Recursion in Actual Java Programming

Alexander Prutzkow

***Abstract*—Recursion has been used in programming since the 1960s. The purpose of the study is to identify the place of recursion in actual Java programming for the preparation of a university course. The materials were 19 books on Java describing recursion, as well as articles on mental models of recursion. We have the following results of the study. Recursion is, first of all, a technique and a tool. Recursion is described in books together with the concepts of stack overflow, tail recursion, tail call optimization (TCO), and finite or infinite. TCO can be converted into iterations by techniques or integrated development environment (IDE). The book authors prefer calculation of factorial and Fibonacci numbers, binary search, MergeSort, QuickSort, Towers of Hanoi as examples of recursion. Most authors devote a chapter to recursion. Programmers prefer iteration in many problems. In rare cases (such as tree traversal), most programmers (over 51%) use recursion. The most viable mental model is the Copies Model. There are rules for designing recursive methods. When teaching recursion, it is necessary to carefully select assignments. We have not found an exact answer to the question of when to use recursion. Recursion is used when solving well-defined problems in recursive fashion. Recursion has a strong place in Java programming. We will not include recursion in the course of introductory programming, but we will include it in the course of algorithms and data structures. Recursion in actual Java programming is not necessary, merely useful.**

***Keywords*—recursion, Java, programming, teaching, tail call optimization.**

To iterate is human, to recurse divine.
— Laurence Peter Deutsch (from [1])

In order to understand recursion
one must first understand recursion [2, p. 500]

I. Introduction

Recursion (in programming) is the active flow of control to a new invocation/copy of the subroutine called and the passive flow of control back from terminated ones ([3] adapted from [4]). A subroutine is recursive (uses recursion) if it contains a call to itself (directly or transitively) in its body.

The first high-level programming language Fortran had no recursive subroutines [5]. In 1958, J. McCarthy programmed differentiation of algebraic equations in Fortran. He wanted to use recursion, but it could not be introduced into the language. For this reason, J. McCarthy created the first language with recursion – Lisp [6]. The first imperative programming language with recursion Algol 60 appeared in 1960. The phrase about the possibility of recursion in the report on the language was introduced by the editor P. Naur at the suggestion of A. van Wijngaarden and E.W. Dijkstra [7]. The origin of recursion as a mathematical concept is explored in [8].

Recursion is still relevant. However, the place of recursion in actual programming requires a study. In our study we will focus, first of all, into Java programming.

II. Motivation

When preparing for a course on introductory programming in Java, the question arose as to whether it was worthwhile to include a section on recursion. The results of searching for an answer to this question formed the text of this article.

III. The purpose of the study

The purpose of the study is to identify the place of recursion in actual Java programming.

The study involves finding answers to the following research questions:

RQ1: How is recursion stated in Java books?

RQ2: What mental models of recursion do programmers have?

RQ3: What aspects related to recursion emphasize in materials, used to answer to RQ1 and RQ2?

RQ4: When to use recursion in Java?

IV. Materials and methods

The materials are literature selected according to filters:

- indexed by books.google.com or scholar.google.com;
- dedicated to recursion;
- written in English;
- published in 2016–2024 because we explore the actual place of recursion.

Additional filters for materials [9–27] to answer RQ1 are:

- books and;
- dedicated to the Java programming language.

The book [28] presents recursion in the same way as [15].

The map of concepts and text analysis are the methods used to answer to RQ1. The study using the map of concepts is inspired by [29–30]. We have used the map to visualize the relationships between concepts in [31].

The materials [3, 32–37] for answering RQ2 were filtered by the topic of mental models. The method used in answering RQ2 is a systematic analysis of the text of the materials.

The materials for answers to RQ3 and RQ4 are the materials for answers to RQ1 and RQ2. The method is a systematic analysis of the text of the materials.

Manuscript received April 18, 2025.

A. Prutzkow is with Ryazan State Radio Engineering University, 390005, Gagarin str., 59/1, Ryazan, Russia, and with Lipetsk State Pedagogical University, 398020, Lenin str., 42, Lipetsk, Russia (e-mail: mail@prutzkow.com).

V. Results

A. RQ1. A more general concept for recursion

In the materials we found the phrase "recursion is", extracted from them, and aggregated a more general concept. The most common concept is technique (table 1).

Table 1. More general concepts for recursion and their frequencies

Recursion is	Count	Sources (could be duplicated)
technique	9	11, 13, 15, 17, 19, 20, 24, 25, 26
tool	5	10, 15, 20, 25, 27
approach	3	14, 16, 25
way	3	13, 19, 23
alternative	1	17
capability	1	21
concept	1	19
feature	1	22
fundamental	1	15
method	1	9

B. RQ1. Concepts accompanying recursion

We have systematically analyzed the text of the materials. The state of recursion in books on Java is accompanied by the following concepts (fig. 1). We omitted concepts that:

- obviously related to recursion (base case, general case, etc.);
- mentioned in only one book.

The map (fig. 1) is constructed using data that includes the concept, its frequency, and sources:

- Backtracking (5): [13, 14, 16, 21, 27];
- Finite or infinite (8): [10, 11, 12, 13, 14, 21, 26, 27];
- Head recursion (2): [9, 25];
- Indirect / mutual (4): [13, 20, 21, 27];
- Mathematical fundamentals (4): [10, 16, 20, 21];
- Mathematical induction (3): [14, 15, 17];
- Recursive thinking (5): [13, 14, 16, 19, 26];
- Stack overflow (14): [10, 11, 12, 13, 14, 16, 17, 19, 21, 22, 23, 24, 25, 26];
- Tail recursion (14): [9, 11, 14, 16, 17, 18, 19, 20, 22, 23, 24, 25, 26, 27];
- Tail recursion – Tail call optimization (11): [9, 11, 16, 18, 19, 22, 23, 24, 25, 26, 27];
- Tail call optimization – Scala / Kotlin optimization (5): [9, 11, 18, 24, 27].

Examples of recursion is depicted on a submap (fig. 2). Examples are tasks discussed in the book, but are not assignments for the reader. Data for the submap are:

- Binary search (8): [12, 14, 15, 16, 17, 19, 20, 26];
- Binary search tree (6): [14, 15, 16, 19, 20, 27];
- Exponentiation (5): [10, 12, 14, 16, 17];
- Factorial (15): [9, 12, 14, 15, 16, 17, 19, 20, 21, 22, 23, 24, 25, 26, 27];
- Fibonacci numbers (8): [10, 12, 13, 14, 16, 21, 22, 26];
- File system operation (2): [13, 26];
- Greatest common divisor (GCD) (5): [10, 14, 15, 17, 26];
- MergeSort (7): [13, 14, 15, 16, 19, 20, 26];
- Palindrome (4): [12, 13, 17, 26];
- Permutations (2): [13, 19];
- QuickSort (7): [13, 14, 16, 19, 20, 26, 27];
- Recursive graphics (5): [15, 20, 21, 26, 27];
- Towers of Hanoi (7): [13, 14, 15, 17, 20, 21, 26].

C. RQ1. The part of the book devoted to recursion

We analyzed what part of the materials is devoted to recursion (table 2). If several non-intersecting parts were given recursion, the largest one was selected.

Table 2. Part of the book devoted to recursion

Part	Sources
Entire book	27
Chapter	10, 11, 13, 14, 17, 19, 20, 21, 22, 23, 25, 26
Section	12, 15
Subsection	16, 18, 24
Non-titled part of subsection	9

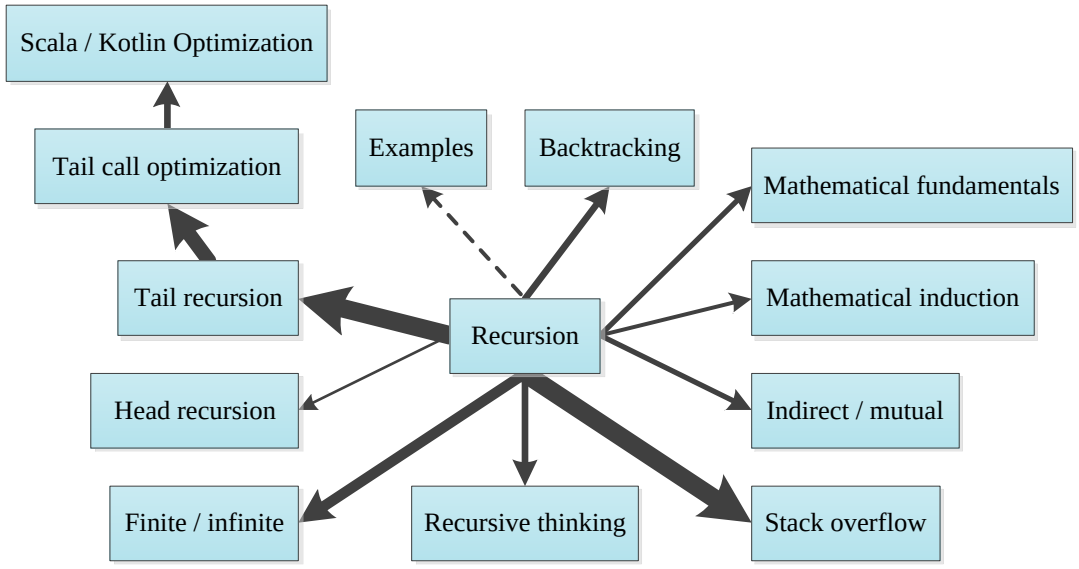

Fig. 1. Recursion and accompanying concepts. The thickness of the arrows is proportional to the frequency of the concepts.

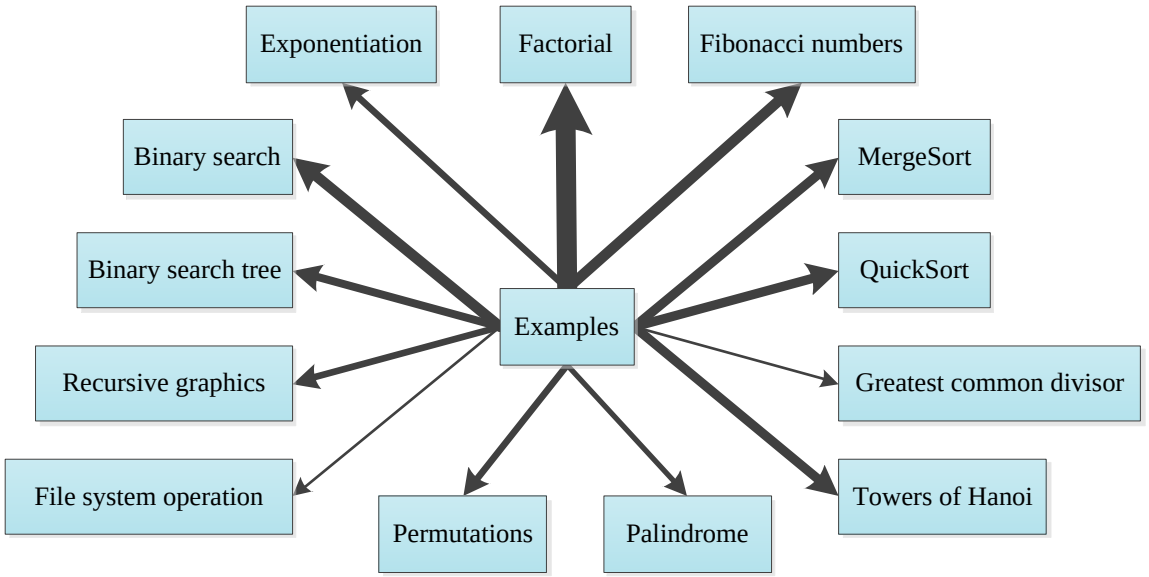

Fig. 2. Examples of recursion. The thickness of the arrows is proportional to the frequency of examples

D. RQ2. Mental models

Mental models of recursion are studied mainly in students, but not in programmers.

There are mental models of recursion that have students when learning programming [3]:

- The Copies Model is the viable model that reveals the active flow of control, and the switch to the passive flow once the base case is reached. The passive flow of control is made explicit.
- The Loop Model views recursion as a kind of iteration that halts once the base case is reached. It ignores both the active and passive flow of control.
- The Active Model reflects only the active flow of control without a passive flow. Students evaluate the solution at the base case. The model can be viable in some cases.
- The Step Model is nonviable, as the students lack understanding of recursion. Either the recursive condition, or the recursive condition and the base case is executed once.
- The Return Value Model mirrors that values are generated by each instantiation, which are then stored and combined to calculate a solution.
- The "Syntactic", "Magic" Model reveals that students have no idea of recursion and how it works. Nevertheless, they can match syntactic elements. The active flow, the base case, and the passive flow can be traced.
- The Algebraic Model describes students who treat the program as algebraic problem.
- The Odd Model encompasses different misunderstandings, which lead to the student not being able to predict the program's behavior.

In [32], the influence of data structures on the programmer's choice of iteration or recursion in the Python language was revealed. For processing arrays and lists, the programmers participating in the study preferred iteration. For processing trees, the programmers preferred recursion. When solving the problem of calculating odd nodes programmers used only recursion. When calculating the factorial, a classic example of the use of recursion (see fig. 2), only 31.3% of programmers used recursion. Similar results of calculating factorials, Fibonacci numbers and generating permutations in C were published in [33]. Recursion chose not more than 34% of participants. To find the deepest common ancestors in trees 19% of students choose to use iteration, 51% choose recursion, and 16% choose to combine both iteration and recursion [34]. Students who choose iteration performed more correct than those who choose recursion and the combination of both.

In [35], the speed of understanding tail and non-tail recursion in Python and the number of errors programmers make were measured. Tail recursion is easier comprehending, "when it is natural to use". An example of such a thing is reversing a list. In [36], the students' correctness of completing assignments in Java was determined, which had high, average, and low compatibility with recursion. Students performed better on high-compatibility tasks than on low-compatibility tasks at both the group level and the individual level.

In [37], students' perception of programs with iteration and recursion was investigated using an eye-tracking device. Students followed a comparable reading behavior for both iterative and recursive programs.

Studies that investigate mental models of recursion during student teaching minutely reviewed in [38].

E. RQ3. Recursion design rules

Designing recursion is drastically different from designing iterations. We identified two approaches to the design of recursion.

In [20], questions are proposed to verify that a recursive method works:

(1) The Base-Case Question: Is there a non-recursive way out of the algorithm, and does the algorithm work correctly for this base case?
(2) The Smaller-Caller Question: Does each recursive call to the algorithm involve a smaller case of the original problem, leading inescapably to the base case?
(3) The General-Case Question: Assuming the recursive call(s) to the smaller case(s) works correctly, does the algorithm work correctly for the general case?

The method works, if answers to all of these questions are yes.

Example (adapted from [20]). Let us apply the questions to the factorial method (listing 1).

Listing 1. The recursive factorial method

```
1  long factorial(int n) {
2      // Precondition: n >= 0
3      if (n == 0) {
4          return 1;
5      } else {
6          return (n * factorial(n - 1));
7      }
8  }
```

Answers to the questions and explanations are:

(1) Yes. The base case occurs when nis 0. The factorial method returns the value of 1. It's the correct value of 0! by definition.

(2) Yes. The parameter is nand the recursive call passes the argument $n - 1$. Therefore each subsequent recursive call passes a smaller value, until the value passed is finally 0, which is the base case.

(3) Yes. Assuming that the recursive call factorial($n - 1$) computes the correct value of $(n - 1)!$ and returns computed $n \times (n - 1)!$. This is the definition of a factorial.

Because the answers to all three questions are yes, we can conclude that the algorithm works. ■

The questions are similar to mathematical induction.

In [27], the rules for writing a working recursive subroutine are formulated:

(1) Handle the base cases first. The purpose of a base case isn't to avoid recursion altogether, but is to make a recursion finite.

(2) Only recur with a simpler case. The base cases are the "simplest" cases. A "simple" case is a clear way to close to the base cases.

(3) Don't use external variables. If a recursive method uses external variables, the programmer must understand and be forced to debug the manipulation of the variable by each recursive method call. It could be complicated or impossible.

(4) Don't look down. You have to trust recursion. If the recursive method is correct at this level, then you don't have to worry about any deeper levels.

It is pointed out that the rules are not absolute, and can be violated for good cause, if care is taken.

F. RQ3. Pedagogical aspect

There are four steps to teach recursion [35]:

(1) Teach the basic idea of a function that calls itself, fundamental concepts (base and general cases). Enrich it with the simplest examples possible, which are problems that are naturally solved using tail recursion.

(2) Emphasize the importance of the passive flow, and the return to additional processing after the recursive call. Use examples of calculating the factorial and the count appearances problem.

(3) Challenge the students with more complex forms of recursion, which require a more advanced mental model than iteration. Use an example is reversing a linked list.

(4) Tail recursion is important for functional programming, so transformations to tail form should also be taught. Emphasize transformation methods and, first of all, on the use of accumulating variables.

Teachers choose examples that based on their own knowledge of subject and practice, but less carefully consider learning perspectives and instructional effectiveness. "With poor instructional design [of assessment task], the cognitive outcomes will be unsatisfactory, and, worse, the affective outcomes, such as confusion and frustration, could result in irreversible damage to students' self-efficacy and academic interest" [36].

G. RQ4. When to use recursion in Java?

We have not found a clear answer to this question. The authors of the materials answer this question subjectively:

- "Many mathematical solutions are expressed more clearly using a recursive definition, and many data structures and algorithms can be written easier using recursion resulting in a less complicated program" [10].
- "Sometimes there is no obvious iterative solution at all [...] There is a certain elegance and economy of thought to recursive solutions that makes them more appealing" [13].
- "If it is easier to conceptualize an algorithm using recursion, then you should code it as a recursive method because the reduction in efficiency does not outweigh the advantage of readable code that is easy to debug" [14].
- "One advantage of using recursion is that often we can develop mathematical models that allow us to prove important facts about the behavior of recursive programs" [15].
- "If running times are equivalent and a recursive implementation is easier to understand than the equivalent iterative implementation, choose recursion over iteration; otherwise, iteration is generally preferred" [17].
- "A recursive approach is preferred over an iterative approach when the recursive approach more naturally mirrors the problem and results in a program that's easier to understand and debug [...] Another reason to choose a recursive approach is that an iterative one might not be apparent" [21].
- "In some cases, using recursion enables you to specify a clear, simple solution for an inherently recursive problem that would otherwise be difficult to obtain ... The decision whether to use recursion or iteration should be based on the nature of, and your understanding of, the problem you are trying to solve" [26].
- "The best answer to the question of when to use recursion is, simply, when you happen to find it useful" [27].

The most concrete answer to RQ4 that we've found is "a recursive definition [of the underlying problem or the data to be treated] may be a necessary condition for using recursive processing, but it is not a sufficient condition" [32].

VI. DISCUSSIONS

A. RQ1: How is recursion stated in Java books?

We start a discussion with our answers to the research questions.

Recursion is outlined, first of all, as a technique or a tool. Recursion is strongly related to the concepts of stack overflow, tail recursion (and its related concepts), finite or infinite, and backtracking. Recursion is demonstrated on examples factorial, Fibonacci numbers, binary search, MergeSort, QuickSort, Towers of Hanoi, and binary search tree. Authors mostly devote a chapter to recursion in their books.

The new breath for recursion was the tail call optimization (TCO). TCO consists in accumulating the result in the method parameter. As a result, the execution of the recursive method is accelerated. Compare the method for calculating the factorial after TCO (listing 2 [24]) with the original method (listing 1).

Listing 2. The recursive factorial method after tail call optimization

```
1  long factorialTCO(long acc, long n) {
2      return n == 1 ? acc : factorial TCO (acc * n, n-1);
3  }
```

Stack overflow when calculating factorial can be eliminated by introducing a functional interface and using the BigInteger class [23].

TCO can be converted into iterations [39]. The IntelliJ IDEA integrated development environment (IDE) has function such transformations [11].

Indeed, a standard recursive method can be translated into one that uses tail recursion (and by transitivity, iteration) through continuation-passing style [16].

B. RQ2: What mental models of recursion do programmers have?

Mental models are formed on the basis of already known ones. Classified mental models have a well-understood basis. The most viable model is the Copies Model.

Programmers and students avoid recursion. They use recursion more often in naturally recursive cases.

C. RQ3: What aspects related to recursion emphasize in materials, used to answer to RQ1 and RQ2?

We have identified two aspects: recursion design rules and pedagogical.

It is difficult for a student new to recursion to begin designing a recursive method. Recursion design rules prevent the student from making wrong steps and are a sign of the correctness of the designed recursive method.

The highlighted pedagogical aspects will facilitate the start in teaching recursion. The first aspect is an instruction on teaching recursion. The second aspect focuses the teacher's attention on the careful selection of recursion examples.

D. RQ4: When to use recursion in Java?

The lack of a clear answer to this question prevents us from teaching recursion to students in a simple and clear manner, and from developing instructions for programmers on how to use recursion. This is also the reason why programmers and students avoid using recursion.

In contrast to the answer to RQ4 from [32], in [40], philosophy is stated: "Implementation, maintenance, and modification generally will be minimized when each piece of the system corresponds to exactly one small, well-defined piece of the problem, and each relationship between a system's pieces corresponds only to a relationship between pieces of the problem". That's why recursive method must map a well-defined (in a recursive fashion) problem.

E. Further discussions

In [41], it's proposed tenets of redesign of an university programming unit. One of them is "'mathematical' programming should be avoided" and, in imperative applications, the data processing model predominates. In the books reviewed, recursion is presented using mathematical examples (factorial, Fibonacci numbers), which have simpler iterative implementations. At the same time, naturally recursive examples (file system operations, QuickSort, tree traversal) attract less attention. However, exactly such examples demonstrate when recursion is worth using. This position confirms study [42] that demonstrated the comprehension of recursion can be enhanced by focusing on recursive data structures and having the students use them in practical applications.

F. Iterative recursion

Recursion can be replaced by iteration using a stack as data structure (listing 3, adapted from [11, p. 150]). The initial data for the recursive call is pushed to the stack (lines 2–3). In the loop (lines 5–11), the data for the call is popped from the stack (line 6), checked to see if it needs to be processed (line 7), processed (line 8), and pushed to the stack (line 9). The raw data for the call is returned as the result (line 12).

Listing 3. Iterative recursion

```
1   Deque<T> stack = new ArrayDeque<>()
2   RecursiveCallData callData = getFirstCallData();
3   stack.addFirst(callData);
4
5   while (!stack.isEmpty()) {
6       callData = stack.removeFirst();
7       if(needsToProcess(callData)) {
8           RecursiveCallData processedCallData
                = processCallData(callData);
9           stack.addFirst(processedCallData);
10      }
11  }
12  return callData;
```

This listing can be considered as an indication of the need to use recursion: if it is simpler (clearer, more productive) to use recursion, and not this listing, then it is necessary to use recursion, otherwise it is necessary to use a such or adapted listing.

VII. CONCLUSION

Recursion has a strong place in Java programming. We will not include recursion in the course on introductory programming, but we will include it in the course on algorithms and data structures.

We explored not only the practical implications of recursion but also the mental models associated with it. Including work on mental models of recursion expanded the material for answering the research questions.

The resulting map of concepts will be useful to programming teachers when developing a lecture on recursion.

We could consider recursion as a command (not data) structure based on the stack of the Java virtual machine.

A supplement to the books on Java and recursion is [43],

which is mainly devoted to examples of recursion. The book on recursion [1] is worth noting. Python is used for examples. The relationship between mathematics and recursion is explored in [44]. The QuickSort algorithm is proven in it as well.

Recursion in actual Java programming "is never absolutely necessary, merely useful" [27].

REFERENCES

[1] Rubio-Sánchez M. Introduction to Recursive Programming. CRC Press, 2018.
[2] Evans B. et al. The Well-Grounded Java Developer, 2nd ed. Manning, 2022.
[3] Kiesler N. Mental Models of Recursion: A Secondary Analysis of Novice Learners' Steps and Errors in Java Exercises // 33rd Workshop of PPIG, 2022:226–240.
[4] George C.E. EROSI – Visualising Recursion and Discovering New Errors // SIGCSE Bulletin, 2000:305–309.
[5] Shasha D., Lazere C. Out of Their Minds. The Lives and Discoveries of 15 Great Computer Scientists. Copernicus, 1995.
[6] Mitchell J. Concepts in Programming Languages. Cambridge University Press, 2002.
[7] van den Hove G. On the Origin of Recursive Procedures // Computer Journal, 2015, 58(11):2892-2899. DOI: 10.1093/comjnl/bxu145.
[8] Soare R. Computability and Recursion // Bulletin of Symbolic Logic, 1996, 2(3):284-321.
[9] Sachdeva D., Ustukpayeva N. Mastering Java: A Beginner's Guide. CRC Press, 2022.
[10] Streib J., Soma T. Guide to Java. A Concise Introduction to Programming, 2nd ed. Springer, 2023. DOI: 10.1007/978-3-031-22842-1.
[11] Valeev T. 100 Java Mistakes and How to Avoid Them. Manning, 2024.
[12] Downey A., Mayfield C. Think Java. How to Think Like a Computer Scientist. O'Reilly, 2016.
[13] Horstmann C. Big Java Late Objects, 2nd ed. Wiley, 2017.
[14] Koffman E., Wolfgang P. Data Structures. Abstraction and Design Using Java, 4th ed. Wiley, 2021.
[15] Sedgewick R., Wayne K. Introduction to Programming in Java, 2nd ed. Addison-Wesley, 2017.
[16] Crotts J. Learning Java. A Test-Driven Approach. Springer, 2024. DOI: 10.1007/978-3-031-66638-4.
[17] Anderson J., Franceschi H. Java Illuminated. An Active Learning Approach, 5th ed. Jones & Bartlett Learning, 2019.
[18] Lelek T., Skeet J. Software Mistakes and Tradeoffs. How to Make Good Programming Decisions. Manning, 2022.
[19] Mongan J. et al. Programming Interviews Exposed, 4th ed. Wrox, 2018.
[20] Dale N. et al. Object-Oriented Data Structures using Java, 4th ed. Jones & Bartlett Learning, 2018.
[21] Deitel P., Deitel H. Java 9 for Programmers, 4th ed. Pearson, 2018.
[22] Saumont P.-Y. Functional Programming in Java. How Functional Techniques Improve Your Java Programs. Manning, 2017.
[23] Subramaniam V. Functional Programming in Java. Harness the Power of Streams and Lambda Expressions, 2nd ed. The Pragmatic Programmers, 2023.
[24] Urma R.-G. et al. Modern Java in Action. Lambdas, Streams, Functional and Reactive Programming. Manning, 2019.
[25] Weidig B. A Functional Approach to Java. Augmenting Object-Oriented Code with Functional Principles. O'Reilly, 2023.
[26] Liang Y. Introduction to Java Programming and Data Structures. Comprehensive Version, 12th ed. Pearson, 2019.
[27] Matuszek D. Quick Recursion. CRC Press, 2023.
[28] Sedgewick R., Wayne K. Computer Science. An Interdisciplinary Approach. Addison-Wesley, 2017.
[29] Sanders K. et al. Student Understanding of Object-Oriented Programming as Expressed in Concept Maps // SIGCSE, 2008:332–336.
[30] Pashukova A.D. Pedagogicheskaja Tekhnologija v Sovremennom Obrazovatel'nom Prostranstve Podgotovki Penitentsiarnykh Psikhologov [Pedagogical Technology in the Actual Educational Space of Teaching Penitentiary Psychologists] // Vectors of Psychological and Pedagogical Research, 2024, 2(3):74–81. [in Rus].
[31] Prutzkow A. Class Functionality and its Related Concepts: Research and Practice // International Journal of Open Information Technologies, 2024, 12(11):63-71.
[32] Baron A., Feitelson D. How a Data Structure's Linearity Affects Programming and Code Comprehension: The Case of Recursion vs. Iteration // 34th Workshop of PPIG, 2023:44–59.
[33] Sulov V. Iteration vs Recursion in Introduction to Programming Classes // Cybernetics and Information Technologies, 2016, 16(4):63–72. DOI: 10.1515/cait-2016-0068.
[34] Esteero R. et al. Recursion or Iteration: Does it Matter What Students Choose? // SIGCSE, 2018:1011–1016. DOI: 10.1145/3159450.3159455.
[35] Baron A., Feitelson D. Why Is Recursion Hard to Comprehend? An Experiment with Experienced Programmers in Python // ITiCSE, 2024. DOI: 10.1145/3649217.3653636.
[36] Chao J. et al. Dynamic Mental Model Construction: A Knowledge in Pieces-Based Explanation for Computing Students' Erratic Performance on Recursion, Journal of the Learning Sciences, 2018, 27(3):431–473. DOI: 10.1080/10508406.2017.1392309.
[37] Aqeel A. Understanding Comprehension of Iterative and Recursive Programs with Remote Eye Tracking // 32nd Workshop of PPIG, 2022.
[38] Mackay S. What Does Literature Tell Us About Recursion? // SIGCSE, 2022, 2:1173. DOI: 10.1145/3478432.3499210.
[39] Abelson H., Sussman G. Structure and Interpretation of Computer Programs, 2nd ed. MIT Press, 1996.
[40] Yourdon E., Constantine L. Structured Design. Fundamentals of a Discipline of Computer Program and Systems Design, 2nd ed. Yourdon Press, 1978.
[41] Jones M. The Redesign of the Delivery of an Introductory Programming Unit // Innovation in Teaching and Learning in Information and Computer Sciences, 2007, 6(4):169–182. DOI: 10.11120/ital.2007.06040169.
[42] Smith D. et al. Dealing with the Challenges of Learning Recursive Programming – Helpful Functions and Incremental Approaches to Encourage Recursive Thinking // Issues in Information Systems, 2024, 25(3):357–370. DOI: 10.48009/3_iis_2024_127.
[43] Campesato O. Data Structures in Java. Mercury Learning and Information, 2023.
[44] Liben-Nowell D. Connecting Discrete Mathematics and Computer Science, 2nd ed. Cambridge University Press, 2022.