Variables and the Primitive Types

NAMES ARE FUNDAMENTAL TO PROGRAMMING. In programs, names are used to refer to many different sorts of things. In order to use those things, a programmer must understand the rules for giving names to them and the rules for using the names to work with them. That is, the programmer must understand the syntax and the semantics of names.

According to the syntax rules of Java, the most basic names are identifiers. Identifiers can be used to name classes, variables, and subroutines. An identifier is a sequence of one or more characters. It must begin with a letter or underscore and must consist entirely of letters, digits, and underscores. ("Underscore" refers to the character '_'.) For example, here are some legal identifiers:

```
N   n   rate  x15   quite_a_long_name   HelloWorld
```

No spaces are allowed in identifiers; `HelloWorld` is a legal identifier, but "Hello World" is not. Upper case and lower case letters are considered to be different, so that `HelloWorld`, `helloworld`, `HELLOWORLD`, and `hElloWorLD` are all distinct names. Certain words are reserved for special uses in Java, and cannot be used as identifiers. These reserved words include:

```
class,    public,   static,   if,   else,    while,
```

and several dozen other words. (Remember that reserved words are **not** identifiers, since they can't be used as names for things.)

Java is actually pretty liberal about what counts as a letter or a digit. Java uses the Unicode character set, which includes thousands of characters from many different languages and different alphabets, and many of these characters count as letters or digits.

The pragmatics of naming includes style guidelines about how to choose names for things. For example, it is customary for names of classes to begin with upper case letters, while names of variables and of subroutines begin with lower case letters; you can avoid a lot of confusion by following this standard convention in your own programs. Most Java programmers do not use underscores in names, although some do use them at the beginning of the names of certain kinds of variables. When a name is made up of several words, such as `HelloWorld` or `interestRate`, it is customary to capitalize each word, except possibly the first; this is sometimes referred to as camel case, since the upper case letters in the middle of a name are supposed to look something like the humps on a camel's back.

Finally, in addition to simple identifiers, things in Java can have compound names which consist of several simple names separated by periods. (Compound names are also called qualified names.) You've already seen an example: `System.out.println`. The idea here is that things in Java can contain other things. A compound name is a kind of path to an item through one or more levels of containment. The name `System.out.println` indicates that something called "System" contains something called "out" which in turn contains something called "println".

Variables

Programs manipulate data that are stored in memory. In machine language, data can only be referred to by giving the numerical address of the location in memory where the data is stored. In a high-level language such as Java, names are used instead of numbers to refer to data. It is the job of the computer to keep track of where in memory the data is actually stored; the programmer only has to remember the name. A name used in this way—to refer to data stored in memory—is called a variable.

Variables are actually rather subtle. Properly speaking, a variable is not a name for the data itself but for a location in memory that can hold data. You should think of a variable as a container or box where you can store data that you will need to use later. The variable refers directly to the box and only indirectly to the data in the box. Since the data in the box can change, a variable can refer to different data values at different times during the execution of the program, but it always refers to the same box. Confusion can arise, especially for beginning programmers, because when a variable is used in a program in certain ways, it refers to the container, but when it is used in other ways, it refers to the data in the container. You'll see examples of both cases below.

In Java, the **only** way to get data into a variable—that is, into the box that the variable names—is with an assignment statement. An assignment statement takes the form:

```
variable = expression;
```

where **expression** represents anything that refers to or computes a data value. When the computer comes to an assignment statement in the course of executing a program, it evaluates the expression and puts the resulting data value into the variable. For example, consider the simple assignment statement

```
rate = 0.07;
```

The **variable** in this assignment statement is `rate`, and the **expression** is the number 0.07. The computer executes this assignment statement by putting the number 0.07 in the variable `rate`, replacing whatever was there before. Now, consider the following more complicated assignment statement, which might come later in the same program:

```
interest = rate * principal;
```

Here, the value of the expression "`rate * principal`" is being assigned to the variable `interest`. In the expression, the `*` is a "multiplication operator" that tells the computer to multiply `rate` times `principal`. The names `rate` and `principal` are themselves variables, and it is really the **values** stored in those variables that are to be multiplied. We see that when a variable is used in an expression, it is the value stored in the variable that matters; in this case, the variable seems to refer to the data in the box, rather than to the box itself. When the computer executes this assignment statement, it takes the **value** of `rate`, multiplies it by the **value** of `principal`, and stores the answer in the **box** referred to by `interest`. When a variable is used on the left-hand side of an assignment statement, it refers to the box that is named by the variable.

(Note, by the way, that an assignment statement is a command that is executed by the computer at a certain time. It is not a statement of fact. For example, suppose a program includes the statement "`rate = 0.07;`". If the statement "`interest = rate * principal;`" is executed later in the program, can we say that the `principal` is multiplied by 0.07? No! The value of `rate` might have been changed in the meantime by another statement. The meaning of an assignment statement is completely different from the meaning of an equation in mathematics, even though both use the symbol "=".)

Types

A variable in Java is designed to hold only one particular type of data; it can legally hold that type of data and no other. The compiler will consider it to be a syntax error if you try to violate this rule by assigning a value of the wrong type to a variable. We say that Java is a strongly typed language because it enforces this rule.

There are eight so-called primitive types built into Java. The primitive types are named byte, short, int, long, float, double, char, and boolean. The first four types hold integers (whole numbers such as 17, -38477, and 0). The four integer types are distinguished by the ranges of integers they can hold. The float and double types hold real numbers (such as 3.6 and -145.99). Again, the two real types are distinguished by their range and accuracy. A variable of type char holds a single character from the Unicode character set. And a variable of type boolean holds one of the two logical values `true` or `false`.

Any data value stored in the computer's memory must be represented as a binary number, that is as a string of zeros and ones. A single zero or one is called a bit. A string of eight bits is called a byte. Memory is usually measured in terms of bytes. Not surprisingly, the byte data type refers to a single byte of memory. A variable of type byte holds a string of eight bits, which can represent any of the integers between -128 and 127, inclusive. (There are 256 integers in that range; eight bits can represent 256—two raised to the power eight—different values.) As for the other integer types,

- short corresponds to two bytes (16 bits). Variables of type short have values in the range -32768 to 32767.
- int corresponds to four bytes (32 bits). Variables of type int have values in the range -2147483648 to 2147483647.
- long corresponds to eight bytes (64 bits). Variables of type long have values in the range -9223372036854775808 to 9223372036854775807.

The float data type is represented in four bytes of memory, using a standard method for encoding real numbers. The maximum value for a float is about 10 raised to the power 38. A float can have about 7 significant digits. (So that 32.3989231134 and 32.3989234399 would both have to be rounded off to about 32.398923 in order to be stored in a variable of type float.) A double takes up 8 bytes, can range up to about 10 to the power 308, and has about 15 significant digits. Ordinarily, you should stick to the double type for real values.

A variable of type char occupies two bytes in memory. The value of a char variable is a single character such as A, *, x, or a space character. The value can also be a special character such a tab or a carriage return or one of the many Unicode characters that come from different languages. Values of type char are closely related to integer values, since a character is actually stored as a 16-bit integer code number. In fact, we will see that chars in Java can actually be used like integers in certain situations.

It is important to remember that a primitive type value is represented using only a certain, finite number of bits. So, an int can't be an arbitrary integer; it can only be an integer in a certain finite range of values. Similarly, float and double variables can only take on certain values. They are not true real numbers in the mathematical sense. For example, the mathematical constant π can only be approximated by a value of type float or double, since it would require an infinite number of decimal places to represent it exactly. For that matter, many simple numbers such as 1/3 can only be approximated by floats and doubles.

Literals

A data value is stored in the computer as a sequence of bits. In the computer's memory, it doesn't look anything like a value written on this page. You need a way to include constant values in the programs that you write. In a program, you represent constant values as literals. A literal is something that you can type in a program to represent a value. It is a kind of name for a constant value.

For example, to type a value of type char in a program, you must surround it with a pair of single quote marks, such as `'A'`, `'*'`, or `'x'`. The character and the quote marks make up a literal of type char. Without the quotes, `A` would be an identifier and `*` would be a multiplication operator. The quotes are **not** part of the value and are not stored in the variable; they are just a convention for naming a particular character constant in a program. If you want to store the character A in a variable `ch` of type char, you could do so with the assignment statement

```
ch = 'A';
```

Certain special characters have special literals that use a backslash, `\`, as an "escape character." In particular, a tab is represented as `'\t'`, a carriage return as `'\r'`, a linefeed as `'\n'`, the single quote character as `'\''`, and the backslash itself as `'\\'`. Note that even though you type two characters between the quotes in `'\t'`, the value represented by this literal is a single tab character.

Numeric literals are a little more complicated than you might expect. Of course, there are the obvious literals such as 317 and 17.42. But there are other possibilities for expressing numbers in a Java program. First of all, real numbers can be represented in an exponential form such as 1.3e12 or 12.3737e-108. The "e12" and "e-108" represent powers of 10, so that 1.3e12 means 1.3 times 10^{12} and 12.3737e-108 means 12.3737 times 10^{-108}. This format can be used to express very large and very small numbers. Any numeric literal that contains a decimal point or exponential is a literal of type double. To make a literal of type float, you have to append an "F" or "f" to the end of the number. For example, "1.2F" stands for 1.2 considered as a value of type float. (Occasionally, you need to know this because the rules of Java say that you can't assign a value of type double to a variable of type float, so you might be confronted with a ridiculous-seeming error message if you try to do something like "`x = 1.2;`" if `x` is a variable of type float. You have to say "`x = 1.2F;`". This is one reason why I advise sticking to type double for real numbers.)

Even for integer literals, there are some complications. Ordinary integers such as 177777 and -32 are literals of type byte, short, or int, depending on their size. You can make a literal of type long by adding "L" as a suffix. For example: 17L or 728476874368L. As another complication, Java allows binary, octal (base-8), and hexadecimal (base-16) literals. I don't want to cover number bases in detail, but in case you run into them in other people's programs, it's worth knowing a few things: Octal numbers use only the digits 0 through 7. In Java, a numeric literal that begins with a 0 is interpreted as an octal number; for example, the octal literal 045 represents the number 37, not the number 45. Octal numbers are rarely used, but you need to be aware of what happens when you start a number with a zero. Hexadecimal numbers use 16 digits, the usual digits 0 through 9 and the letters A, B, C, D, E, and F. Upper case and lower case letters can be used interchangeably in this context. The letters represent the numbers 10 through 15. In Java, a hexadecimal literal begins with `0x` or `0X`, as in `0x45` or `0xFF7A`. Finally, binary literals start with `0b` or `0B` and contain only the digits 0 and 1; for example: 0b10110.

As a final complication, numeric literals can include the underscore character ("_"), which can be used to separate groups of digits. For example, the integer constant for two billion could be written 2_000_000_000, which is a good deal easier to decipher than 2000000000. There is no rule about how many digits have to be in each group. Underscores can be especially useful in long binary numbers; for example, `0b1010_1100_1011`.

I will note that hexadecimal numbers can also be used in character literals to represent arbitrary Unicode characters. A Unicode literal consists of `\u` followed by four hexadecimal digits. For example, the character literal `'\u00E9'` represents the Unicode character that is an "e" with an acute accent.

For the type boolean, there are precisely two literals: `true` and `false`. These literals are typed just as I've written them here, without quotes, but they represent values, not variables. Boolean values occur most often as the values of conditional expressions. For example,

```
rate > 0.05
```

is a boolean-valued expression that evaluates to `true` if the value of the variable `rate` is greater than 0.05, and to `false` if the value of `rate` is less than or equal to 0.05. As you'll see later, boolean-valued expressions are used extensively in control structures. Of course, boolean values can also be assigned to variables of type boolean. For example, if `test` is a variable of type boolean, then both of the following assignment statements are legal:

```
test = true;
test = rate > 0.05;
```

Strings and String Literals

Java has other types in addition to the primitive types, but all the other types represent objects rather than "primitive" data values. For the most part, we are not concerned with objects for the time being. However, there is one predefined object type that is very important: the type *String*. (*String* is a type, but not a primitive type; it is in fact the name of a class.)

A value of type *String* is a sequence of characters. You've already seen a string literal: `"Hello World!"`. The double quotes are part of the literal; they have to be typed in the program. However, they are not part of the actual *String* value, which consists of just the characters between the quotes. A string can contain any number of characters, even zero. A string with no characters is called the empty string and is represented by the literal `""`, a pair of double quote marks with nothing between them. Remember the difference between single quotes and double quotes! Single quotes are used for char literals and double quotes for *String* literals! There is a big difference between the *String* `"A"` and the char `'A'`.

Within a string literal, special characters can be represented using the backslash notation. Within this context, the double quote is itself a special character. For example, to represent the string **value**

```
I said, "Are you listening!"
```

with a linefeed at the end, you would have to type the string **literal**:

```
"I said, \"Are you listening!\"\n"
```

You can also use `\t`, `\r`, `\\`, and Unicode sequences such as `\u00E9` to represent other special characters in string literals.

Variables in Programs

A variable can be used in a program only if it has first been declared. A variable declaration statement is used to declare one or more variables and to give them names. When the computer executes a variable declaration, it sets aside memory for the variable and associates the variable's name with that memory. A simple variable declaration takes the form:

```
type-name  variable-name-or-names;
```

The **variable-name-or-names** can be a single variable name or a list of variable names separated by commas. Good programming style is to declare only one variable in a declaration statement, unless the variables are closely related in some way. For example:

```
int numberOfStudents;
String name;
double x, y;
boolean isFinished;
char firstInitial, middleInitial, lastInitial;
```

It is also good style to include a comment with each variable declaration to explain its purpose in the program, or to give other information that might be useful to a human reader. For example:

```
double principal;    // Amount of money invested.
double interestRate; // Rate as a decimal, not percentage.
```

For now, we will only use variables declared inside the `main()` subroutine of a program. Variables declared inside a subroutine are called local variables for that subroutine. They exist only inside the subroutine, while it is running, and are completely inaccessible from outside. Variable declarations can occur anywhere inside the subroutine, as long as each variable is declared before it is used in any way. Some people like to declare all the variables at the beginning of the subroutine. Others like to wait to declare a variable until it is needed. My preference: Declare important variables at the beginning of the subroutine, and use a comment to explain the purpose of each variable. Declare "utility variables" which are not important to the overall logic of the subroutine at the point in the subroutine where they are first used. Here is a simple program using some variables and assignment statements:

```
/**
 * This class implements a simple program that
 * will compute the amount of interest that is
 * earned on $17,000 invested at an interest
 * rate of 0.027 for one year.  The interest and
 * the value of the investment after one year are
 * printed to standard output.
```

```
 */

public class Interest {

   public static void main(String[] args) {

       /* Declare the variables. */

       double principal;     // The value of the investment.
       double rate;          // The annual interest rate.
       double interest;      // Interest earned in one year.

       /* Do the computations. */

       principal = 17000;
       rate = 0.027;
       interest = principal * rate;   // Compute the interest.

       principal = principal + interest;
             // Compute value of investment after one year, with interest.
             // (Note: The new value replaces the old value of principal.)

       /* Output the results. */

       System.out.print("The interest earned is $");
       System.out.println(interest);
       System.out.print("The value of the investment after one year is $");
       System.out.println(principal);

   } // end of main()

} // end of class Interest
```

This program uses several subroutine call statements to display information to the user of the program. Two different subroutines are used: `System.out.print` and `System.out.println`. The difference between these is that `System.out.println` adds a linefeed after the end of the information that it displays, while `System.out.print` does not. Thus, the value of `interest`, which is displayed by the subroutine call "`System.out.println(interest);`", follows on the same line as the string displayed by the previous `System.out.print` statement. Note that the value to be displayed by `System.out.print` or `System.out.println` is provided in parentheses after the subroutine name. This value is called a parameter to the subroutine. A parameter provides a subroutine with information it needs to perform its task. In a subroutine call statement, any parameters are listed in parentheses after the subroutine name. Not all subroutines have parameters. If there are no parameters in a subroutine call statement, the subroutine name must be followed by an empty pair of parentheses.