

The Java Language

The Java Language Reference (2nd ed.) is the defining document for the Java language. Most beginning programming students expect such a document to be totally beyond them. That expectation is wrong. The JLR is a mix of computer science jargon and some really clear English descriptions. It is not a book to read from cover to cover, in order. It is a reference book that you use to answer questions as you learn to program. The JLR has 18 chapters which we will describe very briefly in the following sections. Some we will basically skip while others we will examine in some detail. The 18 chapters are titled:

1. Introduction
2. Grammars
3. Lexical Structure
4. Types, Values and Variables
5. Conversions and Promotions
6. Names
7. Packages
8. Classes
9. Interfaces
10. Arrays
11. Exceptions
12. Execution
13. Binary Compatibility
14. Blocks and Statements
15. Expressions
16. Definite Assignment
17. Threads and Locks
18. Syntax

The JLR can be purchased in book form or can be downloaded from java.sun.com in either html or pdf format. The following sections are a brief introduction to the JLR. While there are many books on Java, including this one, remember that it is always best, no matter what field you are studying, to look at source documents. If you are studying US government, you need to read the US Constitution. If you are studying Java, you need to read the JLR.

Introduction

The introduction provides sample programs, a note on the notation used in the rest of the book, a list of special classes and a list of reference works. These special classes include Object, String, Thread, Class, ClassLoader and some interfaces. Remember that a class is the Java representation of an object.

Grammars

As with English, grammar is the set of rules that determine the validity of an expression or sentence. For example, in a natural language there is a rule that a sentence must have a noun and a verb to be a valid sentence. Computer languages have similar rules. It is not necessary to understand how computer grammars work to be able to effectively program.

Lexical Structure

```
1. /* Hello World Example */
2. class HelloWorld {
3.     public static void main(String[] args) {
4.         System.out.println("Hello World");
5.     }
6. }
```

Lexical structure refers to the Java source code which is the text that you type in. The elements of lexical structure include comments, identifiers, keywords, literals, separators, operators and line terminators, white space.

Lexical analysis is the process of determining the lexical structure of the source code. What that means is figuring out what each character in the source code means. This is a two part process. First the source code is

broken up into a list of tokens, where a token is simply one of the lexical elements, e.g. identifier, operator, etc. Each token is delimited, surrounded by, white space (space, tab, carriage return, etc.) or delimits itself by being a separator or operator. After each token is found, *parsed*, a lookup is performed to determine which kind of lexical element the token is.

Looking at the sample program at left, there are concrete examples of many lexical elements. The program starts with a comment on line 1. A comment is, essentially, a single token and anything inside of the comment delimiters, `/* & */`, is ignored by the compiler. Line 2 has three tokens: a keyword, class, an identifier, HelloWorld, and a separator, `{`. Line 3 has 11 tokens. Line 4 has 9 tokens while the last two have 1 token each. Lexical analysis is not something that you will be doing as a programmer but learning the rules is very important.

Unicode

Java uses Unicode for storing character data, including the source code. You are probably familiar with *ASCII*, American Standard Coding for Information Interchange, the traditional character set for small computers. Standard ASCII consists of 128 characters. This

is all the characters on your computer keyboard, including upper case and lower case letters, numerals and punctuation characters plus a few more. Since computers are digital devices, there are no letters inside a computer; there are only numbers. ASCII is a *mapping* which makes equivalence between symbols and numeric values. For example, 'A' is 65, 'B' is 66, 'a' is 97, '2' is 50, etc. As it turns out, for English ASCII works well since there are less than 100 distinct symbols used. However, computers are no longer English-only devices. Languages like French and German have additional characters like 'ç' and 'ë'. Languages like Russian, Hebrew and Arabic have entirely different alphabets while Chinese does not have an alphabet at all. The additional characters make 128 a ridiculously small number. Unicode is a superset of ASCII and has room for 65536 distinct symbols. The first 128 characters of Unicode are the same as ASCII, while the remaining 65408 characters contain a variety of other symbols. In Java, most things use only the ASCII subset of Unicode. But comments, identifiers and string literals can contain any Unicode character that you choose.

Comments

Java, like C, has two types of comments, block and end of line. A block comment starts with a `/*` and ends with `*/`. Everything in between the delimiters is ignored by the compiler. You can put any text you like inside a comment except `*/`. That is, `/* this is a comment */` is a valid comment but `/* /* this is a comment inside of a comment */ */` is not valid. This concept is called *nesting* and in Java, comments do not nest. Block comments are also used for Javadoc, which is used to document source code. A later section will describe Javadoc in enough detail for it to be useful to you.

End of line comments are very simple. They start with a `//` and end at the end of the line.

Identifiers

Identifiers are the names that you create for classes, methods, properties, etc. In the Hello World example, the class name HelloWorld is an identifier. In Java, an identifier begins with a letter, either upper or lower case, and is followed by letters, digits, “_” or “\$”. Note that operators and separators act as delimiters and so cannot be part of an identifier.

Keywords

Keywords are the identifiers that make up the Java language. Java has 48 reserved words, although it uses only 46 of them. It may seem surprising but Java does not include functions to read from the keyboard, write to the screen, manipulate files, do math functions or many other things. That functionality is included in the standard packages, which will be introduced later.

The keywords are in this table, categorized by function. There are four basic categories: defining or declaring classes, methods and properties, specifying variable types, controlling the program flow and using objects. Note that primitive variable types are used in method declarations if a method returns a value. The two reserved words that are not used in Java are `const` and `goto`.

class and method definition or declaration	<code>abstract</code> , <code>class</code> , <code>const</code> , <code>extends</code> , <code>final</code> , <code>implements</code> , <code>import</code> , <code>interface</code> , <code>native</code> , <code>package</code> , <code>private</code> , <code>protected</code> , <code>public</code> , <code>static</code> , <code>strictfp</code> , <code>synchronized</code> , <code>throws</code> , <code>transient</code> , <code>void</code> , <code>volatile</code>
primitive variable types	<code>boolean</code> , <code>byte</code> , <code>char</code> , <code>double</code> , <code>float</code> , <code>int</code> , <code>long</code> , <code>short</code>
flow control	<code>break</code> , <code>case</code> , <code>catch</code> , <code>continue</code> , <code>default</code> , <code>do</code> , <code>else</code> , <code>finally</code> , <code>for</code> , <code>goto</code> , <code>if</code> , <code>instanceof</code> , <code>return</code> , <code>switch</code> , <code>throw</code> , <code>try</code> , <code>while</code>
object manipulation	<code>new</code> , <code>super</code> , <code>this</code>

Operators

Operators are special characters that are usually used to manipulate primitive variables. There are 37 different operators in the Java language, many of which you will never use. The most commonly used operators are assignment, `=`, equals, `==`, the four math functions, `+`, `-`, `*` and `/`, logical or, `||`, and logical and, `&&`. For every simple operation, like `+` or `-`, there is a corresponding combined operation and assignment, `+=` or `-=`.

Literals

Literals are values for primitive types and Strings. The best way to explain what literals are is to give examples. Integer literals are numbers without decimal points, e.g. `0`, `33`, `-157`, `1234321`, etc. Real literals are numbers with decimal points and, optionally, exponents, e.g. `3.3`, `-47.0`, `5E03`, etc. Character literals are single characters surrounded by single quote marks, e.g. `'a'`, `'D'`, `'?'`, etc. There are two boolean literals, `true` and `false`. String literals, the only non-primitive type with literals, are zero or more characters surrounded by double quotes, e.g. `"string"`, `" another string "`, `""`, etc. Finally, there is one literal for reference types, `null`, and it is used to indicate that a reference variable is not referring to anything.

Note that this is an introduction to literals. There are many variants beyond what is described here so you should refer to the JRL for more information. However, this introduction should be adequate to get you started programming Java.

Separators

Separators are the final elements of lexical structure. There are nine separators in Java and six come in pairs, `{` and `}`, `(` and `)`, and `[` and `]` while the other three are `;`, `.`, and `,`. All of them act as delimiters. The squiggly brackets are used to delimit a block of code, creating a block statement. The parentheses delimit the parameters for a method. The square brackets delimit the index of an array.

The semicolon delimits the end of a simple statement. The period connects instance variables with member methods or properties. The comma is used to delimit parameters in method calls. Examples of what this means will follow.

Types, Values and Variables

Variables are things with names, identifiers that are used to hold other things. The things that variables can hold come in two flavors, primitive values and references to objects. Java is a strongly typed language. That means that when you declare a variable you must also specify its type.

Primitive Types

There are 8 primitive data types: boolean, int, short, long, byte, float, double and char. These 8 types break into 4 groups: logical, integer, real and character. Logical variables hold one of two values: true and false. Logical variables are used almost exclusively in program flow control statements. That is, if a condition is true your code does one thing and if it is false your code does something else. Integers are natural numbers and include the byte, short, int and long types. byte is the smallest, 1 byte, of the integer types and has a range of -128 to 127. short is next in size, 2 bytes, and has a range of roughly -32000 to +32000. int is the next in size, 4 bytes, and has a range of roughly -2 to +2 billion. long is the biggest integer type, 8 bytes, with a range of roughly $\pm 10^{20}$. Integers are used for counters in loops and as properties in objects. Real numbers are numbers that have a decimal point. Real variables come in two sizes, float, 4 bytes, and double, 8 bytes. float is accurate to about 7 significant digits while double has roughly 16 significant digits. Real variables are used primarily to hold data about the real world. Finally, char is used to hold a single character. In Java, char is a Unicode character which means that not only can it hold a single Latin character, it can also hold characters from a variety of non-English alphabets and Chinese and Japanese characters.

Reference Types

In previous sections, we discussed objects at length. Creating an instance of an object without having a place to hold it would be pointless. Reference type variables are used to hold references to created instances of objects.

Conversions and Promotions

This section describes what is and is not allowed when attempting to make assignments between variables of different types. In general, such assignments are not permitted. Within the integer and real groups, assignments between variables of different types are allowed but if the assignment is from a larger sized variable to a smaller sized one (*narrowing*), it may cause an error. For example, if you try to assign 1000000 to a short integer variable, that causes a silent error because variables of type short can only hold values

from roughly -32000 to 32000. Likewise for reference types, between classes in the same family tree, it is possible to assign an instance of a derived class to a variable of the super class. For example, any object can be assigned to a variable of type Object, since Object is the starting point for all classes. And it is possible to assign a variable of type Object to a variable of its own class, provided you specify which class it is. But it is not possible to hold references to instances of unrelated classes in reference variables.

Names

A name is an identifier that has been declared to be of a specific type. A name has scope; a name is defined only within the block in which it is declared.

There are conventions for how to create names. Nothing enforces these rules but violating them indicates that you are not a “good” Java programmer.

1. Class names are concatenated nouns with the first letter of each capitalized and the rest lowercase, e.g. MyRobot.
2. Method names are verb phrases with the first word all lowercase and the rest initial uppercase, e.g. startMyRobot. Methods that change or test properties should use “set” or “is” as the first word of the phrase, e.g. setPowerLevel, isRunning.
3. Field names are concatenated nouns with the first word all lower case and the rest initial uppercase, e.g. powerLevel. Care should be taken to not obscure method names with field names.
4. Constants should noun phrases with each word separated by an underscore, “_”, and all uppercase, e.g. MAX_POWER.
5. Local variables should be short and meaningful. An acronym is good, e.g. mr for a variable of type MyRobot. Shortening of words in noun phrases is good, e.g. buf for buffer or out for output. One character names should be avoided.

Packages

A package is a set of classes that share a common theme. Most of Java’s functionality comes from the standard packages which are java, the core java package, and javax, the java extensions package. The Java language is tailored to different environments, PCs, microwave ovens, etc., by using different versions of the standard packages. LeJOS replaces the standard java package with its own for use on the RCX. The replacement package is significantly smaller with far fewer methods than the standard package because the RCX has extremely limited memory. Yet, it is still Java that is run on the RCX. Later, we will compare the java package from the JDK with the LeJOS java package.

Classes

A class is Java's way of representing an object. All classes are derived, either directly or indirectly, from the class `java.lang.Object`. Classes have two kinds of members: methods and fields. Methods are functions or subroutines that are related to the object. Fields are attributes or properties of the object. Both methods and fields can be either instance or class. Classes can be derived from other classes, extending the class.

There are several reserved words that act as modifiers for class, method and / or field declarations. These reserved words are `abstract`, `extends`, `final`, `implements`, `native`, `private`, `protected`, `public`, `static`, `strictfp`, `synchronized`, `throws`, `transient`, `void`, and `volatile`.

`extends` and `implements` modify only class names. `extends` declares that a new class inherits all of the methods and fields of another class. In previous examples, the class `BassetHound` extends the class `Dog`, e.g. `class BassetHound extends Dog`. A class that extends another class almost always adds new methods or fields and / or overrides existing methods or fields. A class that extends another is called a *subclass* while the class that is extended is called the *super-class*. Note that any method that can be called from an instance of a super-class can be called from an instance of the subclass. `implements` is used to specify that a class has all the methods and fields that are specified in a special type of class called an interface. There are two major differences between extending and implementing. First, a new class can extend only one other class while it can implement as many as needed. Second, methods from a super-class already exist whereas an interface requires the programmer to write the definition of the methods.

`abstract` is a modifier for class that creates a hybrid between a super-class and an interface. An abstract class has some methods that are declared but not defined. The programmer must write the declarations for the abstract methods. But there usually are non-abstract, i.e. declared, methods in an abstract class so it is different from an interface.

`private`, `public` and `protected` are modifiers that refer to where classes, methods and fields can be used and where they cannot be used. Something declared `public` can be used anywhere. Something declared `private` can be used only within the class where it is declared and nowhere else, not even in subclasses. Something declared `protected` is between `public` and `private`; it can be used only within the class where it is declared and within any subclasses. The convention in Java is to make most fields `private` and most methods `public`.

`static` modifies classes, methods and fields. It is used to say that there is only incarnation of whichever type of thing it modifies. For example, `Math` is a static, Java class. Ponder what an instance of `Math` would be. All the methods in the class `Math` are static, e.g. `Math.cosine`, `Math.log`, etc. Likewise, `Math.PI` is the value of pi. These are all things that you only want one of. In LeJOS, `Motor.A`, `Motor.B` and `Motor.C` are all static variables and refer, respectively, the motors attached to ports A, B and C. static fields or methods are also called class fields or methods. This is in contrast to instance fields or methods, which is the default. In a program, there is exactly one incarnation of a class field or method whereas there is one incarnation per instance for non-static fields.

`final` says that whatever it modifies cannot be changed. A final method cannot be overridden. A final field cannot have its value changed. `Math.PI` is an example of a final field. Unless you change universes, it would be unwise to change the value of `pi`.

`transient` and `volatile` indicate a field that changes. A transient field is one that is not stored when its instance is stored. A volatile field is one that can have a new value assigned to it outside of your program. It is very unusual for a beginning programmer to encounter either one.

`native` indicates that a method is object code for the real machine your program is running on and not Java byte code. You cannot declare a method `native` and then write code in Java to define it. You must, instead, write the method in another language, like C++ and then perform magic to make it work in Java. Note that native code is used in only a very few places, like the JVM, and makes a program non-portable. A beginning programmer should never declare native methods.

`strictfp` requires classes or methods to use special libraries for real number manipulation. Ordinarily, floating point numbers, `float` and `double`, use the computer's hardware math routines. This means that a JVM running on a MS Windows machine may produce slightly different results than a Sun or Macintosh for floating point numbers. The differences are usually very, very small and of no real consequence. However, there are some cases where the results must be identical. In that case, specifying `strictfp` causes the JVM to use software math routines instead of the computer's hardware routines. This causes the program to produce identical results without regard to the computer you use at the cost of, usually, much longer execution time. A beginning programmer will never need to use `strictfp`.

`synchronized` indicates that a method must be executed without interruption. Java is a multi-threaded environment. At any time, the JVM can stop executing one thread and start another. Obviously, when the JVM does this, it saves enough information so that it can resume the stopped thread without any problem. Usually, this method is good enough. However, there are times when a method cannot be interrupted in this manner. In those cases, declaring a method `synchronized` guarantees that the body of the method will not be interrupted. Note that you should be very careful in declaring methods `synchronized` because `synchronized` methods interfere with the normal scheduling of the JVM, which may result in truly obscure and difficult to debug errors.

`throws` is used to declare which exceptions a method can throw. Any method that throws an exception must be placed inside a `try {} catch () {}` block.

`void` indicates that a method does not return anything.

Interfaces

Interfaces are a special kind of class. In Java, a class can be derived from only one parent class. However, a class can implement as many interfaces as needed. An interface is simply a specification that guarantees that a class contains methods and/or fields with specific names and signatures.

Arrays

An array is simply an ordered list of things. The things in an array can be any of one type, either primitive or reference. A specific element in the array is accessed by supplying an index integer.

Exceptions

An exception is an unusual or unexpected event that interferes with the normal flow of a program. Dividing by zero is a classic example. After dividing by zero, it is fairly clear that your program should not continue running as if nothing happened. But what should it do? Java is an unusual language in that it allows a program to detect and deal with almost any unexpected event instead of having the system simply kill the program.

Java divides these events into two classes: errors and exceptions. Errors are events that are totally unpredictable or could happen in some many places that constantly checking for them is impractical and how to recover from them is unclear. Running out of memory is an error. Events that are predictable are exceptions. Java uses exceptions for events such as end of file while reading in a file. Exceptions are handled using a `try {} catch () {}` set of blocks. Programs can generate their own exceptions using a `throw` statement. Exceptions and exception handling adds a get deal of flexibility to Java for dealing with unusual events.

Execution

Java is a very secure language. In order to achieve that security, there are very specific rules on executing Java programs on different. The JRL defines those rules. Most beginning programmers will not need to be concerned about these specifics.

Binary Compatibility

Binary compatibility is the ability of a single executable program to be run on a variety of different computers. Java, through the use of a virtual machine, maintains binary compatibility on all supported platforms. The JRL description of binary compatibility is extensive and not especially necessary for a novice programmer to understand.

Blocks and Statements

A block is one or more statements surrounded by curly braces, “{” and “}”. A variable is local to a block if it is declared inside of the block. The scope of (it is accessible) a variable is the block that it is declared in and all sub-blocks of that block.

A statement causes something to happen and does not have a value. Statements are delimited by a trailing semicolon. Java has 13 distinct statements, most of which correspond to reserved words. The two that do not correspond to reserved words are the empty statement and the expression statement. An empty statement is simply a semicolon preceded by nothing but white space. An expression statement is simply an expression, which may be an assignment, a method call or an increment operator, followed by a semicolon. The remaining 11 statements are `if / if else`, `switch`, `while`, `do`, `for`, `break`, `continue`, `return`, `throw`, `synchronized`, and `try catch / try catch finally`. These statements all control the flow of execution of the program, that is, they determine which statement to execute next. We will not describe the syntax of these statements now. Instead, as we continue programming their usage will be clear.

Expressions

An expression is something that evaluates to a value. This includes numeric literals, e.g. 3, method calls, e.g. `mr.getPowerLevel()`, mathematical operators, e.g. `3 * 4`, among other things. A variety of operations and expressions will be introduced through the programming exercises. Listing them here would create a long and boring list. You are encouraged to look through the JRL if you are interested.

Definite Assignment

Definite assignment is simply the requirement that every variable declared in Java must be assigned a value before it can be used. That seems like a reasonable requirement because there are exceedingly few instances when your program would want to use a variable before it had explicitly assigned a value to it. Almost always, using a variable before having assigned it a value is an error. Having programmed in languages that don't have this requirement, I can assure you that it is an error that can be extremely difficult to find. The JRL devotes many pages to this topic because from a computer science perspective, it is very important and very hard to prove that a variable really has been assigned a value. For a beginning programmer, definite assignment is not an important consideration. If you do forget to initialize a variable, you will get a compiler error and be required to correct it before your program will successfully compile. The simplest thing to do is whenever you declare a variable, give it a 0 or null value, e.g. `int intvar = 0;` or `MyMotor mm = null;`. This is a good habit to get into because this kind of habit makes it harder for you to make a coding error. The best programmers have all developed good habits like this and so should you.

Threads and Locks

A thread is separate execution path. When you start a Java program, one thread is created. Your program can start additional threads. For example, a robot may start one thread for each sensor it has. This is a very efficient method of programming a robot.

Threads all share the same memory space. For example, Motor.A is a static variable that controls the motor attached to port A on the RCX. All the threads in your program have the ability to change Motor.A. This could cause problems if, for instance, the thread for one sensor was turning the motor on forward and another sensor was turning it on backward.

Java provides a mechanism of locks based upon methods, wait, notify and notifyAll, built into java.lang.Object, the super-class of all classes. These locks allow one sensor's thread to take ownership of the motor for a long enough time to complete its task and then yield to another sensor's thread so the second thread can do what it needs to do.

Threads and locks are one of the most powerful and advanced features built into Java that really distinguishes Java from other languages. Understanding how this works is best done through code examples for the robots.

Syntax

The last chapter presents the BNF, Baccus Naur Form, of the Java grammar. To computer science students wanting to write a parser and lexer for Java, the chapter is quite interesting. For most programmers, you might learn something from reading it but you might not.