Introduction to the Java Language

Basic Syntax

Comments

Java has two kinds of comments: traditional comments (also allowed in C, C++, SQL, and some other languages), which begin with `/*`, end with `*/`, and may span multiple lines; and "trailing" or end-of-line comments (also allowed in C++; many other languages have their own styles of end-of-line comments), which begin with `//`, and continue until the end of the physical line.

Example: *Code Comments*

```
/*
   This is a traditional comment. It may include any amount of text, over any
   number of lines; the compiler will ignore everything up until the closing
   characters.
 */

/*
 * This is a traditional comment, in the Sun-recommended format for multi-line
 * comments. The leading asterisks are not required (in fact, they are ignored
 * by the Java compiler, like all text in the comment), but they help the
 * reader to see comment blocks more easily.
 */

/* This is a traditional single-line comment. */

// This is a trailing/single-line comment; it ends when the physical line ends.

x = 5; // This is a trailing comment, following a statement (recommended form).

y = 1; /* This is a traditional comment, following a statement. */
```

Case Sensitivity

The Java syntax is largely derived from the C and C++ languages, and shares with those languages many basic characteristics. For example, Java is case-sensitive: keywords (such as `if`, `for`, `class`, `return`, etc.) are always typed in lowercase, and if uppercase versions are used in Java code, the Java compiler will not recognize them as keywords; similarly, the case used in a variable, class, interface, or package reference must match the case used in the corresponding declaration, or the Java compiler will not recognize the reference.

Example: *Case sensitivity*

```
int sum = 0;
int upperLimit = 100;
int lowerLimit = -100;

If (sum > upperLimit) { // Compiler error - The keyword is "if", not "If".
    return;
}

if (Sum < lowerLimit) { // Compiler error - the variable is "sum", not "Sum".
    return;
}
```

By long-established convention (codified in Sun's recommendations), class names usually begin with an uppercase letter, while variable and method names begin with a lowercase letter. Class, variable, and method names consisting of multiple words cannot have embedded spaces; to make such names easier to read, it is conventional to start each word after the first with an uppercase character (e.g. we might have a variable named `upperLimit`, or a class named `TicTacToe`).

Simple Statements

There are many kinds of statements in Java: assignment statements; method calls; declarations of variables, classes, and interfaces; flow control statements; exception management statements; etc. Some of these (certain declaration and flow control statements) will be addressed later in this session, and in the more in-depth Java session. For the moment, we will focus on simple statements, consisting of assignments, method calls, and/or expression evaluation.

In an assignment, the value of an expression is assigned to a variable, or to an element of an array. A single equals sign (`=`) is used as the assignment operator.

An expression consists of one or more operands (variables, literals, or method invocations which return values) and one or more operators (`+`, `-`, `/`, `-`, `%`, etc.). Note that the operator consisting of two equals signs (`==`) is used to compare two operands for equality, while a single equals sign is used for an assignment; *be careful* not to confuse these two operators.

A call to a method which returns a value may be an operand in an expression; a call to a method which does not return a value cannot.

If the statement does not include an assignment, it may consist of a call to a method which does not return a value.

Example: *Simple statements*

```
x = 10;                // Assignment statement; expression has one operand.
y = x + 5;             // Assignment; expression has multiple operands.
z = x * Math.sqrt(y);  // Assignment; expression includes call to a method.
a = (x == y);          // boolean assignment, based on equality of operands.
System.out.println(z); // No assignment; call to a method.
```

Statement Termination

Every simple Java statement must end with a semicolon. Note that this does not mean that each physical line must end with a semicolon: a simple Java statement may span multiple lines (in fact, if such a statement is very long, it may be a very good idea to split it into multiple physical lines, for legibility).

Example: *Statement termination*

```
int x = 0;     // OK - the statement is properly terminated.
int y = x + 1  // Compiler error - the statement isn't properly terminated.
int z =
    y + 10;    // OK - the statement is properly terminated.
```

Statement Blocks

A statement block is a group of statements, enclosed within a pair of curly braces. In most cases, a statement block may be used in place of a simple statement; this usage is very common (and recommended) within `for`, `if`, `if-else`, `while`, and `do-while` statements. There are also some cases — namely, the definition of a class, interface, method, or static initializer — where a statement block is required.

Note that a semicolon is not needed (nor is it recommended) after the closing right curly brace of a statement block.

Example: *Statement blocks*

```
public int getBalance() {       // A method definition; must use a block.
    int sum = 0;
    int upperLimit = 1000;

    while (sum < upperLimit) { // The left curly brace begins a block.
        sum += getNextDeposit();
        sum -= getNextWithdrawal();
    }                           // This brace closes the while loop block.
    return sum;
}                               // This brace closes the method block.
```

White Space

There are certain cases where white space (one or more spaces, tabs, line feeds, carriage returns, or form feeds) is required. In particular, white space is required between identifiers (keywords, variable names, primitive type names, class names, interface names) and other identifiers — but not between identifiers and operators (`+`, `-`, `/`, `-`, `%`, etc.). On the other hand, white space is not allowed within identifiers; thus, class names and variable names (for example) cannot have spaces embedded in them. Also, there are a few operators that consist of more than one character (`<=`, `>=`, `&&`, `+=`, `instanceof`, etc.); white space is not permitted between the characters of such operators.

When white space is allowed, any amount of white space can be used; anything beyond the minimum required white space is ignored by the Java compiler. Thus, the primary purpose of white space in such cases is to improve readability.

One case which might not be immediately apparent, where certain kinds of white space — namely, line feeds, carriage returns, and form feeds — are forbidden, is within string literals. If we want a string literal to include a line break, we would do that by including the new line character (`\n`) in the quoted string literal. On the other hand, if we want to break a quoted string literal across multiple lines to improve readability of our code, we would do that by breaking the string literal into multiple parts (on multiple lines), and concatenating them with the `+` operator.

Example: *White space*

```
int x = 0           // This line is fine ...
int y=1;            // ... and so is this one (but harder to read) ...
intz = 2;           // ... but not this one: "int" and "z" need a space between.

String s = "A";     // This line is fine ...
String t = "B\nC";  // ... and so is this one (it includes a line break)...
String u = "D
    E";             // ... but not these two: they break in a string literal.

x++;                // Good: "++" is a special operator.
y+ +;               // Bad: we have a space in the operator.
```

Variables

In Java, variables may be declared within a method or constructor (in which case, the variables and their values are not visible outside the method or constructor), or outside of methods and constructors, but within a class (in which case, the variables and their values are visible to the methods of the class, and may also — depending on the accessibility declared for the variables — be visible outside the class).

A variable is minimally declared by specifying the type and the name of the variable. For class-level variables, the type may be preceded by accessibility and scope modifiers.

A variable declaration can include assignment of an initial value to the variable in the same line.

Multiple variables of the same type can be declared in a single declaration; this should be done sparingly, and should never be done in combination with assignment of an initial value.

Example: *Variable declaration and use*

```
class Bacterivore {
    public static int populationSize = 0; // Integer, visible outside class.
    private int energy = 100;             // Floating-point private variable.
    private float x, y;                   // Multiple variables declared.

    private void move() {
        float dist = Math.random();       // Visible only within the method.
        float dir = 2 * Math.PI * Math.random();
        x += dist * Math.cos(dir);        // Modifying a class variable.
        y += dist * Math.sin(dir);        // Modifying a class variable.
    }
}
```

Flow-control Statements

This section is intended to be a quick introduction to flow-control techniques in Java. While some examples are given, they are not exhaustive, and most people learning Java will probably want to review additional reference or tutorial material. Sun Microsystems publishes several such books, in hardcopy and online form, including The Java Tutorial and The Java Language Specification, Third Edition.

if-else

The Java conditional statement is the `if` statement, which includes a condition (a `boolean` expression) which is evaluated, and a statement (or block of statements) which is executed if the condition is true. If the `if-else` form is used, a second statement or statement block is executed if the condition is false.

The general syntax of the `if` and `if-else` statements is as follows:

Syntax: *`if` and `if-else` statements*

```
if (condition) {
    statements
}

if (condition) {
    statements
} else {
    statements
}
```

If only one statement is to be executed conditionally, the enclosing curly braces can be omitted; however, this usage tends to be very error-prone, and is strongly discouraged.

`if` and `if-else` statements can be nested.

Example: *`if` and `if-else` statements*

```
if (age > 18) {
    vote();
}

if (distanceTo(prey) <= maximumKillDistance) {
    if (Math.random () < (distanceToPrey / maximumKillDistance)) {
        killPrey(prey);
        energy += 100;
    } else {
        energy -= 20;
    }
} else {
    energy -= 10;
}
```

switch

The Java `switch` statement is used to execute one of a set of alternative statement groups, based on the value of a selector expression. The result of the selector expression must be of type `int`, or a type which can be promoted to `int`.

This statement can be confusing at first, to those without previous experience in C/C++ or similar languages. While it might appear to be simply a more flexible conditional statement (similar to the `Select Case` statement in Visual Basic, or the `CASE` function in SQL), this is not exactly the case. In particular, evaluation of the `switch` expression does not result in selection of a unique statement block to execute; instead, there is one statement block for all of the alternatives, and the code corresponding to each alternative is marked by a `case` label; after evaluation of the selector expression, execution jumps to the statement immediately following the `case` label corresponding to the value of the selector expression. While this may seem no different than having separate statement blocks, there is one big difference: without the use of a `break` statement to exit the statement block, execution that starts with the `case` label corresponding to one alternative will proceed through and past any subsequent `case` labels, executing statements associated with those labels (and thus, with different alternatives than the one identified by the selector expression).

Here is the general form:

Syntax: *`switch` statement*

```
switch (expresson) {
case value1:
    statements
case value2:
    statements
…
case valueN:
    statements
default:
    statements
}
```

Any number (including zero) of `case` labels can be included, and the `default` label (to which execution jumps if *expresson* is not equal to any of the `case` values) is optional.

In the example below, assume that `DOG`, `CAT`, and `COYOTE` are named constants (i.e. variables declared `static final` — by convention, that type of variable is named with all uppercase letters, and with underscores between multiple words) of type `int`.

Example: *`switch` statement*

```
switch (breed) {
case DOG:
    eat();
    wander(5);
    break;
case CAT:
    eat();
    sleep(20);
    hunt(10);
    break;
case COYOTE:
    sleep(50);
    hunt(20);
    break;
default:
    eat();
    break;
}
```

At runtime, the value of `breed` will be compared with the values in `DOG`, `CAT`, and `COYOTE`. If a match is found, execution will jump to the first statement below the corresponding `case` label; if no match is found, execution will jump to the first statement below the `default` label. Note the `break` statements; these prevent execution from continuing past the group of statements associated with one alternative, into the group of statements associated with the next. When a `break` statement is encountered, the current statement block is terminated, and execution proceeds with the statement immediately following the block.

(Note that the statements following the `default` label also include a `break` statement. This is not strictly necessary, but it is strongly recommended, in case any additional `case` labels are later added to the code after the `default` label.)

`for`

The `for` statement is generally used to iterate across an array, or a set of regularly-spaced values. (It is not limited to these uses, but many of the more "creative" applications are generally better handled with a `while` statement.) The general form is as follows:

Syntax: *`for` statement*

```
for (initializer; condition; incrementor) {
    statements
}
```

At runtime, execution of the `for` statement follows this sequence:

1. The *initializer* statement is executed.
2. The *condition* is tested; if it is true:
    a. The *statements* are executed.
    b. The *incrementor* statement is executed.
    c. Execution returns to step #2

As is the case for the `if-else` statement, if *statements* contains a single statement, the curly braces can be omitted; however, that form is discouraged.

In practice, it is quite common for the *initializer* statement to be a variable declaration and assignment combination. A variable declared in that fashion will only be in scope in the `for` statement; any attempt to access it after the `for` statement completes will result in a compiler error.

The following example will print all of the elements of the `args` array (typically used to pass command line parameters to a Java application), along with explanatory text, to the standard output device.

Example: `for` *statement*

```
for (int index = 0; index < args.length; index++) {
    System.out.printf("Command line param #%d=%s\n", index + 1, args[index]);
}
```

`while`

The `while` statement is a more general-purpose looping statement than `for`: it simply repeats a statement block (or a single statement) as long as a condition (a `boolean` expression) is true. This condition is checked before the first time through the loop, and before each successive repetition.

The general syntax is as follows:

Syntax: `while` *statement*

```
while (condition) {
    statements
}
```

As before, if *statements* consists of a single statement, it is permissible (but not recommended) to leave out the curly braces.

Interestingly enough, because of the general nature of the `while` statement, *any* `for` statement can be converted into a `while` statement. We can take the general syntax of the `for` statement, and rewrite it with the `while` syntax, as follows:

Syntax: `while` *statement equivalent to* `for` *statement*

```
initializer
while (condition) {
    statements
    incrementor
}
```

Here is a simple example of the `while` statement:

Example: `while` *statement*

```
while (board.getResult() == Board.Result.GAME_CONTINUES) {
    player.move(board);
}
```

`do-while`

The `do-while` statement is identical to the `while` statement, except for one thing: the loop statement block is executed the first time, without testing the condition; thereafter, the condition is tested before each execution of the loop statement block. This is reflected in the syntax, which shows the condition at the bottom of the loop:

Syntax: *`do-while` statement*

```
do {
    statements
} while (condition)
```

High-level Code Structures and Organization

This section is intended to be a very quick introduction to certain higher-level concepts in Java — namely, classes, methods, constructors, etc. It is not a reference or a tutorial on these topics, and only a few examples are included. For more information, see The Java Tutorial, or The Java Language Specification, Third Edition.

Compilation Units

Java source code is organized into "compilation units" — i.e. source code files. Each source code file may contain a `package` declaration (optional), one or more `import` statements (optional), and one or more type declarations (classes, interfaces, and enumerations) — but only one non-nested class per file may be declared `public`. When a Java file is compiled, the compiler uses the `import` statements to resolve references to external classes and interfaces. After compilation, the classes and interfaces defined in the file are placed in the package given in the `package` declaration; other compilation units may then use this package name, along with the names of the classes and interfaces we have compiled into the package, in their own import statements.

Example: *Basic source file*

```
/*
 * Dice.java
 */
package org.challenge.nm.examples;

import java.util.Random; // Import the Random class from the java.util package.
import java.io.*;        // Import all the classes of the java.io package.

public class Dice {
    /*
     * Right now, this class does nothing - except compile cleanly, into the
     * org.challenge.nm.examples package.
     */
}
```

Note that every Java source file includes an implicit `import java.lang.*;` statement; it is not necessary (nor is it considered good programming practice) to import that package's classes explicitly.

Classes

In Java (and many other programming languages, such as C++ and Visual Basic), a class is a construct in which the data and behavior of customized objects are defined; essentially, such classes become new data types, and we can create variables (objects, or class instances) of these types, just as we create variables of the built-in "primitive" types. A simple example of such a class is the `String` class, which is part of the `java.lang` package, in the standard Java library: this class encapsulates the data required for managing strings of characters, and the typical operations which can be performed on those strings.

We might also create classes as a means to group together related operations, even if they aren't necessarily associated with a single type of object. An example of this kind of class is the `java.lang.Math` class, which includes methods which implement the standard set of basic mathematical functions (trigonometry, logarithms, etc.).

Finally, we also use special classes to define the entry points for Java programs, applets, etc. We can think of these classes as "stage managers": they set the stage, and start the show — but generally, in all but the simplest cases, the objects created from other classes are the primary performers.

A class can extend an existing class, using the `extends` keyword. the new class is referred to as a "subclass" of the class being extended (the "superclass"); it can add functionality (i.e. new methods), or modify existing functionality by overriding the methods of the superclass. Using this technique, class hierarchies can be built, starting with very generic classes at the root, and moving to more specialized subclasses.

A class may be declared `abstract`, in which case it cannot be directly instantiated; a common use of this is to declare (but not implement) functionality in a superclass, and leave the implementation to subclasses.

(Superclasses, subclasses, abstract classes, and related concepts will be addressed in another document.)

Within a class, we generally have the following elements:

- variable declarations (also called fields);
- static initializers (code which runs at the class level, the first time the class is referenced);
- constructors (special methods — having the same name as the class — which initialize objects of the class type; this operation is often called "instantiation");
- method definitions.

We can also define classes within classes (nested classes). Note that this is not the same as a subclass: a nested class is used for defining a type of object which is only used in the context of another type of object (the enclosing class); a subclass is used to extend or specialize the attributes and/or behavior of another class.

A class may be declared `public`, in which case it can be accessed by other classes (in practical terms, this usually means that other classes can create objects of the given class' type). If a class is not declared `public`, it can only be used by other classes in the same package.

The class members (variables, constructors, methods, and nested classes — but not static initializers) may be declared `public`, `protected`, or `private`, with the following implications:

- `public` members can be accessed from any other class.
- `protected` members can be accessed only by classes in the same package, or (in some cases) by subclasses of the given class.
- `private` members can only be accessed by the given class.

Class members may also be declared with `static` or default scope:

- `static` members are at the level of the entire class; they represent data or operations which are shared by all members of the class.
- Members with default scope are at the level of individual objects.

Class member variables can also be declared `final`, which means that initial values can be assigned, but never changed; thus, these are really constants, not variables. Generally, such constants are also declared as `static`, since there is little value in having each object instance maintain its own copy of a constant. (Also, by convention, `static final` variables are usually given all-uppercase names.)

Example: *Simple class*

```java
/*
 * Dice.java
 */
package org.challenge.nm.examples;

import java.util.Random;

/**
 * Class which encapsulates a set of one or more six-sided dice, and which
 * can be used to return the sum of the spots showing on the dice in a throw.
 *
 * @author      Supercomputing Challenge
 * @version     1.0, 2005-07-14
 */
public class Dice {
    /** Public constant value for the number of sides on a standard die. */
    public static final int NUMBER_OF_SIDES = 6;
    /** Private variable holding the number of dice to be thrown. */
    private int numberOfDice = 2;
    /** Each set of dice has its own private random number generator. */
    private Random rng;

    /**
     * Public constructor, which creates a set of dice, with the quantity
     * taken from the default value of numberOfDice.
     */
    public Dice() {
        rng = new Random();
    }

    /**
     * Public constructor, which creates a set of dice, with the quantity
     * specified in the numberOfDiceToThrow parameter.
     *
     * @param numberOfDiceToThrow   int value specifying the number of dice
     *                              in the set.
     */
    public Dice(int numberOfDiceToThrow) {
        this();
        numberOfDice = numberOfDiceToThrow;
    }

    /**
     * Returns the sum of random die values for the set.
     *
     * @return                      int sum of the dice throw.
     */
    public int throw() {
        int sum = 0;
        for (int index = 0; index < numberOfDice) {
            sum += (rng.nextInt(NUMBER_OF_SIDES) + 1);
        }
        return sum;
    }
}
```

Note that the above example includes comments which are formatted for use by the Javadoc tool; such comments are standard Java comments, with additional embedded information allowing for the automatic generation of technical documentation.

Interfaces

An `interface` is essentially an abstract class which includes only constants and abstract methods. In other words, an interface declares functionality, but leaves the implementation to classes which implement the interface. We can think of interfaces as contracts: a class which implements an interface must provide the functionality exactly as it is declared in the interface.

Enumerations

An `enum` is a special kind of class, which includes in its definition the definition of a static set of class instances. It is used to implement type-safe sets of enumerated constant values. Further discussion of the `enum` is beyond the scope of this document.

Packages

In general, a package is as a collection of compiled classes and interfaces, organized within a namespace. However, this organization may not necessarily be physical: collections can consist of multiple compiled `.class` files and/or multiple `.jar` archives, and the contents of a `.jar` archive can include multiple packages — in their entirety, or in part.

In practice, the namespace of a package usually maps to the hierarchical directory structure of a filesystem (which may be within a `.jar` archive). For example, if the Java compiler (or Java executive) encounters the statement `import org.nm.challenge.examples.Dice;` , it will look for the `Dice` class in the `org/nm/challenge/examples` subdirectory (if it exists) of the current directory. (If it does not find the class using the current directory as a base directory, the compiler will iterate through the directories in the current Java classpath, using each one as a base directory in searching for the `org/nm/challenge/examples` subdirectory and the `Dice` class.)

Methods

A method is a named statement block, defined within a class (it may be declared, but not implemented, in an interface or an abstract class; it may also be declared `abstract` — and thus not implemented — even in a class which isn't abstract). In general, methods make up the behavior of a class: a method may return information about an instance of the class (or the class as a whole); it may modify the data of a class instance (or static data associated with the entire class); it may use the instance or class data to invoke methods in other classes; etc.

Like a class variable, a method may be declared `public`, `private`, or `protected`, controlling the accessibility of the method from outside the class. Also like a class variable, a method may be declared `static`, in which case it is associated with the entire class, and does not automatically operate on a single instance of the class. (There are other modifiers which may be used in the declaration of methods, but they are outside the scope of this document.)

A method is declared with a list (which may be empty) of parameters; code invoking a method passes values (in the method call) which match these parameters in type (or which can be promoted to the declared parameter types), and the code of the method refers to these passed values by the names of the declared parameters.

Each method is declared with a return type; in the body of the method, a `return` statement is used to return a value (of the declared type) to the caller. Alternatively, a method may be declared with a return type of `void`, in which case no value is returned to the caller, and the `return` statement is not allowed in the method code.

An example of a method can be found in the class example, above.

Constructors

Constructors are special statement blocks which are used to instantiate objects of a class; they are generally used to initialize variables, and create/allocate resources (including arrays and other objects) which will be used over the lifetime of the object. Constructors appear similar to methods, but they have the following unique characteristics:

- The name of a constructor is the same (including case) as the class name.
- A constructor does not return a value, and no return type (not even `void`) is specified.
- A constructor cannot be invoked on an already-instantiated object.

A class may have multiple constructors, distinguished by different numbers and/or types of parameters.

An example of class constructors can be found in the class example, above.