("The volume is:" + topBox.volume (topBox));
   II. System.out.println ("The volume is:" + volume (topBox));
   III. System.out.println ("The volume is:" + BoxUtils.volume (topBox));
8. The programmer seeks to relocate the *volume()* method defined in class *BoxUtils* and to define it at class *Box*; should any change be made? If not, explain your position; if yes, write down the method with the required changes and explain the changes.
9. Where do you think it is more appropriate to define the method *volume()*, in class *BoxUtils* or in class *Box*? Explain your answer.
10. Complete the instruction in line 3 at the *main()* method to build a new box using the *build* method that has two parameters. Choose any two boxes previously created.

11. Complete the instruction in line 4 in the *main()* method in order to print the value of *num*.
12. What will be the output of the instruction you completed in line 4 (question 11)?
13. If the access modifier of the class *Box num* variable is changed from *private* to *public*, can the instruction in line 4 at the *main()* method be written differently from what you suggested in question 11? Explain your determination, and if your answer is positive please write the new instruction.
14. Complete the instruction "int n = …" in the method *build()* defined in class *BoxUtils* so *n* will be assigned the value of *num* from class *Box*.
15. In class *BoxUtils*, two methods named *build* are defined, one with two parameters and the other with none. Those two methods return a new object of class *Box*. Does their declaration follow your answer to question 5 in relation to what the *static* variable *num* stands for? Do they keep the objective of *num* objective?
16. Develop a method in class *Box* to achieve the same task as the method *build()* with no parameters in class *BoxUtils*.
17. Do you think that it is more appropriate to define the two build methods defined in the class *BoxUtils* in the class *Box*? elaborate your considerations.
18. The programmer adds the next *sumHights()* method to the class *Test*. The method aims to return the sum of heights of the boxes in the array of boxes. See the method code and add relevant parameter/parameters, if needed.

```
public static int sumHeights(_____ - ? - _____) {
        int sum = 0;
        for (int i=1; i<boxes.length && boxes[i] != null; i++)  {
              sum+= boxes[i].getHeight();
        }
        return sum;
}
```

19. In line 5 at the *main()* method, write an instruction to call the method *sumHeights()*, in relation to the array of boxes existing in the *main()* method.
20. The programmer wishes to relocate the *sumHeights()* method from the class *Test* to the class *BoxUtils*. Which changes should be made in the method code or in the method call?
21. The class *Box* should fulfill a new request. Each object has to have an attribute that expresses its unique identification: a serial number, which will start with the string "SN_" following its unique number. For example, the serial number for the first box will be "SN_1", the second will be "SN_2", etc. Make all the needed changes in the class *Box*.
22. What additional objective of a *static* variable can fit into the implementation of this project? Write down what the purpose of this variable is, for what it can be used, and where it will be defined.

Appendix C: Mapping All Question Classifications

Table C1

Mapping All Question According to the five Classifications

Question No.	*Static* aspects							Bloom's taxonomy						SOLO taxonomy					Type of question	Category of PSQ*
	A	B	C	D	E	F	G	1	2	3	4	5	6	P	U	M	R	EA		
1		B1 B2 B3	C3	D1	E3					X						X			3 4 6	2 3
2			C4	D2						X						X			3 4 6	2 3
3			C4	D1 D2 D5						X							X		3 4 6	2 3
4						F1 F2 F3						X					X		5 11	1 7
5						F4						X					X		11	1 2
6						F4						X						X	6 11	1 2
7				D1 D2 D4							X						X		3 4 6	2 3
8	A2 A3									X	X	X					X		11 12	2 3 8
9							G1				X	X						X	11	1 2
10				D2						X						X			7	5
11			C1 C2 C3 C4							X						X			7	5
12		B1 B2 B3									X					X			3 4	3
13						F4						X					X		3 4 8 11	2 4 8
14			C4	D1 D2						X						X			7	5

Continued on next page

Table C1 – continued from previous page

Question No.	*Static* aspects							Bloom's taxonomy						SOLO taxonomy					Type of question	Category of PSQ*
	A	B	C	D	E	F	G	1	2	3	4	5	6	P	U	M	R	EA		
15						F2 F4						X					X		11	1 2 3
16	A2		C3							X	X	X	X					X	2	6
17							G1					X						X	11	1 2
18	A3				E1					X	X						X		7	5
19				D3						X					X				7	5
20	A2 A3			D2						X	X					X			11 12	3 6 8
21						F1 F3 F4				X	X	X	X					X		1 11	8 9
22						F1 F2 F3 F4							X					X	10	1 2 8

* Category of Problem-Solving Keywords