Expressions in Java: Essential, Prevalent, Neglected?

Luca Chiodini
luca.chiodini@usi.ch
Software Institute, Università della Svizzera italiana
Lugano, Switzerland

Igor Moreno Santos
igor.moreno.santos@usi.ch
Software Institute, Università della Svizzera italiana
Lugano, Switzerland

Matthias Hauswirth
matthias.hauswirth@usi.ch
Software Institute, Università della Svizzera italiana
Lugano, Switzerland

Abstract

expressions are the building blocks of formal languages such as lambda calculus as well as of programming languages that are closely modeled after it. although expressions are also an important part of programs in languages like java, that are not primarily functional, teaching practices typically don't focus as much on expressions.

we conduct both a theoretical analysis of the java language, as well as an empirical analysis of the use of expressions in java programs by novices, to understand the role expressions play in writing programs. we then proceed by systematically analyzing teaching materials for java to characterize how they present expressions.

our findings show that expressions are an essential construct in java, that they are prevalent in student code, but that current textbooks do not introduce expressions as the central, general, and compositional concept they are.

CCS Concepts: • **Software and its engineering** → *Formal language definitions*; *General programming languages*; • **Social and professional topics** → **Computing education**.

Keywords: expressions, Java, textbooks, Blackbox, trees, education, Abstract Syntax Tree, grammar

ACM Reference Format:
Luca Chiodini, Igor Moreno Santos, and Matthias Hauswirth. 2022. Expressions in Java: Essential, Prevalent, Neglected?. In *Proceedings of the 2022 ACM SIGPLAN International SPLASH-E Symposium (SPLASH-E '22), December 05, 2022, Auckland, New Zealand*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3563767.3568131

SPLASH-E '22, December 05, 2022, Auckland, New Zealand

ACM ISBN 978-1-4503-9900-5/22/12…$15.00
https://doi.org/10.1145/3563767.3568131

1 Introduction

In programming languages that are considered predominantly functional, expressions[1] are the main building blocks of programs. In programming languages like Java, which are not predominantly functional, expressions seem to play a less important role, and this is often also reflected in teaching.

Indeed, in 2008 a study carried out a Delphi process among experts to identify important and difficult concepts in introductory programming [12]. Expressions only appear among topics as "construct/evaluate boolean expressions" and "writing expressions for conditionals", both ranked as very important but moderately difficult. There is no mention of the concept of expressions being treated in a general form, instead of the narrow view of logic and arithmetic.

Expressions are syntactic phrases that are constructed compositionally. They all evaluate to values, and in statically typed languages they all have a type. Because they are built compositionally, we can understand a bigger expression by decomposing it into its smaller components[2]. We can reason about its type and its value by reasoning about the types and values of its subexpressions. This recursive view of expressions is a prime example of decomposition. It allows students to learn to evaluate or type expressions in a general and systematic way [18].

In this paper, we set out to investigate the role of expression in Java. Specifically, we study whether:

- Expressions are **essential** in the Java programming language. We analyze all possible expressions in Java, characterizing their structure, and what's left of Java in the absence of expressions.
- Expressions are **prevalent** in Java code written by students. We do so by mining the Blackbox dataset, the largest repository of Java code written by students.
- Expressions are **neglected** in Java programming textbooks. We do so with a systematic analysis of the contents of current textbooks.

[1] Authors sometimes use the words *term* and *expression* interchangeably. Other times, they use the word term to refer to expressions that produce values and the word expression in a more general sense [21], standing also for phrases in other syntactic categories, including type expressions and kind expressions. Here we use expression to refer to syntactic phrases that produce values.
[2] In impure languages like Java, side-effects complicate this reasoning.

Section 2 defines which Java constructs we consider as expressions. Section 3 analyzes the grammar of the Java programming language to show that the subset of Java involving expressions is considerably larger than what one is typically led to believe. Section 4 presents an empirical analysis of the use of expressions by novices. Section 5 describes a systematic analysis of textbooks that teach Java to show how these constructs are presented. Section 6 and Section 7 point out limitations and related work, respectively. Finally, Section 8 concludes and suggests future directions.

2 Expressions in Java

The Java Language Specification[3] (JLS) [13] contains both a formal specification of the language's concrete syntax, as well as an informal specification of the language semantics. The language specification categorizes expressions into the following six syntactic forms: (i) expression names, (ii) primary expressions, (iii) unary operator expressions, (iv) binary operator expressions, (v) ternary operator expressions, and (vi) lambda expressions. This categorization is too coarse-grained for the purpose of our study. For the analyses in Sections 4 and 5 we need to define exactly which *constructs* of the language we are considering as expressions, and we need to define their structure.

2.1 Grammar is Not Enough to Identify Constructs

The grammar productions that determine the concrete syntax of the language are not good candidates to be used as language constructs. A language construct may have multiple syntactic representations and thus it may correspond to multiple grammar productions (e.g., array instances may be created with or without array initializers, which affects whether or not they contain subexpressions denoting the array dimensions). A grammar production may also correspond to multiple language constructs when more contextual information is needed to determine the exact construct. For example, a simple name can be a local variable access or a field access depending on the context in which it occurs. A grammar production may even correspond to only part of a language construct, which allows for the reuse of a grammar production in the definition of different language constructs. Moreover, grammar productions are sometimes built with the purpose of enforcing associativity and precedence rules. In essence, the concrete syntax of a language is not the right level of abstraction to define its constructs. The level of abstraction that we are looking for is captured by the abstract syntax of a language.

The abstract syntax of Java is not defined in the language specification, so we will consider the one defined by Eclipse's Java Development Tools (JDT) [24]. Although JDT is closely modeled after the language specification, it diverges a little from it, mostly for practical implementation reasons. For example, it represents deeply nested expressions of the form `L op R op R2 op R3`, where the same binary operator appears between all the operands, with one AST (Abstract Syntax Tree) node holding all the operands. The language constructs that we will consider to be expressions mostly correspond to JDT's AST nodes that are subtype of `Expression`, with small modifications whenever we found aspects that diverge from the language specification.

2.2 Java Expression Constructs

For the analyses in Sections 4 and 5 we need an exhaustive and unambiguous definition of expression constructs in Java. We provide this in Table 1. Each row names a language construct, refers to the main JLS section where it is discussed, and specifies its structure. We represent the structure of a construct with a grammar.

EBNF Symbols

The bold symbols follow the conventions of EBNF:

- **[** a **]** denotes that a is an optional part of the construct;
- **{** a **}** denotes the absence or presence of one or more occurrences of a in the construct;
- a **|** b denotes the presence of either a or b in the construct (grouped where needed with **(** ... **)**).

Java Tokens

The `colored tokens` are used to denote tokens of the Java language.

Subexpressions

The meta-variable e denotes a subexpression. Some constructs restrict one of their subexpressions to only *variables* (JLS 15.26), represented as e_{var}, which according to the specification can be "named variables" (e.g., local variables) or "computed variables" (e.g., field accesses and array accesses). In terms of the constructs defined in Table 1, these are:

- `Id` - Simple Variable Access;
- **[** **(** e **|** $\mathtt{T}_r$ **).]** `Id` - Field Access;
- **[** $\mathtt{T}_r$ **.]** `super` `.Id` - Super Field Access;
- e`[`e`]` - Array Access.

e_{var} is used in the left-hand side of an Assignment, as an operand of a Postfix Expression, and as an operand of some Prefix Expressions (the Prefix Increment / Decrement Expression (JLS 15.15.[1-2])).

Auxiliary Productions

The remaining `MetaVariables` (described in Table 2) are auxiliary grammar productions, mostly corresponding to productions in the JLS grammar with some simplifications.

Some constructs in this list compound various parts of the language as described in the JLS. In particular, Simple Variable Access may be an access to a local variable or a parameter. A Field Access may be an access to an instance variable, a class variable, or an enum constant. Another example is Class Instance Creation, which may be the creation

[3] In this paper, we are considering Java 11 (a Long-Term Support version), excluding modules and annotations.

of a class instance, an anonymous class instance, or even a qualified class instance.

Notice that array initializers (JLS 10.6) are not expressions. Even though they are used to instantiate arrays (in a field or local variable declaration, or as part of an Array Instance Creation expression), they cannot by themselves be evaluated to produce a reference to an array instance. Thus, they cannot be used wherever a value of an array type is expected. We also do not consider parenthesized expression (JLS 15.8.5) as a separate expression construct because they only affect the order of evaluation[4].

3 What's left of Java without Expressions?

One could assume that there is much more to Java than expressions, and that expressions are not exactly the most important construct of an imperative, object-oriented language like Java. Java is a large complex language that, like many languages, evolved by accumulating more constructs over the years. Thus, although there are many constructs that are expressions, perhaps there are many more constructs that are *not* expressions and expressions in Java are really not that important?

In this section, we answer that question by looking at what is left of Java if we remove all expressions. We perform this analysis based on the grammar of Java 9[5] using ANTLR [20]. In Java, it is possible to determine unambiguously if a program contains an expression (but not to determine which exact construct!) simply by looking at which productions of the grammar were used to construct its parse tree (concrete syntax tree)[6].

We identify a set of grammar productions P_e containing the productions introduced in Chapter 15 (Expressions) of the language specification with the addition of *ExpressionName*, as described in Section 15.2 (Forms of Expressions) of the specification. The set P_e has the important property that a program contains an expression iff its parse tree (concrete syntax tree) contains a node corresponding to one of the productions in P_e.

We then use the set P_e and its complement set P_e', which contains all the productions in the grammar except for the productions in P_e, to devise the program in Listing 1. This program is interesting because to construct its parse tree, one must use all productions in P_e' and none of the productions in P_e. Note that the constructs in this program are *not* all the constructs in the language that are not expressions: they are, instead, all the constructs that are not expressions and do not require expressions. A `while` loop, for example, is not included even though it is not an expression, because it requires an expression as its condition. So the program contains all the constructs that can be used to write programs without expressions. This illustrates how essential expressions are when writing programs in Java. At the extreme, this is all the Java one teaches if expressions are not taught!

That is not to say that constructs like class definitions and interfaces are not important. In fact, they are extremely important for programming-in-the-large [9]: defining interfaces between components as well as connecting and organizing smaller program components into larger programs, applications, or systems. But expressions are still required to carry out computational processes that use them to produce results.

```
package p;
import java.util.*;
import java.util.List;
import static java.lang.Math.log;
import static java.lang.Math.*;
public abstract class C<T> extends Object
        implements RandomAccess {
    private interface I extends Formattable {
        public int[][] a = {{}, {}};
        abstract <E extends Formattable>
            void m() throws Exception;
    }
    enum E { E1(); }
    static {}
    {}
    public List<? extends T>[][] b;
    public C(float a, String b) { this(); }
    public C() {
        l: ;
        for (int i;;) { break; }
        for (int i;;) { continue; }
    }
    private <A extends C<T> & Formattable>
            void n(C<T> this) {
        final List<int[]> v;
        try {} catch (Exception e) {} finally {}
        return;
    }
}
```

Listing 1. All that's left of Java without expressions: Source code that compiles and uses *all* the Java grammar productions coverable without expressions.

[4]Except for a corner case whereby -2147483648 and -9223372036854775808L are legal but -(2147483648) and -(9223372036854775808L) are illegal because those two decimal literals are allowed only as an operand of the unary minus operator.

[5]We use Java 9 here instead of Java 11 because it is the latest version for which ANTLR provides a grammar with productions closely mirroring those of the grammar described in the language specification.

[6]That may not be the case for other programming languages. The syntax of a construct may not be enough to determine its nature. That determination may require additional contextual information. The meaning of a name in Java, for example, may depend on the context of its occurrence. But those ambiguous names are captured by the production *AmbiguousName* and they can only happen as part of an expression name.

Table 1. Java expression constructs. The meta-variable *e* denotes subexpressions. The bold symbols follow the conventions of EBNF. The colored tokens denote tokens of the Java language. The remaining meta-variables are described in Table 2.

Group	Construct	Java Spec.	Structure
Class Instance Creation	Class Instance Creation	15.9	[*e*.]new [<T{,T}>]T_r([*e*{,*e*}])[Block]
This	This Expression	15.8.3	[T_r.]this
Variable	Simple Variable Access	6.5.6.1	Id
	Field Access	15.11	[(*e* | T_r).]Id
	Super Field Access	15.11.2	[T_r.]super.Id
Method Invocation	Method Invocation	15.12	[(*e* | T_r).][<T{,T}>]Id([*e*{,*e*}])
	Super Method Invocation	15.12	[T_r.]super.[<T{,T}>]Id([*e*{,*e*}])
Array	Array Access	15.10.3	*e*[*e*]
	Array Instance Creation	15.10.1	new T[<T{,T}>][*e*]{[*e*]}{[]} new T[<T{,T}>][]{[]}ArrayInit
Type Comparison and Cast	Type Comparison	15.20.2	*e* instanceof T
	Cast Expression	15.16	(T)*e*
Lambda	Lambda	15.27	Params -> (Block | *e*)
Method Reference	Constructor Reference	15.13	T_r::[<T{,T}>]new
	Method Reference	15.13	(*e* | T_r)::[<T{,T}>]Id
	Super Method Reference	15.13	[T_r.]super::[<T{,T}>]Id
Operator	Conditional Expression	15.25	*e* ? *e* : *e*
	Assignment	15.26	e_{var} AssignOp *e*
	Postfix Expression	15.14.2	e_{var} PostfixOp
	Prefix Expression	15.15.1	PrefixOp (*e* | e_{var})
	Infix Expression	15.18.2	*e* InfixOp *e*
Literal	Boolean Literal	15.8.1	true | false
	Character Literal	15.8.1	CharacterLiteral
	Null Literal	15.8.1	null
	Number Literal	15.8.1	IntegerLiteral | FloatingPointLiteral
	String Literal	15.8.1	StringLiteral
	Class Literal	15.8.2	(T | void).class

If we partition the constructs of the language into expressions, statements, and definitions, we see that these are not three independent parts of the language, but they complement each other:

- *expressions* "depend" on *definitions* because they rely on previously defined names and operations, unless they only use built-in constructs;
- *definitions* "depend" on *expressions* because they are of little use without other definitions or ultimately expressions that use them;
- *statements* "depend" on *expressions* because their use only makes sense in the presence of side-effecting computations, which in Java are mostly carried out by expressions.

Clearly, expressions are an essential part also of imperative languages like Java.

Table 2. Meaning of `MetaVariables` used in Table 1 with reference to relevant section(s) of the Java Language Specification.

MetaVariable	Meaning	Java Spec.
`T`	Any Type	4.1
T_r	Reference Type	4.3
`Block`	Code Block	4.2
`ArrayInit`	Array Initializer	10.6
`Id`	Identifier	3.8
`Params`	Lambda Parameters	15.27.1
`AssignOp`	Assignment Operator	15.26
`PostfixOp`	Postfix Operator	15.14
`PrefixOp`	Unary Operator (except cast)	15.15
`InfixOp`	Binary Operator (except `instanceof`)	15.[17-24]
`IntegerLiteral`	Integer Literal	3.10.1
`FloatingPointLiteral`	Floating-Point Literal	3.10.2
`CharacterLiteral`	Character Literal	3.10.4
`StringLiteral`	String Literal	3.10.5

4 How Much Do Students Use Expressions?

So far we studied how expressions *could* be used in Java, based on an analysis of the language. We have not yet explored how expressions *are* used in Java; in particular, how they are used by novices. Maybe expressions are usually small, simple, and not very prevalent in student code, so that explicitly introducing them as a central concept for novices is overkill? We formulated the following research questions to drive our investigation:

RQ1 Which fraction of code consists of expressions?
RQ2 How many expression constructs are typically used in a method?
RQ3 How large are expressions?
RQ4 How deeply nested are expressions?
RQ5 Which expression constructs are more used?
RQ6 Which expression constructs tend to be always used in a project?

4.1 Methodology

We now describe our methodology to answer the above questions.

4.1.1 Getting Student code. To get a corpus of student code to analyze, we requested access to the Blackbox dataset. The Blackbox dataset [5] is a large-scale repository of Java code written using the BlueJ IDE. BlueJ accompanies the popular "Objects first with Java" textbook [16]. Users, typically in their introductory programming courses, can opt-in to anonymously send their code that can be analyzed later by researchers.

4.1.2 Selection of Projects. As part of our analysis, we not only need to parse Java source files, but also to resolve name bindings: we want to know what each name refers to. We need therefore to analyze projects that successfully compile.

Compilations in BlueJ can happen at different granularities: a user might request (or, autocompilation might trigger) the compilation of a single file or a whole package. While we can determine the set of files being compiled at a given compilation event, this set does not include "dependencies" (e.g., the compilation might have been requested for class `C`, which contains a reference to the type `D`: `D` is part of the project and is needed to compile `C`, but is not included in the set of files for which the compilation was requested).

We adopt BlueJ's notion of a project. However, given their use in education, projects in BlueJ cannot be equated to traditional projects in Software Engineering: they are often long-lived, with several classes being added and deleted, and they might contain "logically different" projects. Nevertheless, we include in our dataset each project only once.

Taking into account all these aspects, we gather data as follows: out of all compilations events stored in the Blackbox dataset, we keep only the successful ones; in the specified time window, we consider only the most recent successful compilation event per each project; finally, we retrieve all the Java source files in that project at that compilation event. Only a residual fraction ($\approx$ 0.2%) of source files cannot be downloaded due to internal consistency errors. We selected 7 days scattered throughout the first part of 2022[7] and gathered 88 408 projects, for a total of 988 883 Java source files.

Even after these careful considerations, a minority ($\approx$ 9.4%) of source files cannot be compiled successfully. This can be due to a variety of reasons, of which the two main ones are the use of external dependencies (e.g., testing frameworks, libraries provided by textbooks and institutions) and the presence of non-compiling files that were in the project at the selected successful compilation event but were not involved in the compilation (e.g., a project contains a class `E` that has

[7]Exact days were January, February, March, April, May, July and August 1st. June has been excluded due to a Blackbox server outage.

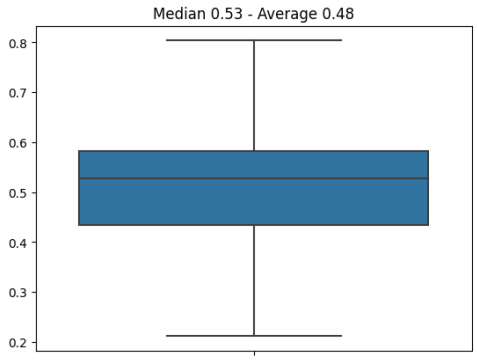

Figure 1. RQ1: Fraction of tokens dedicated to expressions per project (outliers not shown).

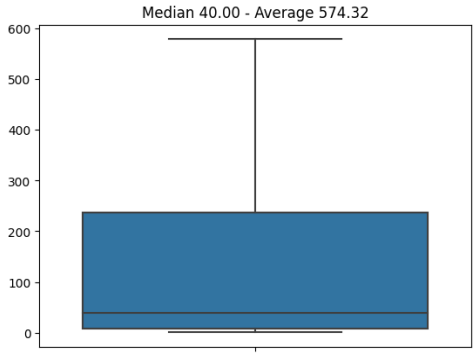

Figure 2. RQ2: Number of expression constructs per method (outliers not shown).

errors, but the compilation was requested for class `C` which did not require `E`).

4.1.3 Non-expressions inside expressions.

In a language like Java, it is not strictly true that all the descendants of an expression node in an Abstract Syntax Tree are themselves expressions. We identified three cases where this happens:

- class instance creation expressions containing an anonymous class declaration (i.e., `AnonymousClassDeclaration` as a child);
- lambda expressions containing a body (i.e., `Block` as a child);
- array initializers (see the discussion in Section 2 for the rationale).

We do *not* consider tokens belonging to these constructs as expressions, given that they belong to nodes that are not expressions. This also implies that further expressions potentially found inside those constructs are considered *distinct* expressions. As an example, the fragment of code `() -> { return 42; }` contains two distinct expressions (the lambda expression and the numeric literal); correspondingly, the four tokens `{` · `return` · `;` · `}` are not counted as part of any expression.

4.2 Prevalence of Expressions

The first two research questions explore how prevalent expressions are in Java code. We answer RQ1 by counting the fraction of tokens belonging to expression constructs over all the tokens in a project. Figure 1 shows that more than half (53%) of tokens in a median project are dedicated to expressions, a significant amount considering the imperative nature of the language.

Figure 2 provides a more absolute measure of the prevalence of expressions: the number of expression constructs per method (RQ2). The figure shows that a typical method contains 40 expression constructs. If we see each construct as a Lego-style building block, this number indicates that within a single method 40 pieces (quite a significant number!) are used and composed to express computation.

An important clarification about how we conservatively count expressions inside methods: the task is actually subtler than it seems at a first glance, because expressions in Java can also appear in other contexts. The language specification states that an expression can be declared and thus occur "in a field initializer, in a static initializer, in an instance initializer, in a constructor declaration, in an annotation, or in the code for a method".[8] We call the AST nodes corresponding to these places "declaring nodes". We say that the declaring node of an expression is the closest ancestor that is a declaring node. As a consequence, an expression e found in the declaration of a field f of a class C defined inside a method m, will have f as its declaring node, not C or m (thus, it will not be counted as "inside a method" for this statistic).

4.3 Size and Height of Expressions Trees

Expressions are recursive in nature: they can be arbitrarily nested, or, viewed conversely, increasingly larger expressions can be built by composing smaller elements. Do students take advantage of this power or do they tend to write entirely trivial, almost flat expressions? RQ3 and RQ4 help to investigate these aspects.

Figure 3 shows that, on average, expressions are made of more than 3 constructs. They are often not just the combination of two atomic expressions, as would be the case of a simple addition between two numbers (e.g., 1 + 2, which would have 3 constructs): they are, on average, slightly bigger than that. One could have expected this number to be larger. On the one hand, it can be argued that big expressions are difficult for a student to parse, and it is good practice to break them down into smaller sub-expressions. On the other hand, it is possible that novices have neither been exposed

[8] The Java Language Specification here is missing another context in which expressions can occur that was revealed by our analysis. Expressions can also occur as arguments in enum constant declarations. We intend to report this correction to the authors of the specification.

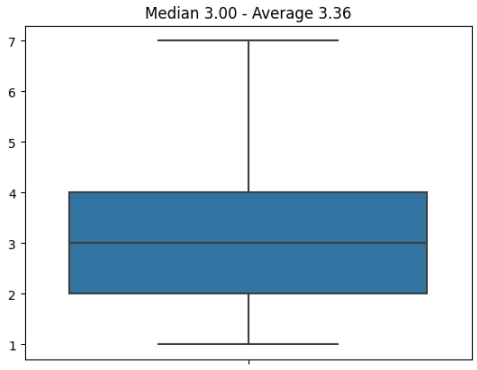

Figure 3. RQ3: Number of expression constructs per unique expression (outliers not shown).

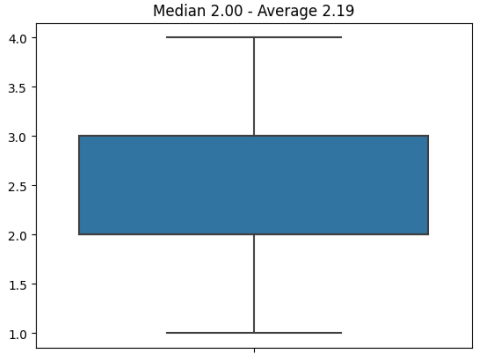

Figure 4. RQ4: Number of levels in expression trees (outliers not shown).

to nor explicitly taught how expressions are composed in general, and thus there is less composition of expressions in their own code.

We can also study the height of expressions trees. Trees of atomic expressions consist only of one level. Expressions consisting of a single binary operator and two atoms (e.g., 4 + 5) correspond to trees with 2 levels. On average, the trees of expressions written by students have more than 2 levels (2.19, as shown in Figure 4). This has a significant implication, because for trees with just two levels one could try to reason linearly about expressions, in a way that organizes them in a list (e.g., op `arg1` ... `argn`). While this is not a correct way to reason about expressions in general, it would "get the job done" in such cases. When expression trees have more than two levels, though, it becomes essential that students understand the recursive structure of expressions, and that they are able to reason about expressions as trees.

4.4 Distribution of Expression Constructs

Java has many different expression constructs, as shown in Table 1, but they are not all equally used. Although prevalence does not determine importance, it is nonetheless telling to see which expression constructs students are actually writing. This can suggest which parts of the language are more frequently covered by courses and textbooks, as well as which expression constructs appear to be essential as they occur in almost every project.

Table 3. Distribution of expression constructs.

Construct	%
Simple Variable Access	25.9
Field Access	14.4
Method Invocation	13.8
Number Literal	11.4
Infix Expression	11.2
String Literal	7.8
Assignment	6.1
Array Access	2.1
Class Instance Creation	2.0
Postfix Expression	1.7
This Expression	0.9
Boolean Literal	0.7
Character Literal	0.6
Prefix Expression	0.4
Null Literal	0.3
Cast Expression	0.3
Array Instance Creation	0.3
Super Method Invocation	0.0
Conditional Expression	0.0
Type Comparison	0.0
Class Literal	0.0
Lambda	0.0
Super Field Access	0.0
Method Reference	0.0
Constructor Reference	0.0
Super Method Reference	0.0

Table 3 shows the prevalence of each expression construct: the left column reports the construct name (as defined in Table 1), and right column indicates the percentage of nodes of that type over all the nodes that correspond to expression constructs.

We note that field and simple variable accesses (that include local variables and parameters) account for a significant fraction (40%) of all expression constructs. All literals combined make up 21% of expression constructs. Except for Field Access (which may or may not have an explicit target), the most used non-atomic expression construct is Method Invocation (14%).

These proportions, however, only tell part of the story. Certain expression constructs are consistently used in almost every project, despite occurring sparingly within a project. As a concrete example, it is difficult to imagine a meaningful Java project without any class instance creation, and it would be easy to gather consensus around the idea that it is

Table 4. Percentage of projects containing at least one occurrence of a given construct.

Construct	%
Field Access	93.4
Simple Variable Access	91.5
Infix Expression	88.2
Number Literal	87.4
Method Invocation	86.7
Assignment	86.1
String Literal	84.3
Class Instance Creation	72.3
Postfix Expression	54.4
Boolean Literal	36.4
Prefix Expression	33.7
Array Access	32.9
Cast Expression	30.4
Array Instance Creation	29.2
This Expression	25.3
Character Literal	24.9
Null Literal	20.0
Super Method Invocation	6.7
Conditional Expression	6.1
Type Comparison	2.9
Class Literal	1.7
Lambda	1.1
Super Field Access	0.4
Method Reference	0.2
Constructor Reference	0.2
Super Method Reference	0.0

important to teach it, but obviously that construct will not appear nearly as frequently as variable accesses do (indeed, class instance creations only account for 2% of expression constructs).

Table 4 tries to capture this phenomenon. For each expression construct, we report the fraction of projects that contain at least one occurrence of that construct.

Unsurprisingly, almost all projects contain at least one field access (≈ 93%) and one simple variable access (≈ 92%). Those are constructs used pervasively to read from variables (local variables, parameters, instance and static variables). Expressions with an infix binary operator (Infix Expression) and assignments follow in this ranking. The literal `null`, rarely needed in a well-designed program, appears at least once every five projects (20%). Arrays are used in roughly one third of the projects. The usage of string literals is widespread. Finally, functional-oriented features introduced in Java 8 (Lambda expressions and method / constructor references) are among the least used expression constructs.

Table 5. Programming textbooks included in the analysis.

Title	Edition	Ref.
Objects First with Java	6th	[16]
Java How to Program, Early Objects	11th	[8]
Java Programming	9th	[11]
Big Java	7th	[14]
Java - Software Solutions	9th	[17]
Absolute Java	6th	[22]

5 Do Textbooks Neglect Expressions?

We have seen in Sections 2 and 3 that expressions are indeed an essential construct of the Java language, and Section 4 showed that they are prevalent in the code of programming students. Our last question, then, is whether current Java programming textbooks present expression constructs and their children with descriptions and examples that indicate a lack of generality.

5.1 Methodology

We conducted a systematic study on Java textbooks to understand the extent to which expressions are covered.

5.1.1 Book Selection. We adopted the list of textbooks selected by [4] that include a variety of styles to teach CS1. Our exclusion criteria are books not using Java as the primary programming language (1 book excluded) and books whose last edition was published more than 10 years ago (6 books excluded). Table 5 shows the resulting list of 6 textbooks, all retrieved at their latest available edition.

5.1.2 Analysis. We set out to analyze how Java expression constructs are covered, reading the chapters in which they are introduced and scrutinizing the source code used in the related examples.

This effort aims to substantiate "anecdotal" observations noted in prior work that "many programming teachers and introductory textbooks do not emphasize expressions and evaluation, except when it comes to arithmetic and logic" [10].

We therefore considered each construct from Table 1, excluding arithmetic and logical operators, atomic constructs (such as literals) that cannot contain subexpressions (i.e., they are leaves in an Abstract Syntax Tree), and functional-oriented constructs introduced only in Java 8.

For each construct, we read the chapters in textbooks that aimed to present it. We selected such sections by looking at the table of contents and at the index. We tried to use our best judgement to reasonably augment these sections to include additional ones in close proximity that contained further examples using the construct. We acknowledge that this might lead to missing some examples, but on the other hand it is prohibitive to take into consideration all source code fragments in books often longer than a thousand pages.

After all, it is not our intention to blame a specific textbook or authors, but rather to derive an overall picture.

We scanned the English text in search of misleading sentences: problematic wordings that signal a loss of generality in the presentation of an expression construct and its children. Additionally, snippets of code showing example usages of a construct are checked to verify whether they include at least once a non-atomic expression for each child (so that a reader gets exposed to children not necessarily being atomic expressions such as literals or names).

5.2 Results

As argued earlier, a key characteristic of expressions is their compositionality. When the text or the examples (or both) only show narrow usages of the language, students are induced to believe that these constructs offer much less flexibility than what they do in reality.

Table 6 summarizes our findings. While most constructs are properly described, Class Instance Creation is presented in two books with problematic sentences. In general, subexpressions are much more neglected: 4 books do not properly describe the target of an array access expression, 3 books mispresent the operand of a cast expression, and again 4 books contain misleading statements about the target of a method invocation expression. Example snippets are even more problematic. We do not claim that textbooks should introduce those constructs exhibiting intricate code snippets. It makes perfect sense to introduce them with simple examples, but – as the "contrasting cases" [23] pedagogy suggests – one should also present examples using non-atomic expressions, to reinforce their generality.

It is instructive to read some of the sentences we classified as misleading. For example, in the array access expression, several textbooks mentioned that an "array name" should go in front of the square brackets, which excludes all other expressions that evaluate to an array reference. "The cast operator consists of the name of a type written in parentheses in front of a variable or an expression" [16] seems to imply that variables (i.e., uses of names) are not expressions but something different. "The new operator is used to create an object of a class and associate the object with a variable that names it" [22] seems to imply that class instance creation expressions necessarily need to be the right hand side of an assignment and cannot be used in a different way (e.g., as an argument of a method invocation expression). "When you use a nonstatic method, you use the object name, a dot, and the method name" [11], "to call a method [one uses] a variable that contains a reference to an object" [8] both imply that the target of a method invocation expression cannot be anything but a simple name.

These problematic sentences might very well instill misconceptions in students, such as the ones documented in [7]. For example, when a class instance creation is not taught as an expression, students start to believe that they cannot chain members to constructors (the misconception called CannotChainMemberToConstructor).

Being aware of how hard it is to write a programming textbook, we tried to be as lenient as possible while reading them, giving books' authors the benefit of the doubt. The description of what is allowed as the left-hand side of an assignment exemplifies this: we did not raise complaints about it being described as a "variable", in light of the fact that the language specification also uses this word. However, the specification also expands on this to clarify what is meant by "variable" (which is different than what most readers probably expect!): variables can also be "computed", meaning that their determination might involve the evaluation of an arbitrarily complex expression.

6 Limitations

We do not study the most recent version of Java, which would be Java 18 (March 2022) at the time of this writing. Our code analysis is limited to the Long Term Support (LTS) version 11 (September 2018), and the grammar analysis to version 9 (September 2017). However, the language features introduced since the LTS version 11 are barely covered in textbooks for introductory programming.

While we collected and analyzed almost 1 million Java source files, mitigating the bias by carefully spreading the selection of projects across time so as to represent a meaningful sample, this still constitutes only a fraction of the large Blackbox dataset.

We did not perform a comprehensive study of all Java textbooks. It is clearly possible that some books we did not analyze treat expressions more thoroughly. However, we based our selection on prior work [4], and picked those books that focus on Java and are current (i.e., we excluded books where their most recent edition was more than ten years old). The analysis was carried out by a single author, which constitutes a threat to validity.

This paper focuses exclusively on Java. Many other programming languages are used in education, one of the most prominent probably being Python. While we hypothesize that our results might generalize to Python and Python textbooks, this paper does not provide any evidence to back that up. Future work might want to replicate this study for Python or other similar languages.

The paper focuses on an imperative language. Textbooks for functional languages, especially those for purely functional languages like Haskell or Racket BSL, will have little risk of neglecting the treatment of expressions, given the absence of statements and mutation and the dominance of expressions.

7 Related Work

Our theoretical analysis of Java is based on the language specification. Another approach would be to use a well-known

Table 6. For each selected expression construct and its subexpressions, the number of textbooks containing misleading sentences about each expression and containing only atomic examples of each expression (type arguments omitted for clarity).

Construct	Structure	Misleading text	Only atomic examples
Array Access	e_{arr}`[`e_{idx}`]`	0	-
	e_{arr}	4	6
	e_{idx}	0	1
Array Instance Creation	`new T[`e_{size}`]{[`e_{size}`]}{[]}`	1	-
	e_{size}	0	2
Assignment	e_{var} `AssignOp` e	0	-
	e_{var}	1	0
	e	1	0
Cast Expression	`(T)`e	0	-
	e	3	4
Class Instance Creation	[e_{ref}`.`]`new T`$_r$`(`[e_{arg}{`,`e_{arg}`}`]`)`[`Block`]	2	-
	e_{ref}		(not discussed)
	e_{arg}	0	6
Conditional Expression	e_{cond} `?` e `:` e	0	-
	e_{cond}	0	0
	e	3	4
Type Comparison	e `instanceof T`	0	-
	e	0	6
Field Access	[`(`e`|T`$_r$`).`]`Id`	0	-
	e	2	6
Method Invocation	[`(`e_{ref}`|T`$_r$`).`]`Id(`[e_{arg}{`,`e_{arg}`}`]`)`	0	-
	e_{ref}	4	6
	e_{arg}	3	1

formally specified subset of Java, such as Featherweight Java [15] or Middleweight Java [2]. An advantage of using formally specified languages is that the constructs that are expressions in those languages are unambiguously specified. On the other hand, an approach based on either of those two sublanguages would inevitably cover only a subset of the language. There does exist a formalization of the full Java semantics using the $\mathbb{K}$ framework [3], however it only covers up to Java 1.4.

With a different focus, other repository-mining studies have been conducted to investigate how programmers use Java. Brown analyzed the Blackbox dataset to "learn how novices use different features of the Java language" [6]. Our work is different in that it focuses solely on expressions and thus dives deeper in this key part of the language. Mining open-source GitHub projects, Mazinanian et al. [19] studied the usage of lambda expressions introduced in Java 8.

Other studies reviewed introductory programming textbooks, for instance to analyze examples of object-oriented programming [4] and to compare and contrast pedagogies [1].

8 Conclusions

When teaching a language that is not predominantly functional, one might be tempted to downplay the role of expressions. In this paper, we have shown that expressions are an essential part of Java programs, even though Java is not a predominantly functional language. We also saw that expressions are heavily used in Java code written by students. Nevertheless, current Java programming textbooks at least partially neglect covering expressions in the appropriate generality and depth. We believe that future textbook revisions may want to put more emphasis on the beautiful compositional structure of expressions and the orthogonality between expressions, the contexts in which they may appear, their evaluation, and their types. Table 1 may provide a starting point for a discussion of the structure of Java expressions in future textbook revisions.

Given its potential wider interest for other studies, and for reproducibility purposes, we publish at https://doi.org/10.5281/zenodo.7225345 both a tool to verify the set of productions used to parse a Java program (used to create Listing 1) and the infrastructure built to extract projects from Blackbox, analyze Java source code to find expressions, and compute statistics (Section 4).

Acknowledgments

We are thankful to Neil Brown for the support in working with the Blackbox dataset.

This work was partially funded by the Swiss National Science Foundation project 200021_184689.

References

[1] Byron Weber Becker. 2002. Pedagogies for CS1: A Survey of Java Textbooks. *manuscript, http://www. math. uwaterloo. ca/~ bwbecker/papers/javaPedagogies. pdf, last visited* (2002), 03–01.

[2] G. M. Bierman, M. J. Parkinson, and A. M. Pitts. 2003. *MJ: An Imperative Core Calculus for Java and Java with Effects.* Technical Report UCAM-CL-TR-563. University of Cambridge, Computer Laboratory. https://doi.org/10.48456/tr-563

[3] Denis Bogdănaş. 2015. *A Complete Semantics for Java.* Ph. D. Dissertation. Alexandru Ioan Cuza University of Iaşi.

[4] Jürgen Börstler, Mark S. Hall, Marie Nordström, James H. Paterson, Kate Sanders, Carsten Schulte, and Lynda Thomas. 2010. An Evaluation of Object Oriented Example Programs in Introductory Programming Textbooks. *ACM SIGCSE Bulletin* 41, 4 (Jan. 2010), 126–143. https://doi.org/10.1145/1709424.1709458

[5] Neil Christopher Charles Brown, Michael Kölling, Davin McCall, and Ian Utting. 2014. Blackbox: A Large Scale Repository of Novice Programmers' Activity. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education (SIGCSE '14).* Association for Computing Machinery, New York, NY, USA, 223–228. https://doi.org/10.1145/2538862.2538924

[6] Neil C. C. Brown, Pierre Weill-Tessier, Maksymilian Sekula, Alexandra-Lucia Costache, and Michael Kölling. 2022. Novice Use of the Java Programming Language. *ACM Transactions on Computing Education* (July 2022). https://doi.org/10.1145/3551393

[7] Luca Chiodini, Igor Moreno Santos, Andrea Gallidabino, Anya Tafliovich, André L. Santos, and Matthias Hauswirth. 2021. A Curated Inventory of Programming Language Misconceptions. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1 (ITiCSE '21).* Association for Computing Machinery, New York, NY, USA, 380–386. https://doi.org/10.1145/3430665.3456343

[8] Paul J. Deitel and Harvey M. Deitel. 2018. *Java: How to Program Early Objects* (11th edition ed.). Pearson, New York.

[9] Frank DeRemer and Hans Kron. 1975. Programming-in-the Large versus Programming-in-the-Small. *ACM SIGPLAN Notices* 10, 6 (April 1975), 114–121. https://doi.org/10.1145/390016.808431

[10] Rodrigo Duran, Juha Sorva, and Otto Seppälä. 2021. Rules of Program Behavior. *ACM Transactions on Computing Education* 21, 4 (Nov. 2021), 33:1–33:37. https://doi.org/10.1145/3469128

[11] Joyce Farrell. 2019. *Java Programming* (nineth edition ed.). Cengage, Australia United States.

[12] Ken Goldman, Paul Gross, Cinda Heeren, Geoffrey Herman, Lisa Kaczmarczyk, Michael C. Loui, and Craig Zilles. 2008. Identifying Important and Difficult Concepts in Introductory Computing Courses Using a Delphi Process. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '08).* ACM, New York, NY, USA, 256–260. https://doi.org/10.1145/1352135.1352226

[13] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. 2000. *The Java Language Specification.* Addison-Wesley Professional.

[14] Cay S. Horstmann. 2019. *Big Java. Early Objects* (seventh edition ed.). John Wiley & Sons, Inc, Hoboken, NJ.

[15] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. 2001. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems* 23, 3 (May 2001), 396–450. https://doi.org/10.1145/503502.503505

[16] Michael Kölling and David Barnes. 2017. *Objects First With Java: A Practical Introduction Using BlueJ* (sixth ed.). Pearson.

[17] John Lewis and William Loftus. 2017. *Java Software Solutions: Foundations of Program Design* (ninth edition ed.). Pearson, NY, NY.

[18] Guillaume Marceau, Kathi Fisler, and Shriram Krishnamurthi. 2011. Do Values Grow on Trees?: Expression Integrity in Functional Programming. In *Proceedings of the Seventh International Workshop on Computing Education Research - ICER '11.* ACM Press, Providence, Rhode Island, USA, 39. https://doi.org/10.1145/2016911.2016921

[19] Davood Mazinanian, Ameya Ketkar, Nikolaos Tsantalis, and Danny Dig. 2017. Understanding the Use of Lambda Expressions in Java. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (Oct. 2017), 1–31. https://doi.org/10.1145/3133909

[20] T. J. Parr and R. W. Quong. 1995. ANTLR: A Predicated-LL(k) Parser Generator. *Software: Practice and Experience* 25, 7 (1995), 789–810. https://doi.org/10.1002/spe.4380250705

[21] Benjamin C. Pierce. 2002. *Types and Programming Languages.* MIT Press, Cambridge, Mass.

[22] Walter Savitch and Kenrick Mock. 2015. *Absolute Java* (6th edition ed.). Pearson, Boston.

[23] Daniel L. Schwartz, Catherine C. Chase, Marily A. Oppezzo, and Doris B. Chin. 2011. Practicing versus Inventing with Contrasting Cases: The Effects of Telling First on Learning and Transfer. *Journal of Educational Psychology* 103, 4 (2011), 759–775. https://doi.org/10.1037/a0025140

[24] JDT/Core Team. 2022. JDT Core Component | The Eclipse Foundation. https://www.eclipse.org/jdt/core/index.php.