JavaLite Language Reference Manual

Frances Cao (fc2679)
Mateo Maturana (mm5589)
Hongfei Chen (hc3222)
Ian Chen (yc3936)

February 24, 2021

Contents

1 Overview

While Java is one of the most popular programming languages for beginners, it could be challenging for new programmers to get familiar with its excessive syntax and strict object-oriented programming rules. Aiming to design a more beginner-friendly language, we propose JavaLite, a partially object-oriented language derived from Java. JavaLite has simplified syntax compared to Java and incorporates functional programming. Further, JavaLite supports more built-in functionalities for non-primitive data types, including String and Array. These built-in methods are inspired by Python and the goal is to provide a more intuitive way of String/Array manipulation.

This language reference manual describes the grammars for the language. The manual starts by presenting the lexical structure of JavaLite, which is followed by the section on types and variables. The syntactic structure of the language is also specified in the later sections, including the structures of statements and expressions. Section 5 describes the definition of classes and the functionalities supported by them. The built-in functions for string and array manipulation that are unique to the JavaLite language are defined in Section 6. Finally, the manual provides some style guide and sample code for reference.

2 Lexical Structure

2.1 Comments

JavaLite supports both single line and multi-line comments.

```
// This is a single line comment
```

A single line comment starts with "//" and ends at the end of the line. Any ASCII characters before the end of the line are considered comments.

```
/* This is a
muli-line
comment */
```

A multi-line comment is wrapped by "/*" and "*/". Namely, any ASCII characters after "/*" are considered comments until there is a "*/".

JavaLite does not allow nested comments. The comments will be ignored by the compiler and will not be executed.

2.2 Identifiers

An identifier in JavaLite is an unlimited-length character sequence consisting of letters and digits "a-zA-Z1-9", except for those reserved for keywords (2.3). Note that identifiers must begin with letters.

```
// valid identifiers
javalite
name
person12

// invalid identifiers
14
1java
```

2.3 Keywords

There are a total number of 13 keywrods reserved in JavaLite. Each keyword is a character sequence consisting of ASCII letters. The following are all the keywords.

boolean	else	this
char	for	void
class	if	while
constructor	int	string
double	return	

2.4 Separators

There are 9 separators in JavaLite. They are tokens formed from ASCII characters. The following are all the separators.

(	]	;
)	{	,
[	}	.

2.5 Operators

There are 17 operators in JavaLite. They are tokens formed from ASCII characters. The following are all the operators.

=	-	/
==	<	&&
+	<=	||
>	$\star$	!=
>=	%	!

2.6 Literals

Literals in JavaLite repesent any constant value of a primitive type (int, float, boolean, char) or the string type.

2.6.1 Int Literals

An integer literal is either the single ASCII digit 0, representing the integer zero, or any ASCII digit between 1 and 9, optionally followed by any sequence of ASCII digits between 0 and 9.

The int data type is a 32-bit signed primitive data type. The valid range of the integer literal is between $-2,147,483,648 (-2^{31})$ and $2,147,483,647 (2^{31}-1)$.

If the integer literal is larger than $2,147,483,647$ or less than the unary minus operator $2,147,483,648$, it will result in a compile-time error.

Examples of int literals:

```
0
-5640
42014
```

2.6.2 Float Literals

A floating point literal has a whole number, a decimal or hexadecimal point, a fraction, an exponent, and a type suffix.

In decimal float literals, at least one digit, and either a decimal, an exponent, or a float type suffix is required. The exponent is indicated by the letter e or E followed by an optional signed integer.

Floating point literals are always of type double.

Examples of floating point literals of double type:

```
.23
3.145
0.0
516431E-10
-1.3
```

2.6.3 Boolean Literals

The Boolean type is represented by the literals true and false.

2.6.4 Char Literals

A char literal is specified as a character or an escape sequence enclosed in single quotes. Char literals also represent UTF-16 code units.

Examples of char literals:

```
'a'
'\uFFFF\'
'\t'
'\n'
```

2.6.5 String Literals

A string literal is any sequence of zero or more characters enclosed in double quotes. Characters such as newlines may be represented by escape sequences.

Examples of string literals:

```
""   // the empty string
"\""
"Hello World!"
"This is a string literal\n that spans two lines"   // this forms a string with
    ↪ two lines of text
```

3 Types and Variables

JavaLite is a statically typed language. All variables and expressions will have a known type at compile time. JavaLite is also a strongly typed language, and limits the values that a variable can hold. Variables must be defined before use.

3.1 Primitive Types

JavaLite support four primitive data types:

- int (number, 4 bytes)
- double (float number, 8 bytes)
- char (character, 2 bytes)
- boolean (true or false, 1 byte)

3.1.1 Int Operations

JavaLite provides a number of operations for ints.

The first type of operators are the comparison operators, which result in a value of type boolean. These operators are $<, <=, >, >=, ==, !=$.

```
2 < 4;    // true
5 <= 4;   // false
0 > -10;  // true
3 >= 3;   // true
2 == 0;   // false
2 != 0;   // true
```

The second type of operators are the numerical operators, which result in a value of type int. These operators are $+, -, *, /, \%$. Note that / returns values of type int, not double, so the result will be the floor of the division.

```
10 + 2;   // 12
-8 - 10;  // -18
2 * 3;    // 6
4 / 5;    // 0
17 % 6;   // 5
```

3.1.2 Double Operations

JavaLite provides a similar number of operations for doubles.

The first type of operators are the comparison operators, which result in a value of type boolean. These operators are $<, <=, >, >=, ==, !=$.

```
2.3 < 2.1;     // false
5.0 <= 4.3;    // false
0 > -.4;       // true
3.2 >= 3.2;    // true
0.1 == 0.2;    // false
2.10 != 2.100; // false
```

The second type of operators are the numerical operators, which result in a value of type double. These operators are $+, -, *, /$.

```
10.3 + 4.7; // 16.0
-8 - 10;    // -18.0
2.4 * 3;    // 7.2
4 / 5;      // 0.8
```

3.1.3 Char Operations

JavaLite provides a number of comparison operators for chars, which result in a value of type boolean. These operators are $<, <=, >, >=, ==, !=$ and compare ASCII values.

```
'A' < 'a';   // true
'b' <= 'c';  // true
'z' > 'z';   // false
'z' >= 'Z';  // true
'b' == 'B' ; // false
'e' != 'd';  // true
```

3.1.4 Boolean Operations

JavaLite provides two relational operators for booleans, which result in a value of type boolean. These are != and ==.

```
false != false;  // false
true != false;   // true
false == false;  // true
true == false;   // false
```

JavaLite also provides three conditional operators for booleans, which result in a value of type boolean. These are !, ||, and &&.

```
!false;         // true
!true;          // false
true || true;   // true
true || false;  // true
false || false; // false
true && true;   // true
true && false;  // false
false && false; // false
```

3.2 Objects

Aside from primitive types, JavaLite contains three kinds of reference types (class types, type variables, and array types). Class types have a type name. If arguments appear in a class type, then it is a parameterized type. Objects are a class instance or an array. Reference values are pointers to these objects, along with a special null reference that refers to no object.

3.3 Variables

Variables in JavaLite provide a storage location that is named. A variable in JavaLite must have a specific type to determine the size and layout of its memory. Once a type is declared, the value held in the variable must be of that exact type. A variable is declared using the operator '=' and must be declared with a name that of a valid string type.

```
// valid variable declarations
int x;
int x = 1000;
char ch = 'c';

// invalid variable declarations
int = 3;
4;
int x = 4.5;
char c = "string";
boolean b = 2;
```

Each variable must have a value before its value is used. Each variable is initialized with a default value when it is created.

- int (default is 0)
- double (default is 0.0)
- char (default is the null character '\u0000')
- boolean (default is false)

3.4 Strings

JavaLite uses the String class to create and manipulate strings. A string object has a constant value and represents sequences of Unicode characters. String literals are references to instances of a string object. Strings in JavaLite are mutable. Strings can be declared like follows.

```
string str = "Hello, World!";
```

3.4.1 String Operations

JavaLite provides two comparison operators for strings, which result in a value of type boolean. These are != and ==.

```
"str" != "stra";  // true
"hi" == "Hi";     // false
```

JavaLite also provides the abilitty to use the '+' operator to concatenate two strings.

```
"str" + "ing";    // "string"
```

3.5 Arrays

An array in JavaLite is an object, and is assigned to variables of an object type. An array contains zero or more elements, in which the array is said to be empty in the case of zero elements. The elements in an array have no names and are referenced by array access expressions that use positive integer index values. In an array with n elements, with n being the length of the array, the elements of the array are referenced using integers from 0 to n-1.

An array type in JavaLite is written as the name of the elements type followed by one or more empty pairs of square brackets [], determining the depth of array nesting. All elements in an array are of the same type. If the components in an array are of type T, then the type of the array is written T[]. Elements in an array may be of any primitive or reference type. Arrays can be declared like follows.

```
int[] arr;            // Arrays of ints
// arr = [];
int[] arr = int[5]    // Array containing five int elements.
// arr = [0, 0, 0, 0, 0]
string[] arr = ["hello", "world"];
```

3.5.1 Array Operations

JavaLite provides two comparison operators for Arrays, which result in a value of type boolean. These are != and ==.

```
int[] arr;
string[] strarr;
arr == strarr;          // false
[3, 4, 5] != [3, 4, 6]; // true
```

JavaLite also provides the ability to use the '+' operator to concatenate arrays and use the "[]" operator to access elements of an array.

```
int[] arr1 = [3, 4, 5];
int[] arr2 = [10, 3, 2];
arr1[2];      // 4
arr2[0];      // 10
arr1 + arr2;  // [3, 4, 5, 10, 3, 2]
arr2 + arr1;  // [10, 3, 2, 3, 4, 5]
```

4 Statements and Expressions

4.1 Statements

Statements in JavaLite are of the following forms:

- If-Else Statements

- While Statements
- For Statements
- Expressions
- Return Statements

4.1.1 If-Else Statements

If statements in JavaLite allow for the conditional execution of a statement.

```
if (expression) {
  statement1;
}
else {
  statement2;
}
```

The expression must be of type boolean, or a compile-time error occurs.

In this case, the first statement will execute if expression evaluates to true and otherwise, the second statement will be executed. The conditional must be wrapped in a () and the statement to be executed must be wrapped in {}.

It is also possible to have standalone if statements or if-else if blocks without the trailing else.

```
if (expression) {
  statement;
}
```

Here, statement would be executed if expression evaluates to true. Nothing is executed if the expression evaluates to false.

4.1.2 While Statements

While statements in JavaLite execute an expression and a statement repeatedly until the value of the expression evaluates to false.

```
while (expression) {
  statement;
}
```

The expression must be of type boolean, or a compile-time error occurs. Similar to If-Else Statements, the expressions must be wrapped in () and the statements must be wrapped in {}.

4.1.3 For Statements

For statements in JavaLite execute an initial statement and then executes an expression, statement, and update statement until the expression evaluate to false.

```
for (initial; expression; update) {
  statement;
}
```

The expression must be of type boolean, or a compile-time error occurs.

4.1.4 Return Statements

In JavaLite, return statements are used in functions (4.4) to return control (and data in some cases) to the invoker of a method.

```
T fun() {
  return expression;
}
```

Expression must be of type T, or a compile-time error occurs. In return statements, the expression is first evaluated then returned. For example,

```
int fun() {
  int x = 2;
  return x + 1;
}
```

In this case, $x + 1$ will be evaluated first, so the function will return 3.

4.2 Expressions

Expressions in JavaLite are a type of Statement that are of the following three syntactic forms:

- Primary Expressions
- Unary Operator Expressions
- Binary Operator Expressions

4.2.1 Primary Expressions

Primary Expressions in JavaLite are of the form of literals, or in classes using the this (5.2) keyword.

```
2;     // evaluates to 2
true;  // evaluates to true
```

4.2.2 Operator Expressions

Operator Expressions can be of unary or binary form. Operators specific to each of the types provided by JavaLite are discussed in 3.1.1, 3.1.2, 3.1.3, 3.1.4, 3.4.1, and 3.5.1.

4.2.3 Assignment Operator

In JavaLite, the assignment operator '=' stores values into variables. This expression is evaluated right-to-left.

```
T var = expression;
```

The expression is first evaluated then stored into the variable of type T with name var. If the variable and evaluated value of the expression are not of the same time, a compile-time error occurs.

4.3 Operator Precedence

JavaLite executes operators with different precedence. The following table details the different levels of precedence with the highest levels of precedence at the top.

Operator	Description	Associativity
[]	Array Access	Left
.	Object Member Access	
()	Parentheses	
!	Not	Right
*	Multiplication	Left
/	Division	
%	Modular	
+	Addition	Left
−	Subtraction	
+	Array/String Concatentation	
<, <=, >, >=	Comparison	Left
==, !=	Equality	Left
&&	Logical AND	Left
||	Logical OR	Left
=	Assignment	Right

4.4 Functions

Functions in JavaLite can be standalone or associated with objects (5). Functions take in a list of arguments and return one value.

```
T fun(R arg1, V arg2, ...) {
  /* do something */
  return expression;
}
```

Expression must be of type T. If not, a run-time error will occur. For example, we can write a function that increments values by 5 based on a boolean condition.

```
int fun(int x, boolean b) {
  if (b) {
    x =  x + 5;
  }
  return x;
}
```

Functions are called by calling its name and providing the required number of paramaters. A mismatch in the evaluated type of paramters or number of parameters will result in a compile-time error.

```
int fun(int x) {
  return x + 5;
}

fun();      // compile-time error
fun("hi");  // compile-time error
fun(2.3);   // compile-time error
fun(5);     // evaluates to 10
```

5 Classes

JavaLite supports classes similar to how Java does. Classes have instance variables and methods. Classes can be declared using the following syntax.

```
class Test {
  T x = expression;
  constructor (T x) {
    this.x = x;
  }

  T method() {
    /* some implementation */
    return val;
  }
}
```

The constructor is called when a object is initialized. For example,

```
x = Test(3);
```

An object of type Test and name x would be initialized by passing one argument of value 3 into the constructor function. The constructor is assumed to be of return type void.

5.1 Dot Operator

In JavaLite, the dot operator, '.', is used to access values of an object (an instance of a class). These values can be in the form of variables or methods. The dot operator is used in the following format.

```
x.var;
x.method(arg1, arg2, ...);
```

5.2 This Keyword

In JavaLite, the this keyword is used as a reference variable the refers to the current object. Most commonly this is used in the constructor of a class to remove confusion between instance variables and parameters that may have the same name.

```
class Person {
  string name;

  constructor(string name) {
    this.name = name;
  }
}
```

"this" can only be used inside class declarations. If used outside a class declaration, a compile-time error will occur.

6 Built-In Functions

JavaLite provides a number of Built-In functions that can be used.

6.1 Arrays

6.1.1 int indexOf(T e)

indexOf(T e) returns the index of the first occurence of element e in an array. If element e is not in the array, -1 will be returned.

```
int[] arr = [1, 2, 3];
arr.indexOf(2));  // 1
```

6.1.2 T pop()

pop() removes the last element of an array and returns it.

```
int[] arr = [1, 1, 2];
int a = arr.pop();
// arr = [1, 1], a = 2
```

Calling pop on an empty array results in a run-time error.

6.1.3 void append(T e)

append(T e) will insert element e to the end of the array.

```
int[] arr = [1, 1];
arr.append(2);
// arr = [1, 1, 2]
```

6.1.4 int length()

length() returns the length of the array.

```
int[] arr = [1, 1, 1];
int length = arr.length(); // 3
```

6.2 Strings

6.2.1 void upper()

upper() converts all characters in a string to upper-case.

```
string str = "hello";
str.upper();
// str = "HELLO"
```

6.2.2 void lower()

lower() converts all characters in a string to lower-case.

```
string str = "HeLLO";
str.lower();
// str = "hello"
```

6.2.3 string substring(int a, int b)

substring(a, b) returns the substring between index a (included) and b (excluded).

```
string str = "hello";
string str_sub = str.substring(0, 2);
// str_sub = "he"
```

- If a < 0, a compile-time error will occur
- If b $<$ a, a compile-time error will occur
- If b $>$ str.length(), a run-time error will occur

6.2.4 int indexOf(char c)

indexOf(c) returns the index of the first occurence of c in the string. If there is no occurence of c, -1 is returned.

```
string str = "hello";
int a = str.indexOf('h');
int b = str.indexOf('l');
int c = str.indexOf('k');
// a = 0, b = 2, c = -1
```

6.3 Print

JavaLite uses print to print strings to the terminal/console. Print can accept inputs of any type and arrays.

```
int[] arr = [1, 2, 3]
print("hi");  // "hi"
print(10);    // "10"
print(arr);   // "[1, 2, 3]"
```

- If print is called with more than one argument, a compile-time error will occur
- If print is called with any type other than int, double, float, boolean, char, string, or array of any of these types, a compile-time error will occur

7 Syntax/Style Guide

- Any line that is not the beginning or end of a bracket block must end with a semi-colon. If not, a compile-time error will occur.
- All opening symbols must be matched with a closing symbol. More specifically, (), {} and [] must be matched. If not, a compile-time error will occur.
- Each line should be a maximum of 80 characters wide.
- Proper indentation of 4 spaces should be used, however identation is ignored by the compiler.

8 Sample Code

8.1 Hello World

```
print("Hello, world!");
```

8.2 GCD

```
int gcd(int a, int b) {
  int divisor;
  int dividend;
  if (a > b) {
    dividend = a;
    divisor = b;
  }
  else {
    dividend = b;
    divisor = a;
  }
  while (divisor != 0) {
    int remainder = dividend % divisor;
    dividend = divisor;
    divisor = remainder;
  }
  return dividend;
}
```

8.3 Person Object

```
class Person {
  string name;
  // constructor is assumed to be void
  constructor (string name) {
    this.name = name;
  }
  void changeName(string newName) {
    this.name = newName;
  }
}

Person me = Person("Adam");
me.changeName("Mark");
print(me.name);
```

8.4 Functional Hello World

```
void sayHello() {
  print("Hello, world!");
}

sayHello();
```