An Antipattern Documentation about Possible Misconceptions related to Introductory Programming Courses (CS1) in Java

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

Technical Report - IC-18-20 - Relatório Técnico
December - 2018 - Dezembro

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

The contents of this report are the sole responsibility of the authors.
O conteúdo deste relatório é de única responsabilidade dos autores.

An Antipattern Documentation about Possible Misconceptions related to Introductory Programming Courses (CS1) in Java

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

Abstract: 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. Each incorrect choice corresponds to a specific misconception – an inaccurate line of thought students often follows. **In previous work** we identified misconceptions for Introductory Programming Courses (CS1) in C and Python programming languages. Based on these misconceptions, **in this work** we mapped and documented **possible** misconceptions that could be found in CS1 courses based on JAVA programming language. We developed Java programming codes in order to check if the misconceptions detected in C and Python also could occur in Java language. Also, we used these codes to identify possible new misconceptions in Java, exploring its syntax and object-oriented paradigm. Finally, the misconceptions identified (N=34) were classified into 8 programming topics and documented through an Antipattern structure, composed by: code (a label to identify the misconception); name; description (a brief explanation about the misconception); example; rationale (a possible reason why the misconception happens); consequences; detection (where and how the misconception happens); and improvement (solutions in order to prevent the misconception). **Future work** relates to the validation of the misconceptions identified on this work, through CS1 specialists and students, following the CI design process proposed by Almstrum *et al.*

Keywords: Misconception, Introductory Programming, Concept Inventory, Antipattern, JAVA,

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 [3]. Each incorrect choice corresponds to a specific misconception – an inaccurate line of thought students often follows.

This Technical Report 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] and Python [6] programming languages.

Based on these misconceptions, **in this work** we mapped and documented possible misconceptions that could be found in CS1 courses based on JAVA programming language. We developed Java programming codes in order to check if the misconceptions detected in C and Python could happen in Java language. Also, we used these codes to identify possible new misconceptions in Java, exploring its syntax and object-oriented paradigm.

Finally, the misconceptions identified (N=34) were classified into 8 programming topics and documented through an antipattern structure [1], composed by: code (a label to identify the misconception); name; description (a brief explanation about the misconception); example; rationale (a possible reason why the misconception happens); consequences; detection (where and how the misconception happens); and improvement (solutions in order to prevent the misconception).

This Technical Report is organized as follows: section 2 presents the methodology employed on this study; Sections 3 describes the antipattern structure used to document the misconceptions and; section 4 presents the conclusions and the future work steps.

2. Methodology

Figure 1 illustrates the methodology employed on this study, based on Almstrum *et at.* [5] process:

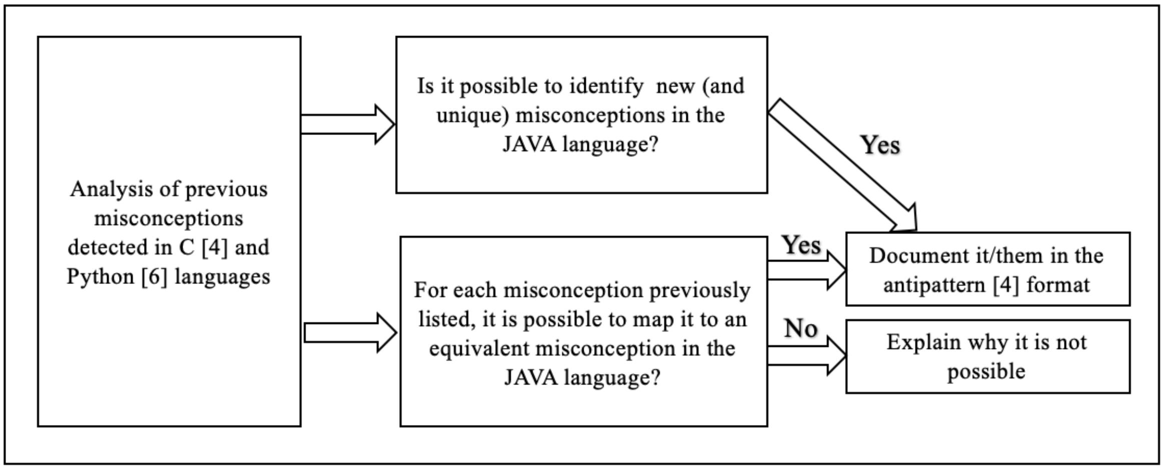
Analysis of previous misconceptions detected in C [4] and Python [6] languages
Is it possible to identify new (and unique) misconceptions in the JAVA language?
Yes
For each misconception previously listed, it is possible to map it to an equivalent misconception in the JAVA language?
Yes
No
Document it/them in the antipattern [4] format
Explain why it is not possible

Figure 1. Methodology employed on this study. When possible, the misconceptions identified on previous work in C [4] and Python [6] programming languages were mapped to the JAVA language and then documented in the antipattern [1] format.

A **limitation** of this study is that it identifies only **possible** misconceptions that could be found in CS1 courses based on JAVA programing language. However, we believe the list and description of misconceptions here documented can be an important starting point for further researches, in which the misconceptions will be validated with CS1 instructors and students.

Table 1 presents the list of misconceptions and is organized in the following way: *Topic* column presents the 8 programming topics in which the misconceptions were grouped, labeled from A to H; *Description* column presents a brief description of each misconception and; *ID* column presents the misconception ID for each programming language (C, Python and JAVA). The programming topics are:

- A: Function Parameter Use and Scope
- B: Variables, Identifiers, and Scope
- C: Recursion
- D: Iteration
- E: Structures
- F: Pointers
- G: Boolean Expressions
- H: Use and Implementation of Classes and Objects *(Python and JAVA)*

The misconception ID follows the format *"LT.N"*, in which L is the programming language ("C" for C, "P" for Python and "J" for Java), T is the topic ("A" to "H") and N is a sequential number. For example, the first misconception in topic A was defined in the C previous work [4] as CA.1[1]; as PA.1 in the Python previous work [6]; and as JA.1 on this work. A grey box indicates the misconception is not present on the respective programming language.

Topic	**Description**	**ID**		
		C [4]	*Python[6]*	*Java* *This Work*
A: Function Parameter Use and Scope	Parameter value set by external source	CA.1	PA.1	JA.1
	Parameters passed as if by reference	CA.2		JA.2
	Attempt to access parameter from outside scope	CA.3	PA.3	JA.3
	Incorrect order of function parameters	CA.4		JA.4
	Function return value not handled by caller function	CA.5	PA.5	JA.5
	Logic error related to parameters when calling function	CA.6		JA.6
B: Variables, Identifiers, and Scope	Attempt to access local variables, except parameters, from outside scope	CB.1	PB.1	JB.1
	Global variables considered local in current scope	CB.2		JB.2
	Parameter mistaken for same-name variable outside function	CB.3	PB.3	JB.3
	Global variables assumed inaccessible from within function	CB.4		JB.4
	Iteration variable used in for statement considered local		PB.5	

[1] Considering the first misconceptions were identified for the C programming language – and the format LT.N was not yet defined at that point – the work [4] presents the misconceptions in the format *T.N* (Topic.Number).

	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	
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
D: Iteration	Improper update of loop counter	CD.1	PD.1	JD.1
	Use of loop result before completion	CD.2	PD.2	JD.2
	Loop iterates the correct number of times, but over the wrong range.	CD.3	PD.3	JD.3
	Loop iterates an incorrect number of times.	CD.4	PD.4	JD.4
	Absence of loop (single iteration)	CD.5	PD.5	JD.5
	Loop construction incoherent with remainder of code	CD.6	PD.6	JD.6
	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	
E: Structures	Structs compared by identifier	CE.1		JE.1
	Struct identifier compared to field	CE.2		JE.2
	Struct accessed as pointer	CE.3		
	Struct field accessed as array index	CE.4		
	Use of "struct" keyword after declaration	CE.5		JE.5
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
	Parameters passed by reference	CF.5		~~JF.5~~[2]
G: Boolean Expressions	Incorrect precedence for boolean operators	CG.1	PG.1	JG.1
	Nested if-statements instead of boolean expression	CG.2	PG.2	JG.2
	Arithmetic expression instead of boolean expression	CG.3		JG.3
	Attempt to evaluate boolean expression through loops	CG.4	PG.4	JG.4
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
	Function call preceded by def keyword.		PH.3	
	Class attribute invoked without being imported or with no class specified.		PH.4	JH.4
	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
	Attempt to change the value of a final variable.			JH.8

Table 1. Misconceptions identified in previous work (C [4] and Python [6] programming languages) and in the current work (Java programming language).

[2] We identified the misconception JF.5 is equivalent to the JA.2. Therefore, although there is this misconception in JAVA, it is only considered and described once.

3. Misconceptions as Antipatterns

As we explain in [4], the antipattern [1] format method shows the programmer how to achieve a good solution from a fallacious solution and how to avoid the specious solution in the future. In the context of this research, the goal is to document the misconceptions found, also describing how to avoid them to happen in the future.

The antipattern format used has the following fields:

- **Name:** The misconception itself
- **Description:** A brief explanation about the Misconception
- **Example:** A code that contains the Misconception
- **Rationale**: The wrong thinking of a student who commits the mistake
- **Consequences**: The output of the code that contains the misconception
- **Detection**: The part of the code where the misconception occurs
- **Improvement**: What the student should do to avoid the mistake

The Appendix 1 shows the misconceptions identified in this work (N=34) documented in the antipattern format.

4. Future Work

Future work relates to the validation of the misconceptions identified in this work through CS1 specialists and students, following the CI design process proposed by [5].

5. Acknowledgments

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) and the University of Campinas (Unicamp). 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.

Appendix 1

The 34 misconceptions found on this work (see Table 1) are described below. They were documented in the antipattern format, as proposed by El-Attar and Miller [1].

JA.1

Code:	JA.1
Name:	Parameter value set by an external source.
Description:	Set the parameter value through an external source outside the function scope.

Example:	`public static int addNumbers (int a, int b) {`
	`   Scanner scanner;`
	`   scanner = new Scanner(System.in);`
	`   a = scanner.nextInt();`
	`   b = scanner.nextInt();`
	`   return a + b;`
	`}`
	
	`public static void main(String[] args) {`
	`   int x = 10;`
	`   int y = 20;`
	`   addNumbers(x, y);`
	`}`

Rationale:	Students do not consider that the parameter of some function already has some value attributed to it. In this situation, they set the parameter value with an external source, such as `scanner.nextInt()`.
Consequences:	The correct parameter value will be lost.
Detection:	**Where:** o On any line of a function that receives a parameter.
	How: o The parameter value is wrongly modified.
Improvement:	Students should be oriented, as an exercise, to verify, for example using `printf`, the value of some parameter inside a function. If they are convinced the parameter has a value already, they are unlikely to assign a value to it.

JA.2

Code:	JA.2
Name:	Parameters passed as if by reference
Description:	Parameters passed as if by reference, instead of by value (*i.e.,* changes in the parameters variables made inside a function would be reflected outside it)

Example:	`public static void addFive (int x) {`
	`    x = x + 5;`
	`}`
	
	`public static void main(String[] args) {`
	`    int x = 10;`
	`    addFive(x);`
	`    System.out.printf("10 plus 5 is" + x);`
	`}`

Rationale:	Students believe the function parameters refer, in memory, to the respective variables in the caller function.
Consequences:	The value of the variable `x` remains the same, producing an unexpected result.
Detection:	**Where:** o In the next instruction that uses a variable previously passed as parameter to some function.
	How: o The value of some parameter is modified and it is expected that such change will reflect on the original variable, inside the caller function.
Improvement:	Students could be guided to verify and compare, for example with a `System.out.printf` or a debugger, the value of some variable before and after a function is called. Using different variable names (in the caller and for the parameter) would also help stress that the variables are not the same.

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:** o When calling a function with multiple parameters.
	How: o 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:** ○ In a function call that returns some value
	How: ○ 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.
	`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:** ○ In a function call.
	How: ○ 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.

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

<table>
<tr><td><b>Code:</b></td><td colspan="2">JB.2</td></tr>
<tr><td><b>Name:</b></td><td colspan="2">Global variables considered as local in the current scope.</td></tr>
<tr><td><b>Description:</b></td><td colspan="2">Out of scope assignment, considering or accessing global variables as if they were local to current scope.</td></tr>
<tr><td><b>Example:</b></td><td colspan="2">

```java
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.

</td></tr>
<tr><td><b>Rationale:</b></td><td colspan="2">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.</td></tr>
<tr><td><b>Consequences:</b></td><td colspan="2">Although the program will compile, it will not run as expected.</td></tr>
<tr><td rowspan="2"><b>Detection:</b></td><td colspan="2"><b>Where:</b>
o In any function that accesses and assigns a value to a global variable.</td></tr>
<tr><td colspan="2"><b>How:</b>
o The program logic is built upon the assumption that global variables behave like local variables.</td></tr>
<tr><td><b>Improvement:</b></td><td colspan="2">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.</td></tr>
</table>

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

<table>
<tr><td><b>Code:</b></td><td>JB.4</td></tr>
<tr><td><b>Name:</b></td><td>Global variables not accessible inside a function.</td></tr>
<tr><td><b>Description:</b></td><td>The scope of any global variable is assumed to be invalid within the functions of a program.</td></tr>
<tr><td><b>Example:</b></td><td>

```
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.

</td></tr>
<tr><td><b>Rationale:</b></td><td>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.</td></tr>
<tr><td><b>Consequences:</b></td><td>Students would feel compelled to pass global variables as arguments to functions, or even to avoid their use altogether.</td></tr>
<tr><td rowspan="2"><b>Detection:</b></td><td><b>Where:</b>
○ Any program making use of global variables.</td></tr>
<tr><td><b>How:</b>
○ Student fails to use a global variable, replaces it with a local counterpart, or passes it as an argument.</td></tr>
<tr><td><b>Improvement:</b></td><td>Students should be oriented to understand that any variable declared outside a function (a global variable) is accessible throughout the program.</td></tr>
</table>

JB.5

We consider this misconception does not exist in Java because the programming language does not contain the iterator `range`.

JB.6

We consider this misconception does not exist in Java because the programming language does not contain the iterator `range`.

JC.1

<table>
<tr><td><b>Code:</b></td><td>JC.1</td></tr>
<tr><td><b>Name:</b></td><td>Wrong formula used to calculate the return value of a recursive function.</td></tr>
<tr><td><b>Description:</b></td><td>A recursive function returns a wrong value, leading to an improperly sequence of recursion iterations.</td></tr>
<tr><td><b>Example:</b></td><td><pre>
1. // My recursive function
2. public int recursive (int a) {
3.   if (a == 0) {
4.   return 1;
5.   } else {
6.   return recursive(++a);
7.   }
8.}
</pre>
In this example, the wrong return value in line 6 would lead to an infinite loop or a <code>RecursionError</code> exception. The correct code to be replaced in line 6 is:
<pre>
6.   return recursive (a - 1)
</pre></td></tr>
<tr><td><b>Rationale:</b></td><td>This misconception relates to the lack of understanding about how recursion works in Java, specifically how recursion instances are stored and retrieved in the stack.</td></tr>
<tr><td><b>Consequences:</b></td><td>The recursive function will present abnormal behavior, leading to an infinite loop or a wrong number of iterations. The program will not run as expected.</td></tr>
<tr><td rowspan="2"><b>Detection:</b></td><td><b>Where:</b>
o Within the recursion function, specifically in the recursive call line.</td></tr>
<tr><td><b>How:</b>
o The recursion never converges to its base case (infinite loop) or it converges, but with a wrong number of iterations.</td></tr>
<tr><td><b>Improvement:</b></td><td>Students should understand that in Java recursion is handled by a stack. In this way, each instance is stored in the current topper position of the stack. To complete the recursion a base case should be satisfied, <i>i.e.</i>, at some point the stack must cease to increase and each stored instance retrieved and resumed <i>(last in, first out scheme</i>). For example, this understanding could be reached through debugging techniques, like the use of the <code>print</code> function to identify the position of each instance in the stack.</td></tr>
</table>

JC.2

Code:	JC.2
Name:	No recursive call
Description:	A (presumably) recursive function lacks a recursive call to itself. The function is called just once, thus not being recursive.

Example:	`// My recursive function`
	`public int recursive (int a) {`
	`    if (a == 0) {`
	`        return 1;`
	`    }`
	`}`

Rationale:	Students do not understand the concept of recursion itself, which leads them to believe the function does not need to contain a recursive call.
Consequences:	As the presumably recursive function contains only the returning value for the stop condition, a compilation error (“Missing return statement”) will be produced.
Detection:	**Where:** o In any recursive function.
	How: o The recursive function lacks a recursive call to itself.
Improvement:	Students should understand that in Java recursion is handled by a stack. In this way, each instance is stored in the current topper position of the stack. To complete the recursion a base case should be satisfied, *i.e.*, at some point the stack must cease to increase and each stored instance retrieved and resumed *(last in, first out scheme*). For example, this understanding could be reached through debugging techniques, like the use of the `printf` function to identify the position of each instance in the stack.

JC.3

Code:	JC.3
Name:	Function does not terminate at the base case.
Description:	A recursive function lacks a base case (stop condition).

Example:	`// My recursive function`
	`public int recursive (int a) {`
	`    return recursive(--a);`
	`}`
	In this example, the misconception would lead to an infinite loop or a `RecursionError` exception.

Rationale:	The recursive function is written upon the assumption that it will indistinctly stop at a given time; the student does not have the knowledge that they need to define stop conditions within the recursive function code to properly allow the recursion stop.
Consequences:	The program will not stop until it reaches a stack overflow error.
Detection:	**Where:** o Within presumably recursive functions.
	How: o The recursive function does not have a stop condition.
Improvement:	Students can be oriented on how the recursive calls mechanism works (aspects such as the memory stack and how the interpreter handles recursive calls), so that they understand the cruciality of stop conditions.

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:** ○ In any loop structure (`for`, `while` or do `while`)
	How: ○ 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.2

Code:	JD.2
Name:	Use of loop result before loop completes.
Description:	Partial results found during loop iterations are used or considered as if they were final.

Example:	Write a program that prints the sum of all numbers from 0 to 9:
	`1. int i;`
	`2. int sum = 0;`
	`3. for (i = 0; i <= 9; i++) {`
	`4.    sum = sum + i;`
	`5.    System.out.printf("The total sum is %d\n", sum);`
	`6. }`
	`7. System.out.printf("The total sum is %d\n", sum);`
	For each iteration the partial result is unnecessarily printed (line 5).

Rationale:	Students feel inclined to group all code related to a given loop inside the body of that loop.
Consequences:	The loop will not produce the expected result.
Detection:	**Where:** ○ Wherever a loop is used and there are operations related to its final result.
	How: ○ Students include in the body of the loop pieces of code that are meant to use its final result.
Improvement:	Students should be oriented to check each instruction within a loop and determine whether it depends on a partial result or a final result, moving the latter outside of the body.

JD.3

Code:	JD.3
Name:	Loop iterates the correct number of times, but over the incorrect range.
Description:	Improper initialization of the loop counter or improper construction of its stop condition, leading to an unexpected number of loop iterations.

Example:	Write a program that prints the sum of all numbers from 1 to 9:
	`1. int i;`
	`2. int sum = 0;`
	`3. for (i = 2; i <= 10; i++) {`
	`4.    sum = sum + i;`
	`5. }`
	`6. System.out.printf("The sum is %d\n", sum);`
	In this case, the `for` loop is executed the correct number of times (9), but the range values are wrong (2 to 10). The correct code would be:
	`3. for (i = 1; i < 10; i++).`

Rationale:	Students does not understand how to construct the loop structure (counter initialization/update or stop condition) that iterates over the desired range of values.
Consequences:	The loop will iterate the correct number of times, but over the wrong range of values, leading to unexpected program behavior.
Detection:	**Where:** o In the instancing or body of a loop structure
	How: o The loop structure has flaws on its specification (counter initialization/update or stop condition)
Improvement:	Students should be oriented to check the range over which the loop will iterate and initialize the loop accordingly – they might wish to print out the indexes once to see if the loop has been instanced appropriately.

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.

JD.7

We consider this misconception does not exist in Java because the programming language does not contain the iterator `range`.

JD.8

We consider this misconception does not exist in Java because the programming language does not contain the iterator `range`.

JE.1

<table>
<tr><td><b>Code:</b></td><td>JE.1</td></tr>
<tr><td><b>Name:</b></td><td>No fields specified in comparison of objects from a class.</td></tr>
<tr><td><b>Description:</b></td><td>Attempt to compare entire Objects using only their identifiers in a boolean expression, rather than comparing each of their respective fields.</td></tr>
<tr><td><b>Example:</b></td><td>

```java
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 == point2) {
      System.out.printf("The points are equal.\n");
   }
}
```

</td></tr>
<tr><td><b>Rationale:</b></td><td>Students consider the program automatically compare all fields of the objects, thus not being necessary to specify any of them on the sentence.</td></tr>
<tr><td><b>Consequences:</b></td><td>Although the program compiles, it will produce an unexpected output.</td></tr>
<tr><td rowspan="2"><b>Detection:</b></td><td><b>Where:</b>
o On any function that compares Objects from the same Class.</td></tr>
<tr><td><b>How:</b>
o Objects are compared directly through their identifiers; no fields are specified.</td></tr>
<tr><td><b>Improvement:</b></td><td>Students should be oriented to understand that, in Java, a comparison of Objects without specify the fields is a comparison of its references. To compare the values of fields, it is mandatory to specify them.</td></tr>
</table>

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:** ○ On the line where a field of some Object is compared against a identifier of another.
	How: ○ 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.3

We consider this misconception does not exist in Java because the programming language does not work directly with pointers. Therefore, the confusion between accessing pointers and accessing fields of a struct does not occur in Java.

JE.4

Although JAVA does not have the `struct` structure, this misconception could be considered similar to the JH.7. We decided to keep the JH.7 and not include the JE.4 as the Topic H relates to objects and its particularities.

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.

JF.1

We consider this misconception does not exist in Java because the programming language does not work directly with pointers. Therefore, the confusion between accessing the address of a variable or the value of this variable does not occur in Java.

JF.2

We consider this misconception does not exist in Java because the programming language does not work directly with pointers. Therefore, the confusion between applying an arithmetic operation in an address or in a value of a variable does not occur in Java.

JF.3

We consider this misconception does not exist in Java because the programming language does not work directly with pointers. Therefore, the problem to store memory addresses in pointer variables does not occur in Java.

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.

JF.5

We identified the misconception JF.5 is equivalent to the JA.2. Therefore, although there is this misconception in JAVA, it is only considered and described once.

JG.1

<table>
<tr><td><b>Code:</b></td><td>JG.1</td></tr>
<tr><td><b>Name:</b></td><td>Incorrect precedence for boolean operators.</td></tr>
<tr><td><b>Description:</b></td><td>Order of precedence of boolean expression is different than the one was intended.</td></tr>
<tr><td><b>Example:</b></td><td>

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))
```

</td></tr>
<tr><td><b>Rationale:</b></td><td>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.</td></tr>
<tr><td><b>Consequences:</b></td><td>The expression will not be evaluated correctly and the program will behave in an unexpected manner.</td></tr>
<tr><td rowspan="2"><b>Detection:</b></td><td><b>Where:</b>
o Whenever boolean expressions are used, in particular those involving more than one boolean operator.</td></tr>
<tr><td><b>How:</b>
o Students either fail to parenthesize the expression, or do so inappropriately.</td></tr>
</table>

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

<table>
<tr><td><b>Code:</b></td><td>JG.2</td></tr>
<tr><td><b>Name:</b></td><td>Nested if-statements where a single boolean expression could have been used.</td></tr>
<tr><td><b>Description:</b></td><td>A boolean expression is written as an <code>if-else</code> nested sequence.</td></tr>
<tr><td><b>Example:</b></td><td><pre>
public static void main(String[] args) {
   int result = 0;
   int a = 100;
   int b = 200;
   int c = 300;

   if (a > 100) {
       if (b < 400) {
           result = 1;
       } else {
           if (c == 300) {
               result = 1;
           }
       }
   }

   if (result == 1) {
       System.out.println("True sentence!\n");
   } else {
       System.out.println("False sentence!\n");
   } }
</pre>
The same code above should be written as:
<pre>
if ((a && (b || c)):
    # Do something
</pre></td></tr>
<tr><td><b>Rationale:</b></td><td>Students do not know how to evaluate multiple-variable boolean expressions, and so they build a sequence of <code>if</code> and <code>else</code> statements, each containing a single variable to be evaluated.</td></tr>
<tr><td><b>Consequences:</b></td><td>Although the program will still work correctly with a sequence of <code>if</code> and <code>else</code> statements (<i>i.e.</i> this is not an error), longer boolean expressions will lead to needlessly complex code, which is difficult to read, understand, and debug.</td></tr>
<tr><td rowspan="2"><b>Detection:</b></td><td><b>Where:</b>
o Whenever a boolean expression must be evaluated within the code.</td></tr>
<tr><td><b>How:</b>
o The boolean expression is evaluated through a sequence of <code>if</code> and <code>else</code> statements.</td></tr>
<tr><td><b>Improvement:</b></td><td>Students should be oriented to understand how boolean expressions are evaluated in Java language, and also how to use the proper operators (<code>or</code>, <code>and</code>, <code>not</code>, and parentheses), to build these expressions. In addition, the equivalence to nested if-statements could be explored.</td></tr>
</table>

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

<table>
<tr><td><b>Code:</b></td><td>JG.4</td></tr>
<tr><td><b>Name:</b></td><td>Attempt to evaluate a boolean expression through loop iterations.</td></tr>
<tr><td><b>Description:</b></td><td>A loop statement is used where a boolean expression would need to be evaluated only once.</td></tr>
<tr><td><b>Example:</b></td><td>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
(...)</td></tr>
<tr><td><b>Rationale:</b></td><td>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.</td></tr>
<tr><td><b>Consequences:</b></td><td>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.</td></tr>
<tr><td rowspan="2"><b>Detection:</b></td><td><b>Where:</b>
○ Whenever a boolean expression with multiple statements must be evaluated within the code.</td></tr>
<tr><td><b>How:</b>
○ A logic problem is solved through a sequence of loop iterations, rather than a boolean expression.</td></tr>
<tr><td><b>Improvement:</b></td><td>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.</td></tr>
</table>

JH.1

We consider this misconception does not exist in Java because the programming language does not have an append method to a list class. Thus, the misconception related to an attempt to assign a value to the identifier of a method does not occur.

JH.2

Code:	JH.2
Name:	Treating method that returns a new instance of the same object as one that changes the instance itself.
Description:	A method that operates on the caller instance returns a new instance with the result of the method execution. However, the returned instance is not considered/processed on the caller instance, leading the program to an unexpected behavior.

Example:	`1. String s1 = "Testing string in Java";`
	`2. s1.replace("string", "vector");`
	`3. System.out.println(s1);`
	
	In line 2, a new string containing "`Testing vector in Java`" is returned, but it is not assigned to a variable and the result is lost. String `s1` will retain the initial string that was assigned. Therefore, the string “`Testing string in Java`” will be printed when line 3 is executed.

Rationale:	Students consider that methods that affect the contents of an instance will automatically change that instance.
Consequences:	The returned instance is not considered/processed on the caller instance, leading the program to an unexpected behavior.
Detection:	**Where:** ○ Whenever a method is called upon to perform some operation on the contents of an instanced object. This is especially the case when `String` objects are in consideration, due to the immutability of strings in Java.
	How: ○ A method is called to generate a result based on the current value, but the result is not assigned to a variable.
Improvement:	Students should be oriented to understand the implementation of the specific classes they want to use. They should learn to check the documentation to understand the return values of methods.

JH.3

We consider this misconception does not exist in Java because the programming language does not have a `def` keyword. Therefore, the misconception related to a function call using this keyword does not occur.

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.5

We consider this misconception does not exist in Java because the programming language does not have an append method to a list class. Thus, the misconception related to an attempt to assign a value to the identifier of a method does not occur.

JH.6

We consider this misconception does not exist in Java because the programming language does not have a `self` keyword. Therefore, the misconception related to a variable reference using this keyword does not occur.

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.8

Code:	JH.8
Name:	Attempt to change the value of a `final` variable.
Description:	A `final` variable is declared and initialized with some value, but at some point of the code its value is changed.

Example:	`public static int addTwo(int a) {`
	`    return a + 2;`
	`}`
	
	`public static void main(String[] args) {`
	`    final int x = 1;`
	`    int y = 2;`
	`    System.out.printf("Initial x = %d\n", x);`
	`    x = addTwo(y);`
	`    System.out.printf("Final x = %d\n", x);`
	`}`

Rationale:	Students consider that a `final` variable could have its value updated.
Consequences:	A compilation error will be produced, followed by the message “`The final local variable cannot be assigned`”.
Detection:	**Where:** o Whenever is attributed a new value to a final variable.
	How: o A new value is directly (*e.g.* `x = 10`) or indirectly (*e.g.* `x = addTwo(y)`) attributed to a final variable.
Improvement:	Students should be oriented to understand how `final` variables work and also the concept (and the differences from) of `immutable objects` in Java.