Fall 2016 COMS4115
Programming Languages & Translators

Java+- Language Reference Manual

Authors
Ashley Daguanno (ad3079) - Manager
Anna Wen (aw2802) - Tester
Tin Nilar Hlaing (th2520) - Systems Architect
Amel Abid (aa3454) - Systems Architect
Zeynep Ejder (ze2113) - Language Guru

Contents

1 Introduction **4**

2 Lexical Elements **4**

2.1 Identifiers **4**

2.2 Keywords **4**

2.3 Operators **4**

2.4 White Space **5**

2.5 Comments **6**

3 Data Types **6**

3.1 Primitive Types **6**

3.2 Composite Types **6**

3.3 Tuples **7**

3.3.1 Creating Tuples 7

3.3.2 Accessing Tuple Elements 7

3.4 Arrays **7**

3.4.1 Declaring Arrays 7

3.4.2 Creating and Initializing Arrays 7

3.4.3 Accessing Array Elements 8

3.4.4 Multidimensional Arrays 8

4 Expressions and Operators **8**

4.1 Assignment Operator **8**

4.2 Arithmetic Operators: **9**

4.3 Comparison Operators: **9**

4.4 Logical Operators **9**

4.5 String Concatenation **9**

4.6 Operator Precedence **10**

4.7 Order of Evaluation **10**

5 Statements **10**

5.1 Expression Statements **10**

5.2 Declaration Statements **11**

5.3 Control Flow Statements **11**

5.3.1 Decision-Making Statements (if-then, if-then else): 11

5.3.2 Looping Statements (while, for) 12

5.3.3 Branching Statements (break, continue) 13

6 Functions **14**

6.1 Function Declarations **14**

6.2 The Return Statement **15**
6.3 Calling Functions **15**
6.4 The Main Function **15**
6.5 Recursive Functions **16**
7 Program Structure and Scope **16**
7.1 Program Structure **16**
7.2 Scope **16**
8 Standard Library Functionality **17**

1 Introduction

Java+- is an object oriented programming language largely based on Java that features the added functionality of tuples and compiles into LLVM.

2 Lexical Elements

2.1 Identifiers

Identifiers are sequences of characters used to name classes, variables, and functions.

Syntax for valid Java+- identifiers:

1. Each identifier must have at least one character.
2. Identifiers may contain capital letters, lowercase letters, digits, and the underscore symbol.
3. Identifiers may not begin with a digit
4. Identifiers are case sensitive.
5. Keywords may not be used as identifiers.

2.2 Keywords

`boolean`	`for`	`and`	`break`
`number`	`while`	`or`	`continue`
`char`	`if`	`public`	`new`
`true`	`elseif`	`private`	`void`
`false`	`else`	`null`	`return`

2.3 Operators

More information on operators can be found in the section titled *Expressions and Operators*. Supported operators include:

Arithmetic	**Comparison**	**Logical**
+	==	and
-	!=	or
*	<=	
/	>=	
	<	
	>	

2.4 White Space

White space is the collective term used for several characters: the space character, the tab character, the newline character, and the vertical tab character. White space is ignored (outside of string and character constants), and is therefore optional, except when it is used to separate tokens. This means that

```
public void getFive() {
      number five = 5;
      print(five);
}
```

and

```
public void getFive() {number five = 5; print(five);}
```

are functionally the same.

No white space is required between operators and operands.

```
/* All of these are valid. */
x++;
x ++ ;
x=y+z;
x = y + z ;
x=array[2];
x = array [ 2 ] ;
```

Furthermore, wherever one space is allowed, any amount of white space is allowed.

```
/* These two statements are functionally identical. */
x++;

x
       =x+1       ;
```

In string constants spaces and tabs are not ignored, but rather are part of the string. Therefore,

```
      "mac and cheese"
```

is not the same as

```
      "mac             and                cheese"
```

2.5 Comments

Java+- supports two types of comments

1. `/* multiline comments */` - The compiler will ignore everything between /* and */
2. `// single line comments` - The compiler will ignore everything from // to the end of the line.

3 Data Types

Java+- is a strongly typed language. One must specify a variable's type at the time of its declaration.

3.1 Primitive Types

Primitive Type	**Description**
`number`	a double-precision 64-bit (8-byte) IEEE 754 floating-point value. (There is no specific type for longs, integers, etc. and so all numeric values are represented via this number type.)
`char`	a single 16-bit Unicode character
`boolean`	a logical entity that can have one of two values: true or false

3.2 Composite Types

Composite Type	**Description**
`Tuple`	a sequence of immutable objects separated by commas and enclosed within parentheses
`array`	a fixed-size sequential collection of elements of the same type separated by commas and enclosed within square brackets
`String`	a sequence of characters surrounded by double quotes. each character in a string has an index, and indexing begins at 0. a string's length corresponds to the number of characters it contains.

3.3 Tuples

3.3.1 Creating Tuples

Upon creation, one must use the *new* operator and indicate the types of the objects that the tuple will contain.

```
Tuple<String, number, boolean> myTup = new Tuple<String, number, boolean>(“A string.”, 49, true);
```

3.3.2 Accessing Tuple Elements

Use square brackets to access an element at a particular index. Indexing begins at 0.

```
print(myTup[0]); // prints “A string.”
```

3.4 Arrays

3.4.1 Declaring Arrays

An array declaration has two components: the type of the array, and the name of the array. The type is indicated as *dataType[]*. The array’s name can be anything so long as it is not a Java+-keyword.

```
number[] myArr;
```

3.4.2 Creating and Initializing Arrays

The size of the array must be indicated at the time of its creation and is immutable.

One way to create an array is with the *new* operator. The following statement will allocate an array with enough memory for five number elements and assign it to the variable *myArr*.

```
number[] myArr = new number[5]; // OK
number[] myArr = new number[];  // NOT OK, must indicate array size
```

Alternatively, if you know what elements you want your array to contain, you can create one using the more concise syntax:

```
number[] myArr = [3, 6, 9, 12, 15];
```

3.4.3 Accessing Array Elements

Elements are accessed by their numerical index.

```
number[] myArr = [3, 6, 9, 12, 15];
print(“Elem at index 2: “ + myArr[2]); // prints “Elem at index 2: 9”
```

To assign or modify array values the syntax is as follows:

```
myArr[0] = 1;
myArr[1] = 45;
```

3.4.4 Multidimensional Arrays

A multidimensional array is one in which each array element is itself an array. Each array in the multidimensional array must be of the same type.

Use the following syntax to declare a multidimensional array.

```
number[][] = new number[5][4]; // OK (5 arrays each of length 4)
number[][] = new number[5][];  // OK (5 arrays of unspecified length)
```

The first set of square brackets indicates how many arrays will be contained - one must specify this dimension. The second set of square brackets refers to the length of each array contained in the multidimensional array and is optional - the array may contain arrays of different lengths.

4 Expressions and Operators

4.1 Assignment Operator

Operator	Type	Associativity	Example
=	Assignment	Right to left	number x = 5

Java +- implements the standard assignment operator, = , to store the value of the right operand in the variable of the left operand. Left and right operands must be of the same type. The left operand cannot be a literal value.

```
String sayHi = “hello world” // good
5 = 8;                       // bad
```

4.2 Arithmetic Operators:

Java +- has standard arithmetic operators that are applied to operands of type number. Left and right operands must be of the type number.

Assume variable X = 20, variable Y = 5:

Operator	**Type**	**Associativity**	**Example**
+	Addition	Left to right	X + Y will give 25
-	Subtraction	Left to right	X – Y will give 15
*	Multiplication	Left to right	X * Y will give 100
/	Division	Left to right	X / Y will give 4

4.3 Comparison Operators:

Comparison operators operate on operands of the same type.
Assume variable X = 20, variable Y = 5:

Operator	**Type**	**Associativity**	**Example**
==	Equal to	Left to right	(X == Y) will return false
!=	Not equal to	Left to right	(X != Y) will return true
>	Greater than	Left to right	(X > Y) will return true
>=	Greater than or equal to	Left to right	(X >= Y) will return true
<	Less than	Left to right	(X < Y) will return false
<=	Less than or equal to	Left to right	(X <= Y) will return false

4.4 Logical Operators

Operator	**Type**	**Associativity**	**Example**
and	Logical AND	Left to right	(true and false) will give false
or	Logical OR	Left to right	(true or false) will give true

4.5 String Concatenation

Strings can be concatenated with the use of the + operator to create a new String value. Both left and right operands must be of type String.

```
String line = “hello”;
line = line + “ world!”;
/* line is now “hello world!” */
```

4.6 Operator Precedence

For an expression that contains multiple operators, the operators are grouped based on rules of precedence.

The following list is presented in order of highest to lowest precedence; operators are applied from left to right:

1. Multiplication and division expressions
2. Addition and subtraction expressions
3. Greater than, less than, greater than or equal to, and less than or equal to expressions
4. Equal to and not equal to expressions
5. Logical AND expressions
6. Logical OR expressions
7. All assignment expressions

4.7 Order of Evaluation

Subexpressions will be evaluated from left to right.

```
 ( A() + B() ) + ( C() * D() )
```

According to this example, A() will be called first followed by B(), C(), and D() regardless of operator precedence.

5 Statements

In Java+-, a statement is used to create actions and to control flow throughout the program.

5.1 Expression Statements

A statement forms a complete unit of execution. Assignment expressions, method invocations, and object creation expressions may all be turned into statements by adding a semicolon (;) at the end.

```
x = 2.4;                          //assignment statement
print(“Hello World!”);            //method invocation statement
Car myCar = new Car();            //object creation statement
```

In addition to expression statements, there are two other types of statements: *declaration statements* and *control flow statements*.

5.2 Declaration Statements

A declaration statement is used to declare a variable by indicating its data type and name. One may also initialize it with a value. One may declare more than one variable of the same data type in the same declaration statement.

```
number x;
number y = 9.99;
number num1, num2, num3;
```

5.3 Control Flow Statements

Statements are generally executed from top to bottom. However, control flow statements break up the execution flow by performing decision making, looping and branching specific blocks of code.

5.3.1 Decision-Making Statements (if-then, if-then else):

These statements tell the program to execute a particular section of code *only if* a specified condition evaluates to true. The opening and closing braces are mandatory to avoid the dangling else problem. One must use the keywords *if*, *elseif*, and *else*.

```
if (x == 10) {       // “if” clause
  print("x is 10"); // “then” clause, will only execute if (x == 10) is true
}
```

An example involving else:

```
if (x == 10) {
  print("x is 10");
} else {
  print("x is not 10"); // executed if (x==10) is false
}
```

An example using a series of if statements to test for multiple conditions:

```
if (x == 1) {
  print("x is 1");
} elseif (x == 2) {
  print("x is 2");
} elseif (x == 3) {
  print("x is 3");
} else {
  print("x is something else");
}
```

5.3.2 Looping Statements (while, for)

While

The while statement repeatedly executes a block of code while a specific condition is true. Its syntax is as follows:

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

The *expression* above must return a boolean value. If it evaluates to true, the while statement executes the statement(s) in the while block. The while statement continues testing the expression and executing its block until the expression evaluates to false.

An example of printing numbers 1 through 5 using a while statement:

```
number count = 1;
while (count < 6) {
  print(“Count is: “ + count);
  count = count + 1;
}
```

For

The for statement repeatedly loops over a code block until a particular condition is satisfied. Its syntax is as follows:

```
for (initialization; termination; increment or decrement) {
   statement(s)
}
```

These rules apply to the for statement:

1. The *initialization* expression is executed once as the loop begins.
2. The *termination* expression must return a boolean value. The loop stops executing once it evaluates to false.
3. The *increment* expression is invoked after each iteration of the for loop.

Here’s how to print numbers 1 through 5 using a for statement:

```
for (number count = 1; count < 6; count=count+1) {
  print(“Count is: “+ count);
}
```

5.3.3 Branching Statements (break, continue)

Break

When the break statement is encountered inside a loop, the loop is immediately stopped and the control flow resumes at the next statement following the loop.

```
/* This code block will print “123” */
number[] numbers = {1,2,3,4,5};
for (number i = 0; i < numbers.length; i = i+1) {
  if (numbers[i] == 4) {
    break;
  }
  print(numbers[i]);
}
```

Continue

The continue statement causes the loop to immediately jump to the next iteration of the loop. In a for loop, it causes the control flow to immediately jump to the *increment/decrement* statement. In a while loop, it causes the control flow to jump to the boolean expression.

```
/* This code block will print “1235” */
number[] numbers = {1,2,3,4,5};
for (number i = 0; i < numbers.length; i = i+1) {
  if (numbers[i] == 4) {
    continue;
  }
  print(numbers[i]);
}
```

6 Functions

Functions are useful to separate your program into smaller logical pieces. Each function must be declared before it can be used.

6.1 Function Declarations

The required elements of a method declaration are a modifier (**public** or **private**), its return type, a name, a pair of parentheses (), and a body between braces {}.

```
public/private return-type function-name (list of parameters)
```

- *public/private* indicates whether or not other classes can access this method. When public the function is globally accessible; when private the method is only accessible within its declared class.
- *return-type* indicates the type of the object being returned from the function. A function may return void, indicating that it does not return anything.
- *function-name* may be any valid identifier.
- *list of parameters* refers to a comma separated list of input parameters and is optional, but empty parentheses must be present if there is no list.
- Each function declaration must be followed by an opening brace { after which the body of the function should begin. The function body should be followed by a closing brace }.

Every function declaration within a class should have a unique combination of *name* and *list of parameters*.

Here is an example of a function declaration with no parameters:

```
public void myFunction() {
  statement(s);
}
```

A function declaration with two parameters that returns a number:

```
public number myFunction(number x, number y) {
  return 5;
}
```

6.2 The Return Statement

The return statement ends the execution of the function and returns the program control to the function that initially called this function.

```
return return-value;
```

The **return** keyword is reserved for denoting a return statement. It should be followed by the object being returned. The type of the returned value should match the return-type defined in the function declaration.

An example of a function with a return statement:

```
public number addOne(number x) {
  number y = x + 1;
  return y;
}
```

6.3 Calling Functions

You can call a function by using its name and its required parameters. Using the above addOne function:

```
addOne(1);
```

A function call can make up an entire statement (example above) or it can be a sub-statement as in the following example:

```
number result = addOne(1);
```

6.4 The Main Function

This is the entry point to your program. Every program in Java+- should have a main function. The **main** keyword is reserved for this function. The function returns void. Java+- supports command line arguments.

Here is how your main function should be structured at all times:

```
public void main(String[] args) {
  body statement(s)
}
```

6.5 Recursive Functions

Java+- supports recursive function calls.
Here is an example of a recursive function that computes the factorial of a number:

```
public number factorial(number x){
  if (x < 1) {
    return 1;
  }
  else{
    return (x * factorial(x - 1));
  }
}
```

Java+- does not have any safeguards to prevent infinite recursion.

7 Program Structure and Scope

7.1 Program Structure

A Java++/-- program can exist in one file, but more commonly an elaborate program will be broken into Objects and each object will have its own source file.

Each source file should have the extension .jpm.

7.2 Scope

Scope refers to what Object can see what variable or function. Anything that is declared as **public** can be seen by anything. Anything declared as **private** can only be seen within the Class it is declared in.

Variables that are declared inside functions are only visible inside that function and their lifespan is only that function.

```
public void example(){
      number x = 0;
}
```

x is not accessible by anything outside the scope of function example().

8 Standard Library Functionality

`print()`	A function that prints the arguments passed to it to stdout.
`Dictionary`	A mapping type which maps keys (of any type) to values (of any type). Keys must be unique within the dictionary.

print()
Raw string arguments must be enclosed in quotes when passed to print(). Separate arguments using the character '+'.

```
boolean printMe = true;
print(printMe);                          // prints "true";
print("Hello World")                     // prints "Hello World"
print("My number is " + 42 + ".");       // prints "My number is 42."
```

Dictionary

One way to declare a dictionary is to use the *new* keyword and pass the size of the dictionary to the Dictionary constructor.

```
Dictionary{} myDict = new Dictionary{}(3);
```

Alternatively, you can use the shortcut syntax:

```
Dictionary{} myDict = {'42': 'Ashley', 'Peanuts': 'Anna', 'Sheep': 'True'};
```

To access dictionary elements, use square brackets:

```
print(myDict[42]);            // prints "Ashley"
```