Identifying and Validating Java Misconceptions – Complementary Material.

Ricardo Caceffo *Pablo Frank-Bolton* *Renan Souza*
Rodolfo Azevedo

Technical Report - IC-19-05 - Relatório Técnico
April - 2019 - Abril

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

Identifying and Validating Java Misconceptions – Complementary Material.

Ricardo Caceffo[1], Pablo Frank-Bolton[2], Renan Souza[3], Rodolfo Azevedo[4]
[1] Institute of Computing, State University of Campinas (Unicamp) – caceffo@ic.unicamp.br
[2] The George Washington University – pfrank@gwmail.gwu.edu
[3] Institute of Computing State University of Campinas (Unicamp) – 147783@students.ic.unicamp.br
[4] Institute of Computing State University of Campinas (Unicamp) – rodolfo@ic.unicamp.br

Abstract: This Technical Report presents complementary material related to the article "Identifying and Validating Java Misconceptions Toward a CS1 Concept Inventory", to be published in the *Proceedings of the 24th Annual ACM Conference on Innovation and Technology in Computer Science Education* (ITiCSE 2019).

Article abstract: A misconception is a common misunderstanding that students may have about a specific topic. The identification, documentation, and validation of misconceptions is a long and time-consuming work, usually carried out using iterative cycles of students answering open-ended questionnaires, interviews with instructors and students, exam analysis, and discussion with experts. A comprehensive list of validated misconceptions in some subject can be used to build formal evaluation methods like the Concept Inventory (CI), a multiple-choice questionnaire that is usually performed as pre-post tests in order to assess any change in student understanding. In CS1, validated misconceptions were identified and documented in C and Python programming languages. Although there are studies related to misconceptions in the Java language, these misconceptions lack the formality, comprehensiveness, and robustness of their C and Python counterparts. On this work, we propose a methodology to adapt the validated misconceptions in C and Python to Java. Initially, through the analysis of an initial list of 33 misconceptions in C and 28 in Python, we identified and documented in an antipattern format 31 possible misconceptions in Java. We then developed a final term exam, composed of 7 open-ended questions, in which each question was designed to address some of the misconceptions covered in the course (N = 27). Through the analysis of the exam's answers (N = 69 students), it was possible to validate 22 of the misconceptions (81%). Also, 6 new misconceptions were identified, leading to a total of 28 valid misconceptions in Java.

Keywords: CS1, Course, Misconception, Concept, Inventory, Java, Student, Assessment, Introductory, Programming, Multiple, Choice, Question

1. Introduction

Introductory Programming Courses (CS1) are taught with a variety of programming languages, like C, Python and Java. These courses, usually common to all STEAM students in a University, are critical to the development of logical and algorithm thinking, being increasingly associated to the new needs and requirements of the labor market in the most diverse areas.

A Concept Inventory (CI) is a set of multiple-choice questions that can be used to assess the students' comprehension on some topic at some point during a course [2, 3]. Each incorrect choice corresponds to a specific misconception – an inaccurate line of thought students often follows.

This manuscript is part of an ongoing work that aims to create and validate CIs for CS1 courses. As proposed by Almstrum et al. [5], the first step to create such a CI is the identification of the student's misconceptions. **In previous work** we identified misconceptions for Introductory Programming Courses (CS1) in C [2, 4, 9] and Python [6] programming languages. We also created a CI for the C programming language [10], available in this link: http://edu.ic.unicamp.br/limesurvey/

Specifically**, this Technical Report presents complementary** material related to the article "Identifying and Validating Java Misconceptions Toward a CS1 Concept Inventory", to be published in the *Proceedings of the 24th Annual ACM Conference on Innovation and Technology in Computer Science Education* (ITiCSE 2019)[1]. The article reference [8] is:

> CACEFFO, R.; FRANK-BOLTON, P.; R, SOUZA.; AZEVEDO, R. (2019). Identifying and Validating Java Misconceptions Toward a CS1 Concept Inventory. *In Proceedings of the 24th Annual ACM Conference on Innovation and Technology in Computer Science Education, ITCSE'19, July 15-17, 2019, Aberdeen, Scotland,* ISBN: 978-1-4503-6301-3/19/07 DOI: 10.1145/3304221.3319771 https://doi.org/10.1145/3304221.3319771

Figure 1 shows the first page of the ITiCSE article. This Technical Report is referenced in the article as reference "[6]".

[1] **Note:** In the moment of this Technical Report publication (April 2019), the ACM ITiCSE 2019 proceedings is not yet available to the general public. The conference will be held in July 2019 at the Aberdeen University (Aberdeen, Scotland, UK). Until then, the authors are glad to make the article available to anyone interested, upon email request or through the following website: http://www.ricardocaceffo.com/publications

Identifying and Validating Java Misconceptions Toward a CS1 Concept Inventory

Ricardo Caceffo
State University of Campinas
Campinas, SP, Brasil
caceffo@ic.unicamp.br

Pablo Frank-Bolton
The George Washington University
Washington, D.C, USA
pfrank@email.gwu.edu

Renan Souza
State University of Campinas
Campinas, SP, Brasil
renan-plsouza@uol.com.br

Rodolfo Azevedo
State University of Campinas
Campinas, SP, Brasil
rodolfo@ic.unicamp.br

ABSTRACT

A misconception is a common misunderstanding that students may have about a specific topic. The identification, documentation, and validation of misconceptions is a long and time-consuming work, usually carried out using iterative cycles of students answering open-ended questionnaires, interviews with instructors and students, exam analysis, and discussion with experts. A comprehensive list of validated misconceptions in some subject can be used to build formal evaluation methods like the Concept Inventory (CI), a multiple-choice questionnaire that is usually performed as pre-post tests in order to assess any change in student understanding. In CS1, validated misconceptions were identified and documented in C and Python programming languages. Although there are studies related to misconceptions in the Java language, these misconceptions lack the formality, comprehensiveness, and robustness of their C and Python counterparts. On this work, we propose a methodology to adapt the validated misconceptions in C and Python to Java. Initially, through the analysis of an initial list of 33 misconceptions in C and 28 in Python, we identified and documented in an antipattern format 31 possible misconceptions in Java. We then developed a final term exam, composed of 7 open-ended questions, in which each question was designed to address some of the misconceptions covered in the course (N = 27). Through the analysis of the exams answers (N = 69 students), it was possible to validate 22 of the misconceptions (81%). Also, 6 new misconceptions were identified, leading to a total of 28 valid misconceptions in Java.

CCS CONCEPTS

• **Social and professional topics → CS1; Student assessment;** • **Software and its engineering → General programming languages;** *Patterns.*

KEYWORDS

CS1, Course, Misconception, Concept, Inventory, Java, Student, Assessment, Introductory, Programming, Multiple, Choice, Question

ACM Reference Format:
Ricardo Caceffo, Pablo Frank-Bolton, Renan Souza, and Rodolfo Azevedo. 2019. Identifying and Validating Java Misconceptions Toward a CS1 Concept Inventory. In *Innovation and Technology in Computer Science Education (ITiCSE '19), July 15–17, 2019, Aberdeen, Scotland Uk.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3304221.3319771

1 INTRODUCTION

Learning to program has increasingly become a required skill both in academic and professional [31] areas. However, as explained by Figueiredo & Garcia-Penalvo [15], the difficulties of teaching and learning programming are concerning enough to require the development of special pedagogical strategies.

For these reason, novel educational learning approaches are being developed, adapted and employed inside the classroom. Particularly, the adoption of Active Learning [1] techniques, like the Problem-based learning (PBL) and Peer-Instruction (PI), has been shown to be a positive factor on students' learning and motivation when compared to traditional lecture-based learning [7].

In order to formally measure the impact of these interventions, formal assessment methods are required. One typical example is the Concept Inventory (CI), a multiple-choice questionnaire in which each wrong choice is mapped to a common misconception among students. Usually performed as pre- and post-tests, it allows the assessment of the student's learning gains in a certain period of time.

For introductory programming courses (CS1), misconceptions were identified and documented in the C and Python programming languages with the purpose of supporting the design of CIs. Although there are studies related to misconceptions in the Java language, these misconceptions lack the formality, comprehensiveness, and robustness required to design a CI in this language. Also, the design process to identify and validate misconceptions to support the creation and assessment of a CI, is long and costly [3].

Therefore, on this study we propose to answer the following research questions:

Figure 1. First page of the article "Identifying and Validating Java Misconceptions Toward a CS1 Concept Inventory", published in the *Proceedings of the 24th Annual ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE).*

The methodology employed in the study, as presented in Figure 2, can be organized in the following steps: **misconceptions identification**, in which possible Java misconceptions (N = 31) were identified from previous mapped misconceptions in C [4] (N = 33) and Python [6] (N = 28) and then documented in the antipattern format [7]; **misconceptions validation**, in which an open response test (N = 7 problems) was designed to address some (N = 27) of these misconceptions, also supporting the identification of new ones (*cf.* Section 2) and; the **updated documentation,** in which the validated (N = 21) and new (N = 7)[2] misconceptions were documented in the antipattern format (*cf.* Section 3).

[2] In the article "Identifying and Validating Java Misconceptions Toward a CS1 Concept Inventory" it is explained that 6 new misconceptions were found. However, the misconception JG.5 was incorrectly not classified as a new misconception. Therefore, the correct number of new misconceptions is 7.

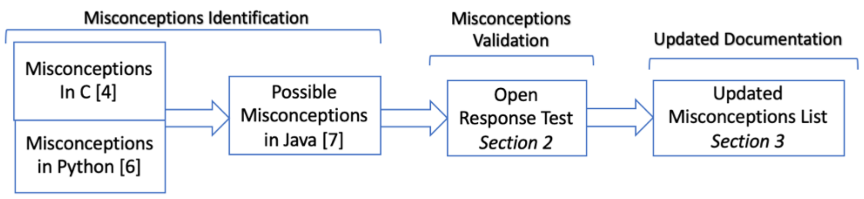
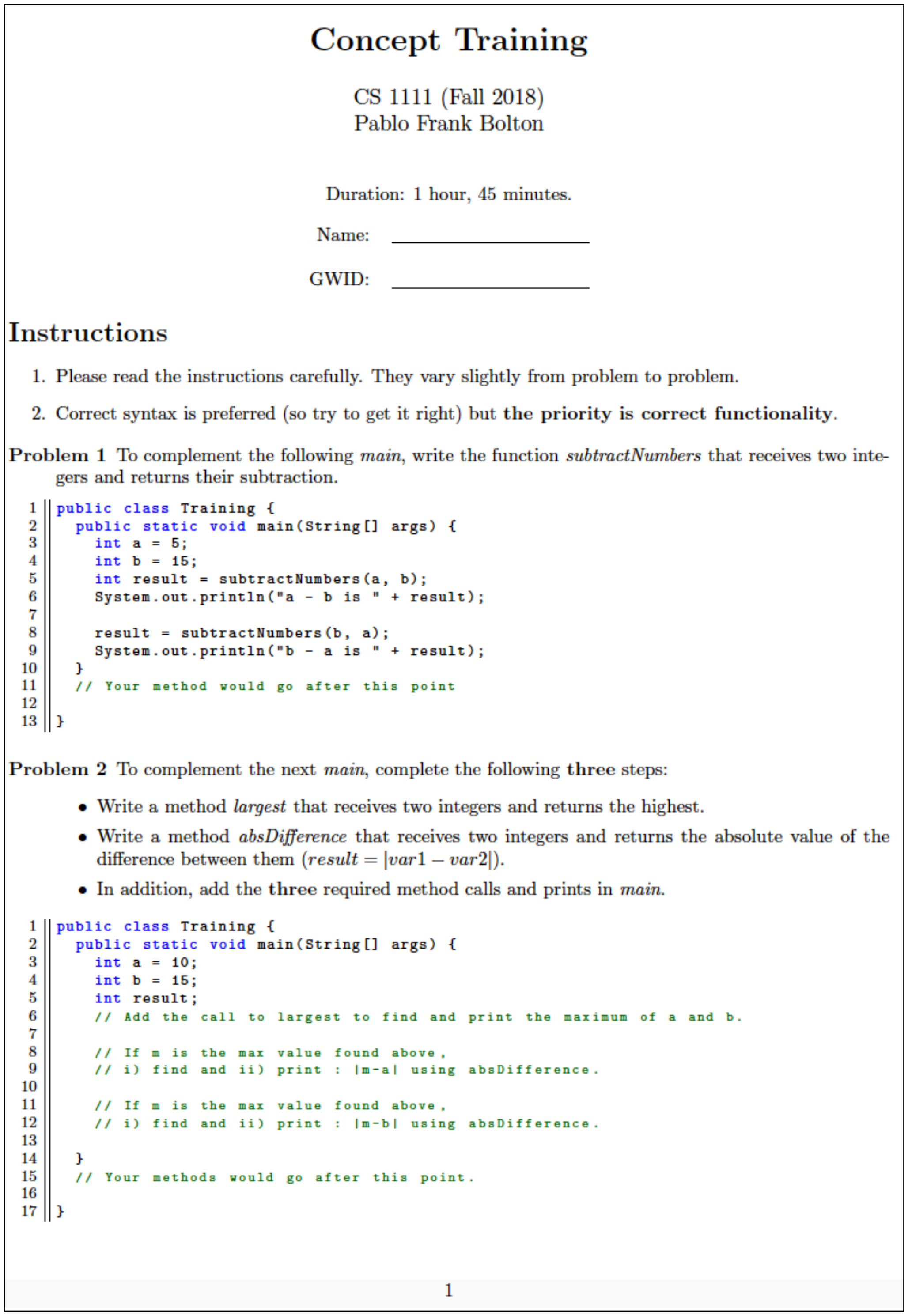

Misconceptions Identification
Misconceptions In C [4]
Misconceptions in Python [6]
Possible Misconceptions in Java [7]
Misconceptions Validation
Open Response Test *Section 2*
Updated Documentation
Updated Misconceptions List *Section 3*

Figure 2. Methodology employed on the study. Initially a set of possible Java misconceptions [7] were identified from previous mapped misconceptions in C [4] and Python [6]. Then an open response test was designed to validate these misconceptions (*cf.* Section 2). The misconceptions found were then documented in the antipattern format (*cf.* Section 3*)*.

2. Open Response Test

The following Figures 3, 4 and 5 presents the Open Response Test, composed of 7 problems, employed in this study. The test was designed by professor Pablo Frank-Bolton, at the George Washington University, and then performed as final exam (N = 69 students) in the fall 2018 by students of the CS 1111 course.



Concept Training

CS 1111 (Fall 2018)
Pablo Frank Bolton

Duration: 1 hour, 45 minutes.

Name: ____________________

GWID: ____________________

Instructions

1. Please read the instructions carefully. They vary slightly from problem to problem.
2. Correct syntax is preferred (so try to get it right) but **the priority is correct functionality**.

Problem 1 To complement the following *main*, write the function *subtractNumbers* that receives two integers and returns their subtraction.

```
public class Training {
  public static void main(String[] args) {
    int a = 5;
    int b = 15;
    int result = subtractNumbers(a, b);
    System.out.println("a - b is " + result);

    result = subtractNumbers(b, a);
    System.out.println("b - a is " + result);
  }
  // Your method would go after this point

}
```

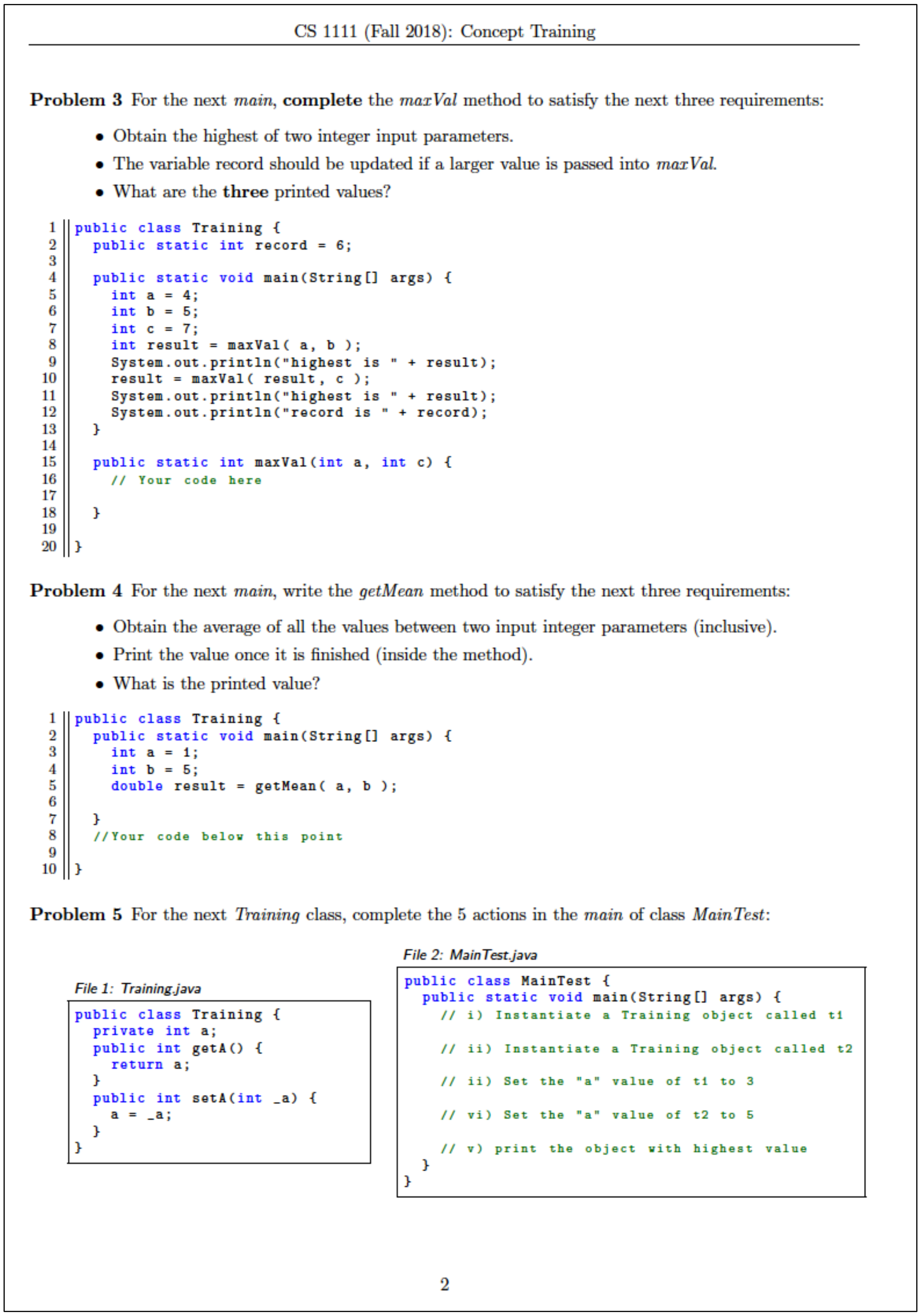
Problem 2 To complement the next *main*, complete the following **three** steps:

- Write a method *largest* that receives two integers and returns the highest.
- Write a method *absDifference* that receives two integers and returns the absolute value of the difference between them ($result = |var1 - var2|$).
- In addition, add the **three** required method calls and prints in *main*.

```
public class Training {
  public static void main(String[] args) {
    int a = 10;
    int b = 15;
    int result;
    // Add the call to largest to find and print the maximum of a and b.

    // If m is the max value found above,
    // i) find and ii) print : |m-a| using absDifference.

    // If m is the max value found above,
    // i) find and ii) print : |m-b| using absDifference.

  }
  // Your methods would go after this point.

}
```

1

Figure 3. Problems 1 and 2 of the open response test employed in this study.



CS 1111 (Fall 2018): Concept Training

Problem 3 For the next *main*, **complete** the *maxVal* method to satisfy the next three requirements:

- Obtain the highest of two integer input parameters.
- The variable record should be updated if a larger value is passed into *maxVal*.
- What are the **three** printed values?

```java
public class Training {
  public static int record = 6;

  public static void main(String[] args) {
    int a = 4;
    int b = 5;
    int c = 7;
    int result = maxVal( a, b );
    System.out.println("highest is " + result);
    result = maxVal( result, c );
    System.out.println("highest is " + result);
    System.out.println("record is " + record);
  }

  public static int maxVal(int a, int c) {
    // Your code here

  }

}
```

Problem 4 For the next *main*, write the *getMean* method to satisfy the next three requirements:

- Obtain the average of all the values between two input integer parameters (inclusive).
- Print the value once it is finished (inside the method).
- What is the printed value?

```java
public class Training {
  public static void main(String[] args) {
    int a = 1;
    int b = 5;
    double result = getMean( a, b );

  }
  //Your code below this point

}
```

Problem 5 For the next *Training* class, complete the 5 actions in the *main* of class *MainTest*:

File 1: Training.java

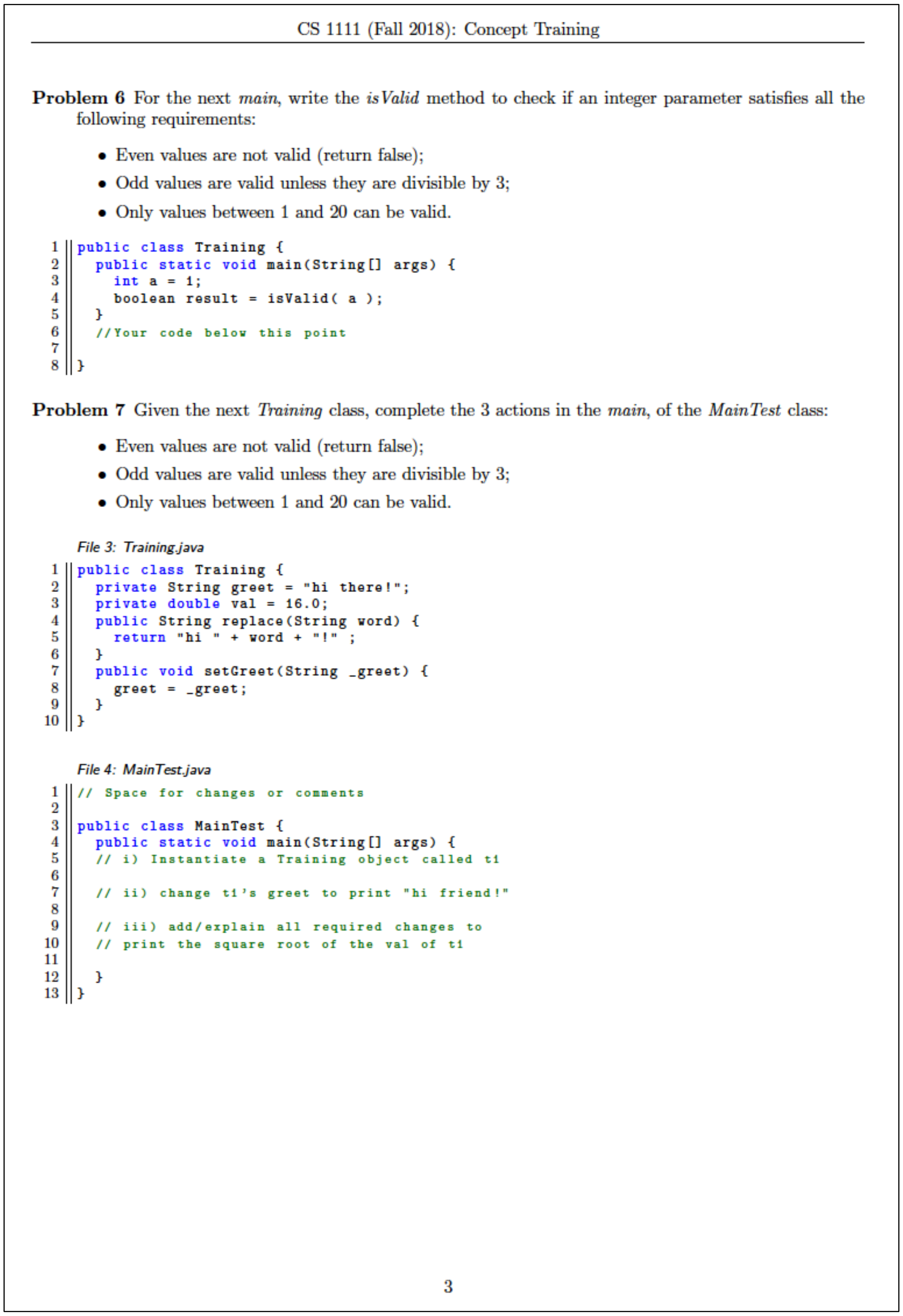
```java
public class Training {
  private int a;
  public int getA() {
    return a;
  }
  public int setA(int _a) {
    a = _a;
  }
}
```

File 2: MainTest.java

```java
public class MainTest {
  public static void main(String[] args) {
    // i) Instantiate a Training object called t1

    // ii) Instantiate a Training object called t2

    // ii) Set the "a" value of t1 to 3

    // vi) Set the "a" value of t2 to 5

    // v) print the object with highest value
  }
}
```

2

Figure 4. Problems 3, 4 and 5 of the open response test employed in this study.


CS 1111 (Fall 2018): Concept Training

Problem 6 For the next *main*, write the *isValid* method to check if an integer parameter satisfies all the following requirements:

- Even values are not valid (return false);
- Odd values are valid unless they are divisible by 3;
- Only values between 1 and 20 can be valid.

```java
public class Training {
  public static void main(String[] args) {
    int a = 1;
    boolean result = isValid( a );
  }
  //Your code below this point

}
```

Problem 7 Given the next *Training* class, complete the 3 actions in the *main*, of the *MainTest* class:

- Even values are not valid (return false);
- Odd values are valid unless they are divisible by 3;
- Only values between 1 and 20 can be valid.

File 3: Training.java

```java
public class Training {
  private String greet = "hi there!";
  private double val = 16.0;
  public String replace(String word) {
    return "hi " + word + "!" ;
  }
  public void setGreet(String _greet) {
    greet = _greet;
  }
}
```

File 4: MainTest.java

```java
// Space for changes or comments

public class MainTest {
  public static void main(String[] args) {
  // i) Instantiate a Training object called t1

  // ii) change t1's greet to print "hi friend!"

  // iii) add/explain all required changes to
  // print the square root of the val of t1

  }
}
```

3

Figure 5. Problems 6 and 7 of the open response test employed in this study.

3. Updated Misconceptions List

3.1 Misconceptions Mapping

Tables 1 to 8 present, for each topic, the previous work documented misconceptions in C [4] (N = 33) and Python [6] (N = 28); the misconceptions identified in Java [7] (N = 31); which of these misconceptions were covered by the open response test (N = 27) and; which misconceptions were validated (N = 21), *i.e.* identified thorough the analysis of the test answers, and also the new ones (N = 7) identified in the process.

Specifically, the columns are:

- Topic: The misconceptions were organized into 8 topics.
    - Topic A: Function Parameter Use and Scope;
    - Topic B: Variables, Identifiers, and Scope;
    - Topic C: Recursion;
    - Topic D: Iteration;
    - Topic E: Structures;
    - Topic F: Pointers;
    - Topic G: Boolean Expressions.
    - Topic H: Classes and objects
- Misconception Description: A brief description of the misconception.
- Misconceptions Identification:
    - Was it identified in the C study [4]?: A **green** cell within a misconception ID indicates the corresponding misconception in C was identified in the study [4]. A **light red** cell indicates the misconception was not identified.
    - Was it identified in the Python [6] study? A **green** cell within a misconception ID indicates the corresponding misconception in C was identified in the study [6]. A **light red** cell indicates the misconception was not identified.
    - Was it identified in the Java [7] study? A **green** cell within a misconception ID indicates the corresponding misconception in C was identified in the study [7]. A **light red** cell indicates the misconception was not identified.

- Misconceptions validation:
    - Was it covered in the open response test? A **green** cell within a problem ID (*e.g* Problem P1) indicates the corresponding misconception was covered by that problem in the open response test. A **light red** cell indicates the misconception was not covered by any problem.
    - Was it identified in the open response test? A **green** cell containing two numbers in the format XX (YY%) indicates the corresponding misconception was identified in XX situations, which corresponds a YY% of the total. Additionally, if the cell contains a misconception ID, like *JA.7 (New),* it indicates that, although that misconception was not previously identified, it was found in some of the open response problems answers. On its turn, a blank **light red** cell indicates the misconception was not found in any answer.

For example, Table 1 relates to Topic A (Function Parameter Use and Scope). In its first row, the misconception "Parameter value set by external source" was initially identified in the C [4] and Python [6] studies, and then documented and mapped as a possible misconception in Java [7]. It was covered by the open response test in problems P1 and P2, but it was not found in any student's answers.

On its turn, the misconception "No return value in a function that returns something" was not identified in any previous studies, therefore not being covered by any open response test question. However, it was found in 21 student's answers, which corresponds to 18.90% of all occurrences of misconceptions found, in all topics (N = 111). This misconception received the ID **JA.7 (New).**

Topic	**Misconception Description**	**Misconceptions Identification**			**Misconceptions Validation**	
		Was it identified in the C study [4]?	*Was it identified in the Python study [6]?*	*Was it identified in the Java study [7]?*	*Was it covered in the open response test?*	*Was it identified in the open response test?*
A: Function Parameter Use and Scope	Parameter value set by external source	CA.1	PA.1	JA.1	Problems P1, P2	
	Parameters passed as if by reference	CA.2		JA.2	Problems P1, P2	
	Attempt to access parameter from outside scope	CA.3	PA.3	JA.3	Problems P1, P2	12 (10.8%)
	Incorrect order of function parameters	CA.4		JA.4	Problems P1, P2	3 (2.70%)
	Function return value not handled by caller function	CA.5	PA.5	JA.5	Problems P1, P2	1 (0.90%)
	Logic error related to parameters when calling function	CA.6		JA.6	Problem P2	3 (2.70%)
	No return value in a function that returns something					**JA.7 (New)** 21 (18.90%)
	No parameters used when calling a method.					**JA.8 (New)** 5 (4.50%)

Table 1. Topic A misconceptions identified and documented in the antipattern format (C [4], Python [6] and Java [7]) and Java misconceptions validated through the open response test. The **green cells** indicate that a misconception was identified, validated, or covered in an open response test related problem. The **light red** cells indicate no misconception was identified, validated or covered by a problem in the open response test.

Topic	Misconception Description	Misconceptions Identification			Misconceptions Validation	
		Was it identified in the C study [4]?	*Was it identified in the Python study [6]?*	*Was it identified in the Java study [7]?*	*Was it covered in the open response test?*	*Was it identified in the open response test?*
B: Variables, Identifiers, and Scope	Attempt to access local variables, except parameters, from outside scope	CB.1	PB.1	JB.1	Problem P3	5 (4.50%)
	Global variables considered local in current scope	CB.2		JB.2	Problem P3	3 (2.70%)
	Parameter mistaken for same-name variable outside function	CB.3	PB.3	JB.3	Problem P3	3 (2.70%)
	Global variables assumed inaccessible from within function	CB.4		JB.4	Problem P3	6 (5.40%)
	Iteration variable used in for statement considered local		PB.5			
	Value that was assigned return value from function that was called with existing variables as parameters will be automatically updated when these parameters are reassigned.		PB.6			
	Wrong order in logical or arithmetic operations.					**JB.7 (New)** 5 (4.50%)

Table 2. Topic B misconceptions identified and documented in the antipattern format (C [4], Python [6] and Java [7]) and Java misconceptions validated through the open response test. The **green cells** indicate that a misconception was identified, validated, or covered in an open response test related problem. The **light red** cells indicate no misconception was identified, validated or covered by a problem in the open response test.

Topic	Misconception Description	Misconceptions Identification			Misconceptions Validation	
		Was it identified in the C study [4]?	*Was it identified in the Python study [6]?*	*Was it identified in the Java study [7]?*	*Was it covered in the open response test?*	*Was it identified in the open response test?*
C: Recursion	Wrong expression used to calculate the return value of a recursive function	CC.1	PC.1	JC.1		
	No recursive call	CC.2	PC.2	JC.2		
	No termination at base case	CC.3	PC.3	JC.3		

Table 3. Topic C misconceptions identified and documented in the antipattern format (C [4], Python [6] and Java [7]) and Java misconceptions validated through the open response test. The **green cells** indicate that a misconception was identified, validated, or covered in an open response test related problem. The **light red** cells indicate no misconception was identified, validated or covered by a problem in the open response test.

Topic	Misconception Description	Misconceptions Identification			Misconceptions Validation	
		Was it identified in the C study [4]?	*Was it identified in the Python study [6]?*	*Was it identified in the Java study [7]?*	*Was it covered in the open response test?*	*Was it identified in the open response test?*
D: Iteration	Improper update of loop counter	CD.1	PD.1	JD.1	Problem P4	1 (0.90%)
	Use of loop result before completion	CD.2	PD.2	JD.2	Problem P4	
	Loop iterates the correct number of times, but over the wrong range.	CD.3	PD.3	JD.3	Problem P4	
	Loop iterates an incorrect number of times.	CD.4	PD.4	JD.4	Problem P4	1 (0.90%)
	Absence of loop (single iteration)	CD.5	PD.5	JD.5	Problem P4	3 (2.70%)
	Loop construction incoherent with remainder of code	CD.6	PD.6	JD.6	Problem P4	2 (1.80%)
	For loop iterating over members of an iterable object is treated as a for loop over a range () instance.		PD.7			
	For loop iterating over range() treated as for loop iterating over members of an iterable object.		PD.8			

Table 4. Topic D misconceptions identified and documented in the antipattern format (C [4], Python [6] and Java [7]) and Java misconceptions validated through the open response test. The **green cells** indicate that a misconception was identified, validated, or covered in an open response test related problem. The **light red** cells indicate no misconception was identified, validated or covered by a problem in the open response test.

Topic	Misconception Description	Misconceptions Identification			Misconceptions Validation	
		Was it identified in the C study [4]?	*Was it identified in the Python study [6]?*	*Was it identified in the Java study [7]?*	*Was it covered in the open response test?*	*Was it identified in the open response test?*
E: Structs	Structs compared by identifier	CE.1		JE.1	Problem P5	
	Struct identifier compared to field	CE.2		JE.2	Problem P5	2 (1.80%)
	Struct accessed as pointer	CE.3				
	Struct field accessed as array index	CE.4				
	Use of "struct" keyword after declaration	CE.5		JE.5	Problem P5	1 (0.90%)
	Using Local variables as member variables of an object.					**JE.6 (New)** 1 (0.90%)
	Value assigned to class reference instead of its member variable.					**JE.7 (New)** 1 (0.90%)

Table 5. Topic E misconceptions identified and documented in the antipattern format (C [4], Python [6] and Java [7]) and Java misconceptions validated through the open response test. The **green cells** indicate that a misconception was identified, validated, or covered in an open response test related problem. The **light red** cells indicate no misconception was identified, validated or covered by a problem in the open response test.

Topic	Misconception Description	Misconceptions Identification			Misconceptions Validation	
		Was it identified in the C study [4]?	*Was it identified in the Python study [6]?*	*Was it identified in the Java study [7]?*	*Was it covered in the open response test?*	*Was it identified in the open response test?*
F: Pointers	Use of & to dereference	CF.1				
	Not dereferencing	CF.2				
	Invalid address assigned to pointer	CF.3				
	Void function returns value	CF.4		JF.4	All problems	1 (0.90%)

Table 6. Topic F misconceptions identified and documented in the antipattern format (C [4], Python [6] and Java [7]) and Java misconceptions validated through the open response test. The **green cells** indicate that a misconception was identified, validated, or covered in an open response test related problem. The **light red** cells indicate no misconception was identified, validated or covered by a problem in the open response test.

Topic	Misconception Description	Misconceptions Identification			Misconceptions Validation	
		Was it identified in the C study [4]?	*Was it identified in the Python study [6]?*	*Was it identified in the Java study [7]?*	*Was it covered in the open response test?*	*Was it identified in the open response test?*
G: Boolean Expressions	Incorrect precedence for boolean operators	CG.1	PG.1	JG.1	Problem P6	3 (2.70%)
	Nested if-statements instead of boolean expression	CG.2	PG.2	JG.2	Problem P6	4 (3.60%)
	Arithmetic expression instead of boolean expression	CG.3		JG.3	Problem P6	6 (5.40%)
	Attempt to evaluate boolean expression through loops	CG.4	PG.4	JG.4	Problem P6	3 (2.70%)
	Condition added after else.					**JG.5 (New)**[3] 1 (0.90%)

Table 7. Topic G misconceptions identified and documented in the antipattern format (C [4], Python [6] and Java [7]) and Java misconceptions validated through the open response test. The **green cells** indicate that a misconception was identified, validated, or covered in an open response test related problem. The **light red** cells indicate no misconception was identified, validated or covered by a problem in the open response test.

[3] The misconception JG.5 was incorrectly not classified in the article "Identifying and Validating Java Misconceptions Toward a CS1 Concept Inventory" [8] as a new misconception.

Topic	Misconception Description	Misconceptions Identification			Misconceptions Validation	
		Was it identified in the C study [4]?	*Was it identified in the Python study [6]?*	*Was it identified in the Java study [7]?*	*Was it covered in the open response test?*	*Was it identified in the open response test?*
H: Use and Implementation of Classes and Objects (Python and JAVA)	Attempt to invoke method outside its class		PH.1			
	Treating method that returns a new instance of the same object as one that changes the instance itself.		PH.2	JH.2	Problem P7	
	Function call preceded by def keyword.		PH.3			
	Class attribute invoked without being imported or with no class specified.		PH.4	JH.4	Problem P7	10 (9.0%)
	Assignment of value to method.		PH.5			
	No *self* keyword to reference instance attributes.		PH.6			
	Attempt to call class attribute through indices.		PH.7	JH.7	Problems P5, P7	3 (2.70%)
	Attempt to change the value of a final variable.			JH.8		
	New class constructed every time a method is needed instead of using an existing class' method.					**JH.9 (New)** 1 (0.90%)

Table 8. Topic H misconceptions identified and documented in the antipattern format (C [4], Python [6] and Java [7]) and Java misconceptions validated through the open response test. The **green cells** indicate that a misconception was identified, validated, or covered in an open response test related problem. The **light red** cells indicate no misconception was identified, validated or covered by a problem in the open response test.

3.2 Misconceptions as Antipatterns

The 28 misconceptions found (*i.e.* marked in the column "*Was it identified in the open response test?*" as **green** cells in tables 1 to 8) are described below. They were documented in the antipattern format, as proposed by El-Attar and Miller [1]. Each misconception has an ID n the format JT.X, in which T is the topic (A to H) and X is a number. The 7 new misconceptions (JA.7, JA.8, JB.7, JE.6, JE.7, JG.5, JH.9) have a "(New)" tag after their IDs.

2.1) Topic A

JA.3

Code:	JA.3
Name:	Parameters accessible outside their scope.
Description:	Assumption that parameters could be accessed from outside their scope, anywhere in the program.

Example:	`1. public static int func1 (int n) {`
	`2.    n = n + 5;`
	`3.    return n;`
	`4. }`
	`5.`
	`6. public static int func2 (int x) {`
	`7.    return x + n;`
	`8. }`
	
	There is an error on line 7, where an attempt is made to access the parameter `N` – which is local to `func1` – from within `func2`, which is outside its scope.

Rationale:	Students consider that the scope of parameter variables is global, and therefore they could be accessed from anywhere in the code.
Consequences:	- The program will not compile or; - If there is a local variable (local or global), anywhere in the code, that has the same name of some parameter, the student could consider the program will use that parameter value instead of the local value, leading to unexpected behavior.
Detection:	**Where:** o Inside a function, when a parameter is used outside the function scope.
	How: o Student wants to access a parameter variable not visible in the current function scope.
Improvement:	Students could be oriented, through examples and exercises, to understand the difference between local and global scope of variables.

JA.4

Code:	JA.4
Name:	Incorrect order of function parameters
Description:	Assumption that programs could identify the context of how variables are used, thus automatically adjusting the parameter order to correctly achieve the goal of the program or function.

Example:

```
public static int subHelper (int a, int b) {
   return a - b;
}

public static void main(String[] args) {
   int a = 5;
   int b = 7;
   System.out.printf("a - b = %d\n",
                        subHelper(a, b));
   System.out.printf("b - a = %d\n",
                        subHelper(a, b));
}
```

Rationale:	Students believe the program will understand the semantics related to some string and/or the variable names and adjust their order properly.
Consequences:	The program will work accordingly to the written order, which can output misleading results.
Detection:	**Where:** ○ When calling a function with multiple parameters.
	How: ○ The function call uses a wrong order for the parameters.
Improvement:	Students should be oriented to verify and compare, for example with a `System.out.printf`, debugging or assertive programming, the parameters values received by a function. Also, examine what would happen when the parameters order is changed.

JA.5

Code:	JA.5
Name:	Not catching the return value from a function.
Description:	Assumption that return values are automatically handled by the caller function.

Example:	`1. public static int squareOf (int n) {`
	`2.   return n * n;`
	`3. }`
	`4.`
	`5. public static void main(String[] args) {`
	`6.    int a = 5;`
	`7.    squareOf(a);`
	`8.    System.out.printf("a = %d\n", a);`
	`9. }`
	
	There is an error in the line 8. The correct code would be:
	`8. a = squareOf(a);`

Rationale:	Students believe the program is intelligent, *i.e.*, it will understand the semantics related to a function call, automatically adjusting the logic and results.
Consequences:	The function's return value will not be received by the calling instance, rendering the function useless.
Detection:	**Where:** o In a function call that returns some value
	How: o The returned value is not correctly handled.
Improvement:	Students should be oriented to always check whether a non-void function's return value is being properly assigned.

JA.6

Code:	JA.6
Name:	Logic error related to parameters when calling a function.
Description:	Presence of logic errors in the function call. The order of values of the parameters are wrong, different than those defined in the function interface/signature.
Example:	Create a main function that prints the result of `2a-b` if `a >= b`. Otherwise the function must print the result of `b - a.` Use the `subHelper` function to process the subtractions.

```java
public static int subHelper (int b, int a) {
   return b - a;
}

public static void main(String[] args) {
   int a = 5;
   int b = 7;
   int r;

   if (a >= b) {
       r = subHelper(a, 2*b);
   } else {
       r = subHelper(b, a);
   }
   System.out.println(r);
}
```

Rationale:	Students believe the program will understand the semantics related to a function call, automatically adjusting its logics and values.
Consequences:	Although the program will compile, it will not run as expected. The function call will return a wrong result, different than planned by the student, leading the program to a wrong (and unexpected) behavior.
Detection:	**Where:** o In a function call.
	How: o The parameter values on their order are not compatible with the function interface/signature.
Improvement:	Students should be oriented, for example with comments, about the purpose and role of each parameter in the function logic. The parameter names could be meaningful, referring to what the parameters are expected to represent.

JA.7 (New)

Code:	JA.7 (New)
Name:	No return value in a method that returns something.
Description:	Although the method signature predicts that it should return a value, no value is returned.
Example:	(see below)

```java
public static int subHelper (int b, int a) {
     result = b - a;
}

public static void main(String[] args) {
    int a = 5;
    int b = 7;
    int r;

    r = subHelper(a, b);
    System.out.println(r);
}
```

Rationale:	Students believe the program will automatically return the calculated/correct value, even though the `return` statement is not present.
Consequences:	The program will not compile.
Detection:	**Where:** o In any method that has a return value in its signature.
	How: o The `return` statement is not present in the method.
Improvement:	Students should be oriented that in all methods that have a return value in its signature the return statement must be present, followed by the correct value to be returned.

JA.8 (New)

Code:	JA.8 (New)
Name:	No parameters used when calling a method that has parameters defined on its signature.
Description:	Although the method signature has parameters defined, no parameters are used when called the method.

Example:	`public static int subHelper (int b, int a) {`
	`    result = b - a;`
	`}`
	
	`public static void main(String[] args) {`
	`   int a = 5;`
	`   int b = 7;`
	`   int r;`
	
	`   r = subHelper();`
	`   System.out.println("b - a = " + r);`
	`}`

Rationale:	Students believe the program will automatically consider the correct/expected parameters values when calling a method, even though no value is passed as parameter.
Consequences:	The program will not compile.
Detection:	**Where:** o In the call of any method that has parameters defined on its signature.
	How: o No parameters are used when calling the method.
Improvement:	Students should be oriented that when calling methods that have parameters defined on their signature, parameters must be included in the call statement.

2.1) Topic B

JB.1

Code:	JB.1
Name:	Local variables, except parameters, accessed outside their scope
Description:	Out of scope assignment, considering or classifying local variables as if they were global variables.

Example:	`public static int func (int a, int b) {`
	`    return a + b + c;`
	`}`
	
	`public static void main(String[] args) {`
	`    int a = 5;`
	`    int b = 7;`
	`    int c = 10;`
	`    int r = func(a,b);`
	`    System.out.println(r);`
	`}`

Rationale:	Students believe the scope of local variables is global, so they could be accessed at any point of the code.
Consequences:	- The program could not compile or; - If there is a local variable in some function that has the same name of another local variable (in a different function), the student could consider the program will share the values of these variables, leading to an unexpected behavior.
Detection:	**Where:** o In any function that has local variables.
	How: o The function tries to access a local variable declared in another function.
Improvement:	Students should be guided about the existing variable scopes in the C language and how they work. Examples to show this might include comparisons between similar codes with different variable names, stressing that the chosen names make no difference regarding their scopes.

JB.2

Code:	JB.2
Name:	Global variables considered as local in the current scope.
Description:	Out of scope assignment, considering or accessing global variables as if they were local to current scope.

Example:

```
public static int max;

public static int func (int a, int b) {
   if (a >= b) {
       B2.max = a;
   } else {
       B2.max = b;
   }
   return B2.max;
}

public static void main(String[] args) {
   int a = 5;
   int b = 7;
   int r;
   B2.max = 10;
   System.out.printf("10 == %d", B2.max);
   r = func(a,b);
   System.out.printf("10 == %d", B2.max);
}
```

The program logic considers the global variable `B2.max` as two independent local variables (related to the `main` and `func` functions). Therefore, the attributions made inside the function `func` would be local, not affecting the value of the `B2.max` variable in the main function.

Rationale:	Students consider that in Java the scope of variables is local, so when a global variable value is changed inside a function, it would not affect other parts of the code.
Consequences:	Although the program will compile, it will not run as expected.
Detection:	**Where:** o In any function that accesses and assigns a value to a global variable.
	How: o The program logic is built upon the assumption that global variables behave like local variables.
Improvement:	Students should be oriented about the existing variable scopes in the Java language and how they work. Debugging and assertive programming should also be employed.

JB.3

Code:	JB.3
Name:	Parameter confusion with same-name variable outside the function.
Description:	The wrong variable is accessed when a parameter has the same name of some local variable in the caller function.

Example:	`1. public static int addTwoInt (int a, int c) {`
	`2.    return a + c;`
	`3. }`
	`4.`
	`5. public static void main(String[] args) {`
	`6.    int a = 5;`
	`7.    int b = 7;`
	`8.    int c = 10;`
	`9.    int r;`
	`10.`
	`11.   r = addTwoInt(a,b);`
	`12.   System.out.printf("The result value`
	`13.   is %d\n", r);`
	`14. }`
	
	The student expects the parameter `c` in function `addTwoInt` would have the value of `c` in the main function (10) instead of the value of `b` (7). Therefore, the `printf` on the line 12 would output the number 15 instead of 12 (actual output).

Rationale:	Students consider variables with the same name refer to the same variable and memory values.
Consequences:	The program will not run as expected, leading to unexpected behavior.
Detection:	**Where:** o In any program that has at least two variables with the same name.
	How: o The program logic is built upon the assumption that variables created in loop statements do not affect variables elsewhere in the scope.
Improvement:	Students should be oriented about the existing variable scopes in the Java language and how they work. Debugging and assertive programming should also be employed.

JB.4

Code:	JB.4
Name:	Global variables not accessible inside a function.
Description:	The scope of any global variable is assumed to be invalid within the functions of a program.

Example:	`public static int max = 10;`
	
	`public static void compare (int a) {`
	`    if (a > max) {`
	`       System.out.printf("%d is greater than`
	`        %d\n", a, max);`
	`    } else {`
	`       System.out.printf("%d is not`
	`        greater  than %d\n", a, max);`
	`    }`
	`}`
	
	`public static void main(String[] args) {`
	`  int a = 5;`
	`  compare(a);`
	`}`
	
	In this example, students would say the program fails to compile as the `max` variable cannot be accessed inside the `compare` function.

Rationale:	Students believe functions are unaware of external variables not passed as parameters. As a global variable is declared outside the scope of any given function, it follows that no function would be able to access it.
Consequences:	Students would feel compelled to pass global variables as arguments to functions, or even to avoid their use altogether.
Detection:	**Where:** o Any program making use of global variables.
	How: o Student fails to use a global variable, replaces it with a local counterpart, or passes it as an argument.
Improvement:	Students should be oriented to understand that any variable declared outside a function (a global variable) is accessible throughout the program.

JB.7 (New)

Code:	JB.7 (New)
Name:	Wrong order in logical or arithmetic operations.
Description:	The order of operands and operators is wrong.

Example:	`public static int maxVal(int a, int c){`
	`    int result = 0;`
	`    result -= a,c;`
	`    return result;`
	`}`

Rationale:	The student believes that the compiler will automatically deduce the correct order of operands and operators in a way similar to the use of the operation-with-assignment operations (*e.g.* if `result += 1;` is equal to `result = result + 1;` then `result -= a,c;` should be the same as `result = a - c;`).
Consequences:	In most cases the program will not compile, but in rare cases the operation will result in an unexpected value.
Detection:	**Where:** o Any program where an arithmetic or logical operation is used.
	How: o The order of operands and operators is wrong.
Improvement:	Students should be shown the proper use of assignment operators in Java and their difference to multi-operand logical or arithmetic operations

2.3) Topic C

None.

2.4) Topic D

JD.1

Code:	JD.1
Name:	Improper update of a loop counter
Description:	Wrong update of a loop counter, leading to an incorrect number of loop iterations.

Example:	`1. int i = 0;`
	`2. int sum = 0;`
	`3. while (i < 10) {`
	`4.     sum = sum + i;`
	`5.      i = 1;`
	`6. }`
	`7. System.out.println(sum);`
	
	The issue is shown on line 5 (the correct counter update would be `i += 1`).

Rationale:	Students don't understand how the loop counter should be changed (increased or decreased) to reach the number of iterations desired.
Consequences:	Infinite loop or wrong results, if the output uses values calculated inside the loop.
Detection:	**Where:** o In any loop structure (`for`, `while` or do `while`)
	How: o The loop counter is not updated or is wrongly updated.
Improvement:	Students should be able to check (*e.g.* through the debug or a `print`) the counter value at each iteration. In this way they would be able to realize the counter is not updating as expected

JD.4

Code:	JD.4
Name:	Wrong flow control in a loop
Description:	The number of times a loop is executed is wrong.

Example:	Write a program that prints all integers, from 0 to 9:
	`int i;`
	`int sum = 0;`
	`for (i = 0; i <= 10; i++) {`
	`    sum += i;`
	`}`
	`System.out.printf("The sum is %d\n", sum);`

Rationale:	Students does not understand how to construct the loop structure (counter initialization or stop condition) to support a specific number of iterations. Related to the former, students do not understand how to properly instance a loop.
Consequences:	The loop behavior will not be as expected. The number of iterations will be greater or lower than it should be.
Detection:	**Where:** o *For* iterations: The issue can be found in how the loop counter is instanced, specifically the stop condition, and the starting index and step attribute if either is declared. o *While* iterations: The issue can be found before the *while* structure declaration, when the loop control variable is initialized; it also can be found in the *while* condition, that determines the condition to the loop be executed and; it can be found in the *while* conditional code, that can have a wrong control variable increment – or even is absence.
	How: To detect this misconception is necessary the analysis of the loop intention, *i.e.*, the determination of what the loop is supposed to do (or calculate) and how many times it is executing the iteration.
Improvement:	Students should be oriented to check if the loop control variable was correctly initialized and incremented. Also, to check if the loop stop condition was correctly defined. To correctly control how many times the loop is executed, students could insert a *print* statement inside the loop block, printing the value of the loop control variables.

JD.5

Code:	JD.5
Name:	No loop, only one simple iteration
Description:	No loop structure is declared, where one would be necessary for the program to function as expected.

Example:	`1. public static int isDivisibleBy (int n, int x) {`
	`2.    if (n % x == 0) {`
	`3.       return 1;`
	`4.    } else {`
	`5.       return 0;`
	`6.    }`
	`7. }`
	`8.`
	`9. public static void main(String[] args) {`
	`10.   int x = 51, isDivisible;`
	`11.   isPrime= isDivisibleBy(x, 2);`
	`12.   if (isPrime == 1) {`
	`13.      System.out.printf("Number %d is`
	`14.       prime \n", x);`
	`15.   } else {`
	`16.      System.out.printf("Number %d is`
	`17.        not prime\n", x);`
	`18.   }`
	`19. }`
	
	In this example, the call to `isDivisibleBy` is made only once (line 11), without any iteration. The correct code would check all numbers in the *2..49* range, looking for a number that divides `x`. If no number is found, then `x` is prime. Otherwise, `x` is not prime.

Rationale:	The call to `isDivisibleBy` is written upon the assumption that the loop statement and its definitions will be automatically imposed by the compiler.
Consequences:	In most of the cases the loop absence will make the program not work as expected. In specific situations (when the loop would iterate only one time) the program will work successfully.
Detection:	**Where:** o Anywhere throughout the code where it can be interpreted an intention of a repetition structure.
	How: o The code logic requires some block to be repeated several times. However, the program does not have any repetition loop statement.
Improvement:	Students should be oriented on how the loop statement is interpreted by the compiler and that is mandatory to use a loop command (`for`, `while` or `do while`) to have a loop.

JD.6

Code:	JD.6
Name:	Loop construction does not consider the logic and its connection with other parts of the code.
Description:	Although the loop is internally well constructed, there is a logic error when the big picture is analyzed, *i.e.*, how the loop interacts with the code before and after it.

Example:	`1. public static int isDivisibleBy (int n, int x) {`
	`2.    if (n % x == 0) {`
	`3.       return 1;`
	`4.    } else {`
	`5.       return 0;`
	`6.     }`
	`7. }`
	`8. public static void main(String[] args) {`
	`9.    int x = 17, c = 2;`
	`10.   int foundDivisible = 1;`
	`11.   while((foundDivisible == 0)&& (c < x)) {`
	`12.      foundDivisible = isDivisibleBy(x, c);`
	`13.      c++;`
	`14.   }`
	`15.   if (foundDivisible == 1) {`
	`16.      System.out.printf("Number %d is`
	`17.       prime. \n",  x);`
	`18.   } else {`
	`19.      System.out.printf("Number %d is`
	`20.       prime. \n",  x);`
	`21.   }`
	`22. }`
	
	The `while` conditional expression on line 12 is evaluated as false on the first attempt, leading to a program that classifies all numbers as prime. The correct code is:
	`10.  foundDivisible = 0`

Rationale:	Students build the loop considering it is an independent part of the code, not related to any previous and former code.
Consequences:	The program behavior will not be as expected, leading to logic errors.
Detection:	**Where:** o Whenever a loop is used within the code
	How: o If considered independently, the loop is well constructed. However, when considering the whole function that contains the loop, there is a logic error in the data that is used by the loop (input) or generated by the loop (output) and used in other parts of the code.

Improvement:	Students should be oriented to understand the loop is not an independent entity, thus being influenced by previous code and also influencing the code that is after it.

2.5) Topic E

JE.2

Code:	JE.2
Name:	Field of an Object is compared to the identifier of another.
Description:	Comparison of a specific field of some Object variable against the bare identifier of another.

Example:

```
public class Point {
     int x, y;
}


public static void main(String[] args) {
    Point point1 = new Point();
    Point point2 = new Point();
    point1.x = 3;
    point1.y = 5;
    point2.x = 8;
    point2.y = 17;
    if (point1.x == point2) {
      System.out.printf("The points are equal.\n");
    }
}
```

Rationale:	Students consider that, because they haves specified a field of the first Object, the program automatically knows they would like to access the same field in the other Object variable.
Consequences:	The program will not compile.
Detection:	**Where:** o On the line where a field of some Object is compared against a identifier of another.
	How: o A field of an Object is compared against the reference of another.
Improvement:	Students should be oriented that, in Java, there is no implicit comparison of Objects, thus it is mandatory to specify the fields to be compared.

JE.5

Code:	JE.5
Name:	Object name preceded by its class name after declaration.
Description:	The class name is added before its object's name.

Example:	`public class Point {`
	`    int x, y;`
	`}`
	
	`public static void main(String[] args) {`
	`    Point point1 = new Point();`
	`    Point point1.x = 3;`
	`    Point point1.y = 5;`
	`}`

Rationale:	Students consider it is necessary to repeat the class name everywhere, not just when declaring an object.
Consequences:	The program will not compile.
Detection:	**Where:** o Anywhere an Object is used after it has been declared.
	How: o Student writes the Class name everywhere before using an Object.
Improvement:	Students should be oriented regarding the right context to explicit the names of objects and classes in Java language.

JE.6 (New)

Code:	JE.6 (New)
Name:	Using local variables as member variables of an object
Description:	Parameters or local variables are considered as object variables

Example:

```java
public class Point {
  public static void main(String[] args) {
      int a = 5;
      int b = 3;
       result = subtractNumbers(a, b);
  }

  public static int subtractNumbers(int x, int y){
    this.a = x;
    this.b = y;
    return this.a - this.b;
  }
}
```

Rationale:	Students disregard the fact that `a` and `b` are local variables in main rather than member variables for the Point class. This is different than JA.3 because of the explicit use of “`this`”, making it clear that the student believes the variables are object-owned member variables. Note that the use of “`this`” occurs despite the `static` keyword in the method.
Consequences:	The program will not compile.
Detection:	**Where:** o Anywhere where method parameters or local variables are used.
	How: o An object instance or the use of the `this` keyword are employed to attempt to “access” a local variable or parameter.
Improvement:	Clarify to students the cases where the object instance and dot operator can be used to access member variables.

JE.7 (New)

Code:	JE.7 (New)
Name:	Value assigned to class reference instead of its member variable.
Description:	The student assigns a value to an object reference instead of the object's member variable.
Example:	`public class Value {` `public int x;` `}` `public static void main(String[] args) {` `Value v1 = new Value();` `v1 = 3;` `}`
Rationale:	The student believes that the assignment will automatically be attached to the objects only member variable.
Consequences:	The code will not compile.
Detection:	**Where:** o Anywhere where an object is instantiated with a publicly accessible member variable.
	How: o Assignment is done to the object reference rather than the object's member variable.
Improvement:	Students must be shown proper methods for accessing and assigning values to member variables, either through accessor methods or by using the dot operator.

2.6) Topic F

JF.4

Code:	JF.4
Name:	Void function returns value.
Description:	A void function is able to return a value.

Example:	`public void changeValues (int a) {`
	`    a = a + 20;`
	`    return a;`
	`}`

Rationale:	Students consider the presence of a `return` statement sufficient to determine a function's return type as non-void; they do not realize a function can only return the type determined at the function declaration.
Consequences:	The program will not compile.
Detection:	**Where:** o In the void functions with `return` statement.
	How: o Although the function returns a value, its declaration contains `void` as return instead of the proper returned type.
Improvement:	Students should be instructed in how the role of functions can be used to modularize the programs and what is the correct syntax make a function return a value.

2.7) Topic G

JG.1

Code:	JG.1
Name:	Incorrect precedence for boolean operators.
Description:	Order of precedence of boolean expression is different than the one was intended.

Example:	Consider the following definition: *A leap year is every year that is divisible by 4, with the exception of century years (i.e. years ending in 00), which will only be leap years if they are also divisible by 400).*
	
	`1. public static void main(String[] args) {`
	`2.    int year = 2000;  // It is a leap year`
	`3.`
	`4.    if ((year % 400 == 0 || year % 4 == 0)`
	`5.     && (year % 100 != 0)) {`
	`6.        System.out.printf("It is a leap year!\n");`
	`7.    } else {`
	`8.        System.out.printf("It is not a`
	`9.          leap year!\n");`
	`10.   }`
	`11. }`
	
	Lines 4 and 5 contain an error. The correct expression would be
	
	`4.     if ((year % 400 == 0) ||`
	`5.     (year % 4 == 0 && year % 100 != 0))`

Rationale:	Students either consider that boolean expressions can be directly transcribed from English, or fail to examine the correct order of precedence followed by the Java interpreter.
Consequences:	The expression will not be evaluated correctly and the program will behave in an unexpected manner.
Detection:	**Where:** o Whenever boolean expressions are used, in particular those involving more than one boolean operator.
	How: o Students either fail to parenthesize the expression, or do so inappropriately.
Improvement:	Students should be oriented to review the order of precedence of C operators and, when writing boolean expressions, to ascertain that they will be evaluated by the program in the desired order.
Feedback:	Remember students that, in Java, there is an order in which boolean expressions are evaluated. Parenthesized expressions are evaluated first; then, in this order, the `not`, `and`, and `or` operators.

JG.2

Code:	JG.2
Name:	Nested if-statements where a single boolean expression could have been used.
Description:	A boolean expression is written as an `if-else` nested sequence.

Example:	`1. public static void main(String[] args) {`
	`2.    int result = 0;`
	`3.    int a = 100;`
	`4.    int b = 200;`
	`5.    int c = 300;`
	`6.`
	`7.    if (a > 100) {`
	`8.        if (b < 400) {`
	`9.            result = 1;`
	`10.       } else {`
	`11.           if (c == 300) {`
	`12.               result = 1;`
	`13.           }`
	`14.       }`
	`15.   }`
	`16.`
	`17.   if (result == 1) {`
	`18.       System.out.println("True sentence!\n");`
	`19.   } else {`
	`20.       System.out.println("False sentence!\n");`
	`21.   } }`
	
	The same code above should be written as:
	`7. if ((a && (b || c)):`
	`8.     # Do something`

Rationale:	Students do not know how to evaluate multiple-variable boolean expressions, and so they build a sequence of `if` and `else` statements, each containing a single variable to be evaluated.
Consequences:	Although the program will still work correctly with a sequence of `if` and `else` statements (*i.e.* this is not an error), longer boolean expressions will lead to needlessly complex code, which is difficult to read, understand, and debug.
Detection:	**Where:** o Whenever a boolean expression must be evaluated within the code. **How:** o The boolean expression is evaluated through a sequence of `if` and `else` statements.
Improvement:	Students should be oriented to understand how boolean expressions are evaluated in Java language, and also how to use the proper operators (`or`, `and`, `not`, and parentheses), to build these expressions. In addition, the equivalence to nested if-statements could be explored.

JG.3

Code:	JG.3
Name:	Arithmetic expression instead of a boolean expression.
Description:	A logic problem is evaluated through arithmetic instead of a boolean expression.

Example:	Consider the variables `a`, `b`, and `c`, representing statements that can be true or false (meaning each variable would be 1 or 0). Write a boolean expression that evaluates whether at least two statements are true:
	
	`(...)`
	`if (a + b + c == 2) {`
	`    System.out.printf("2 true statements!\n");`
	`}`
	
	The correct statement is
	
	`(...)`
	`if ((a && b) || (a && c) || (b && c)) {`
	`    # Do something`
	`(...)`

Rationale:	Students do not know how to construct a boolean expression with multiple statements. So, they use some arithmetic property related to the problem as a shortcut.
Consequences:	Although the program will work correctly, it is not a universal solution, *i.e,* depends on some arithmetical characteristic of the problem or expression.
Detection:	**Where:** o Whenever a boolean expression with multiple statements must be evaluated within the code.
	How: o A logic problem is solved through a specific arithmetic property instead of a boolean expression.
Improvement:	Students should be oriented to understand how boolean expressions are evaluated in Java language, and also how to use the proper operators (`or`, `and`, `not`, and parentheses), to build these expressions.

JG.4

Code:	JG.4
Name:	Attempt to evaluate a boolean expression through loop iterations.
Description:	A loop statement is used where a boolean expression would need to be evaluated only once.
Example:	Consider the variables `a`, `b`, and `c`, of type bool, which represent statements that can be true or false. Write a boolean expression that evaluates whether at least two statements are true: `(...)` `while (a && b || c) {` `# Do something` `(...)` The correct statement is: `(...)` `if ((a && b) || (a && c) || (b && c)){` `# Do something` `(...)`
Rationale:	Students do not know how to construct a boolean expression composed of a sequence of multiple statements. They do, however, have the notion that the desired boolean expression would comprise a sequence of statements that have some correlation among them. Therefore, students conclude that a loop iteration will somehow support this approach.
Consequences:	The boolean expression will not be correctly evaluated and the program will not work properly. Also, as the loop contains logical errors, there is a chance that the loop condition will always evaluate as true, leading to an infinite loop error.
Detection:	**Where:** o Whenever a boolean expression with multiple statements must be evaluated within the code.
	How: o A logic problem is solved through a sequence of loop iterations, rather than a boolean expression.
Improvement:	Students should be oriented to understand how boolean expressions are evaluated in Java language, and also how to use the proper operators to build these expressions.

JG.5 (New)

Code:	JG.5 (New)
Name:	Condition added after `else`.
Description:	After an else statement, students include a condition (logical or a boolean expression), without the `if`.

Example:	`(...)`
	`if (a > b) {`
	`    return a;`
	`else (b > a) {`
	`    return b;`
	`}`

Rationale:	Students consider the `else` statement implicitly has an if statement after it. Therefore, it would be required to include a boolean expression after each `else`, which would behave similarly to an `else if` statement.
Consequences:	The program will not compile. Also, the program logic may be compromised, since the student considers that the sequence of `if` and `else` statements behaves differently than expected.
Detection:	**Where:** o In a sequence of `if` and `else` statements.
	How: o Students include a condition (logical or a boolean expression) after the `else` statement.
Improvement:	Students should be oriented to understand how the `if` and `else` conditional statements work, and also how Java organizes blocks of commands.

2.8) Topic H

JH.4

Code:	JH.4
Name:	Method invoked without being imported and with no class specified.
Description:	A method of a given class (and therefore not built-in) is invoked, without having previously been imported or without being called from a class or instance thereof.

Example:	`1. double i, sum = 0;`
	`2. for (i = 0; i < 10; i++) {`
	`3.    sum = sum + sqrt(i);`
	`4. }`
	
	In this example, the `sqrt` method (presumably from the `Math` library) is called without having previously been imported, leading to a `java.lang.Error` exception.
	
	This code will only function if, before line 3, the method is imported in a statement such as
	
	`sum = sum + Math.sqrt(i);`

Rationale:	The Java interpreter is capable of calling an attribute simply by its name, without being told where to look for it.
Consequences:	A `java.lang.Error` exception.
Detection:	**Where:** o Whenever attributes (particularly non-built-in functions) are used
	How: o The attribute is called by its identifier without being imported itself or called from an imported class.
Improvement:	Students should be oriented to study the built-in functions in Java and understand that other functions or attributes need to be imported.

JH.7

Code:	JH.7
Name:	Attempt to call class attributes through indices.
Description:	An object is instanced, and then one attempts to change the values of its attributes with bracket operators and indices, as if the object was a list or similar structure.

Example:	`1. public class Student {`
	`2.    int age;`
	`3.    String name;`
	`4.    int id;`
	`5. }`
	`6. public static void main(String[] args) {`
	`7.    Student s1 = new Student(23, "Renan", 9999);`
	`8.    s1[2] = 4567;`
	`9. }`
	
	Line 8 will raise a `java.lang.Error` exception, as `s1` is an instance of the `Student` class, which does not support indexing. The correct code would be:
	
	`8. student.id = 4567`

Rationale:	Students consider that class attributes may be referenced and assigned through subscript indices (when no support for indexing has been implemented), particularly when said attributes are consistently defined in a specific order (*i.e.* in the constructor method).
Consequences:	A `java.lang.Error` exception will be raised.
Detection:	**Where:** o Whenever new classes are designed and used.
	How: o An instance of that class is called using the bracket operator and an index, in an attempt to access an attribute.
Improvement:	Students should be oriented to understand how class attributes work and how they are different from list indices.

JH.9 (New)

Code:	JG.9 (New)
Name:	New class constructed every time a method is needed instead of using an existing class' method.
Description:	Right before an object method will be called, the object is instantiated again, with the same features.

Example:	`public class Student {`
	`   int age;`
	`   String name;`
	`   int id;`
	
	`  public int bornYear() {`
	`        int currentYear =`
	`        Calendar.getInstance().get(Calendar.YEAR);`
	`        return currentYear - this.year;`
	`  }`
	`}`
	`public static void main(String[] args) {`
	`   Student s1 = new Student(23, "Renan", 9999);`
	`   Student s2 = new Student(56, "Bob", 9999);`
	`   (…)`
	`  s1 = new Student(23, "Renan", 9999);`
	`  print ("Student was born in" + s1.bornYear());`
	`}`

Rationale:	Students consider the object, because it was created earlier in another part of the code, does not exist (or may no longer exist) in memory. Therefore, the object must be instantiated again.
Consequences:	Although the code will compile, the program execution can be different than the expected if not all attributes from the original object are restored.
Detection:	**Where:** o Right before an object's method is called.
	How: o The object is instantiated again.
Improvement:	Students should be oriented to understand how objects are managed in memory and how the garbage collector works.

4. Acknowledgements

This research is supported by grant #2014/07502-4, São Paulo Research Foundation (FAPESP) and grant #2013/08293-7, São Paulo Research Foundation (FAPESP). The opinions, hypotheses and conclusions or recommendations expressed in this material are the responsibility of the author (s) and do not necessarily reflect the views of FAPESP. Additional support was provided by the Brazilian Federal Agency for Support and Evaluation of Graduate Education (CAPES), the National Counsel of Technological and Scientific Development (CNPq), under grant 309794/2017-0 and the University of Campinas (Unicamp/FAEPEX). We also would like to thank Dr. Breno de França for his support on this research.

References

[1] Mohamed El-Attar and James Miller. 2006. Matching Antipatterns to Improve the Quality of Use Case Models. In *Proceedings of the 14th IEEE International Requirements Engineering Conference* (RE '06). IEEE Computer Society, Washington, DC, USA, 96-105. DOI=http://dx.doi.org/10.1109/RE.2006.42

[2] CACEFFO, R.; WOLFMAN, S.; BOOTH, K. 2016. Developing a Computer Science Concept Inventory for Introductory Programming. *In Proceedings of the 47th ACM Technical Symposium on Computing Science Education (SIGCSE '16)*. ACM, New York, NY, USA, 364-369. DOI=http://dx.doi.org/10.1145/2839509.2844559

[3] G. L. Herman, M. C. Loui, and C. Zilles. Creating the digital logic concept inventory. In Proceedings of the 41st ACM Technical Symposium on Computer Science Education, SIGCSE ’10, pages 102–106, New York, NY, USA, 2010. ACM

[4] CACEFFO, R. FRANÇA, B.; GAMA, G.; BENATTI, R.; APARECIDA, T.; CALDAS, T.; AZEVEDO, R. (2017) An Antipattern Documentation about Misconceptions related to an Introductory Programming Course in C. *In Technical Report 17-15, Institute of Computing, University of Campinas, SP, Brasil. 42 pages*. October, 2017.

[5] V. L. Almstrum, P. B. Henderson, V. Harvey, C. Heeren, W. Marion, C. Riedesel, L.-K. Soh, and A. E. Tew. Concept inventories in computer science for the topic discrete mathematics.

[6] GAMA, G.; CACEFFO, R.; SOUZA, R.; BENNATI, R.; APARECIDA, T.; GARCIA, I.; AZEVEDO, R. (2018) An Antipattern Documentation about Misconceptions related to an Introductory Programming Course in Python. *In Technical Report 18-19, Institute of Computing, University of Campinas, SP, Brasil. 106 pages*. November, 2017.

[7] SOUZA, R.; CACEFFO, R.; FRANK-BOLTON, P; AZEVEDO, R. (2018). An Antipattern Documentation about Possible Misconceptions related to Introductory Programming Courses (CS1) in Java. In Technical Report 18-20, Institute of Computing, University of Campinas, SP, Brasil. 42 pages. December, 2018

[8] CACEFFO, R.; FRANK-BOLTON, P.; R, SOUZA.; AZEVEDO, R. (2019). Identifying and Validating Java Misconceptions Toward a CS1 Concept Inventory. *In Proceedings of the 24th Annual ACM Conference on Innovation and Technology in Computer Science Education, ITCSE'19, July 15-17, 2019, Aberdeen, Scotland,*
ACM ISBN 978-1-4503-6301-3/19/07 https://doi.org/10.1145/3304221.3319771

[9] CACEFFO, R.; WOLFMAN, S.; BOOTH, K.; GAMA, G.; GARCIA, I.; CALDAS, T.; AZEVEDO, R. (2018) An exploratory questionnaire to support the identification and assessment of misconceptions in CS1 courses based on C programming language. In

Technical Report 18-16, Institute of Computing, University of Campinas, SP, Brasil. 41 pages. October, 2018

[10] CACEFFO, R..; GAMA, G.; BENATTI, R.; APARECIDA, T.; CALDAS, T.; AZEVEDO, R. (2018) A Concept Inventory for CS1 Introductory Programming Courses in C. *In Technical Report 18-06, Institute of Computing, University of Campinas, SP, Brasi*l. 107 pages. March, 2018