

Elements of the Java Language

(CISS-110 William Jojo)

Components of the Language

The Java language is actually rather small when you look at the parts that make up the core aspects of the language. First we will discuss the rules and the meaning of the language. Like any foreign language there exist rules that govern how the language may be written and there is understanding or meaning that is derived from how the words are arranged. For example, when we say, "The car belongs to Bill." We know that the subject, car, and the object, Bill, are clearly arranged such that we can derive the meaning of Bill owning a car. We can derive further meaning if there were surrounding sentences of a paragraph in which this original sentence also belongs. Perhaps we would learn that Bill's car was large or small, if it was gas or diesel, or even the color.

The point is every language has *syntax rules* that govern how the language may be written. The *semantic rules* help us to determine meaning. Java is broken down into three basic parts, or *tokens*, for writing the language. These tokens are *symbols*, *reserved words* and *identifiers*.

Symbols are simply the characters we also using in our writing regardless of the subject. So punctuation, arithmetic and relational symbols are used throughout the program. *Table 1* lists some categories of symbols. Note that regardless of the number of characters used to make the symbol, it is only *one* symbol.

Punctuation	.	,	;	“	!
Arithmetic	+	-	*	/	%
Relational	>	<=	!=	<	==

Table 1: Some symbols used in Java.

Reserved words, also called *keywords*, are a standard set of words that are set aside specifically for the Java language. You may not reuse these words as identifiers. If you do, the compiler will let you know that the use of the reserved word is inappropriate by signaling a *syntax error* when you compile your program. When using TextPad, you will also know that the word is reserved when the word you are using turns blue. Some reserved words are listed in Table 2.

for	while	do	if	else
switch	case	int	float	char
throws	return	implements	double	void

Table 2: A partial list of reserved words.

A complete list of reserved words can be found here:

<http://java.sun.com/docs/books/tutorial/java/nutsandbolts/keywords.html>

Identifiers are used to label *things*. These things are often variables, methods or constants. Like the rest of the Java language these identifiers are case sensitive. The identifiers you create may contain letters, digits, the underscore character (`_`) and the dollar sign (`$`). Your identifiers, however, must not begin with a digit. They can begin with any of the other characters. It is conventional to limit the use of the underscore and dollar sign to very special purpose identifiers, or to use the underscore where spaces would ordinarily appear in normal language. A few acceptable identifiers are shown in *Table 3*.

sum	totalWords	Sum
Total_Words	Word_Count\$	myLittleVariable

Table 3: Examples of unique and valid identifiers.

Data Types

There are three primitive data types: *integral*, *floating-point* and *boolean*. You may already know what these types are just from their names. Integral types are whole number quantities, floating-point allows for precision by representing a fractional portion and boolean indicates truth value.

There are five **integral** types and their properties are displayed in *Table 4*.

Data Type	Range of values	Size in bytes (bits)
char	0 to 65535	2 (16)
byte	-128 to 127	1 (8)
short	-32768 to 32767	2 (16)
int	-2147483648 to 2147483647	4 (32)
long	-92233720368454775808 to 92233720368454775807	8 (64)

Table 4: Values and sizes of integral data types.

The only integral data type that has special written properties is `char`. The values that represent the this type use single quotes to denote that value as a *char constant*. A few examples of these constants are `'+'`, `'A'`, `'('` and `'n'`. Without the single quotes the plus symbol may be mistaken for an arithmetic operator, or the open parenthesis as part of the Java

language, or the letters mistaken for identifiers.

Characters are Unicode and JDK 5.0 follows the Unicode 4.0 standard <http://www.unicode.org/versions/Unicode4.0.0/>. Characters are really numeric quantities and the characters we use with single quotes are simply human consumable forms so we don't need to memorize the entire character set. The table of values to characters is standardized and represents a collating sequence for several character ranges.

Since part of Unicode follows the original ASCII collating sequence, three starting points need to be mentioned. The digit '0' (48), capital 'A' (65) and lower case 'a' (97). Following '0' is the digit '1', then '2' and so on. The same goes for the two starting points for the alphabet. If 'A' is 65, then 'B' is 66 and 'C' is 67.

Data Type	Range of Values	Significant Digits	Size in bytes (bits)
float	-3.4E+38 to 3.4E+38	6 or 7	4 (32)
double	-1.7E+308 to 1.7E+308	15	8 (64)

Table 5: Properties of floating-point data types.

There are two **floating-point** data types and their properties are displayed in Table 5.

The **boolean** data type has only two possible values. They are **true** and **false**.

Arithmetic Operators and Precedence

There are five arithmetic operators as shown in Table 6. These operators maybe used to construct *expressions*. An expression is something simple like **3+2** or **8/5**. So our expressions are made up of operators and *operands*. Operands are the values the operators work with. The previous examples are known as integer expressions. They are integer expressions because they have two integers as their operands and the result will be integer.

+	addition
-	subtraction
*	multiplication
/	division
%	modulus

Table 6: Arithmetic operators.

The operators in Table 6 are also known as *binary* operators. Not because they work with binary numbers, but because they require two operands. You are probably familiar with the *unary* operators – and +. These can be seen when talking about the value **-3**. See how the minus sign has only one operand? It says that the value of 3 is negative as opposed to positive.

Why is it necessary to talk about unary and binary operators? Consider the expression **4 - -3**. Some would look at this and say, “Syntax error!” Quite the contrary. It is four minus negative three. The result is 7.

Operator precedence is an issue when multiple operators are mixed within the same expression. Operator precedence allows us to know the order in which the operators will be applied to the values. Of course, we can alter the default precedence rules by simply using parentheses. The operators `*`, `/` and `%` have a higher order of precedence than `+` and `-`. The higher precedence operators are done first. When there are many arithmetic operators of the same precedence, the operators will be evaluated from **left to right**.

Expressions With Mixed Types and Type Conversion

There will come a time when you will have values of varying types from varying sources. When this happens, we have to keep some simple rules in mind for how things will be evaluated and what the resultant type will be.

We can apply the rules to each operator such that:

1. If the operands of an operator are the same type, then the result is that type (`int` plus `int` equals `int`).
2. If the operands of an operator are of different types (`int` and `float`, for example), then the result is the type with higher precision (`int` plus `float` equals `float`).

Rule number 2 is also known as *implicit type coercion*.

If we need to change a type temporarily for a specific purpose, we can also use a *cast* or *explicit type coercion*. The cast acts as a temporary type conversion. First the *simplest* expression to the right of the cast is evaluated then the cast applied. The next few examples show how to perform a cast.

<code>(int)7.9</code>	yields 7 since the <code>double</code> is converted to an <code>int</code> . (The value 7.9 is considered to be a <code>double</code> constant.)
<code>(float)25/6</code>	yields 4.1666665 since integer 25 is converted to <code>float</code> and divided by the integer 6 (see rule 2, above).
<code>(float)(25/6)</code>	yields 4.0 since the two integers are divided which yields no precision and the 4 is converted to 4.0 (see rule 1, above).

String Class

The `String` class is a complex data type that will be used throughout this course in addition to the primitive data types mentioned earlier. Strings represent data that a single char is incapable of representing such as a person's name or street address of the name of a country.

Strings are exceedingly easy to represent. They use double quotes to denote the string value, much the same way as single quotes are used to denote a char constant value.

```
"Bill"  
"Route 2"  
"United States"
```

The empty string or *null* string is represented as empty double quotes (`""`) and has a length of zero.

The characters that make up a string are individually addressable. This means for the string "Bill":

- 'B' is at position 0
- 'i' is at position 1
- 'l' is at position 2
- and '\0' is at position 3

The length of the string is 4 since there is a total of 4 characters that make up that name.

Named Constants and Variables

Ultimately the data that the Java program is intended to work with needs to be held somewhere so that it can be referenced on demand. These values are kept in memory. Further, these location in memory will have names assigned so that we are not responsible for remembering where in memory the data is stored.

Named constants allow you to use a word to represent a value. Consider the value Pi which is estimated to 3.14159. This value can easily be represented in a `float`, so consider the following constant **declaration**.

```
final double PI = 3.14159;
```

A declaration simply announces to the compiler, "I plan on using this named location in memory for the following purpose: PI is a constant representing 3.14159." It is known to be a constant because of the reserved word `final` that has been applied to the declaration.

The declaration is read aloud as, "PI is a double whose final value is 3.14159." Attempts to

change a constant will yield an error from the compiler.

Now, the opposite of a constant is a value that changes or a **variable**. Consider the following variable declaration.

```
int sum;
```

This says, “sum is a variable of type `int`.” By default, there is no value assigned to a variable. Its starting value is considered to be undefined and you should never assume it to be zero. So perhaps we really should have said, “sum is an integer variable whose starting value is unknown.”

If you want a guarantee use the following:

```
int sum = 0;
```

This says, “sum is an integer variable whose starting value is zero.” The equal sign is also known as the **assignment operator**. The assignment operator will get quite a workout as values are always changing in your variables. Instead of saying, “sum equals zero,” which implies some sort of test for equality, perhaps say, “sum gets the value of zero.” This more closely represents what is really happening, that is, the current value of sum is thrown out and a new value, zero, is put in its place.

```
public class variables
{
    public static void main (String[] args)
    {
        int    test1, test2, test3, sum;
        double    average;

        test1 = 90;
        test2 = 85;
        test3 = 87;

        sum = test1 + test2 + test3;

        average = sum / 3.0;

        System.out.println("The average is " + average);
    }
}
```

Example 1: Program to calculate an average using variables.

Consider the program in *Example 1*. The output from this program is:

```
The average is 87.33333333333333
```

There are several things to note about this program.

1. Several variables of type `int` are declared on the same line by using commas to separate the names.
2. The value contained in `sum` is divided by the `double` constant `3.0` to force an implied coercion of the result to the `double` type which is then stored in `average`.
3. The value contained in `average` is appended to the end of the string constant in the output statement. (There are no quotes around `average`)
4. The plus sign (+) is used to join the two parts of the output. It is known as the *concatenation* operator.
5. The variable `sum` could be eliminated. Consider for a moment how you would rewrite the average calculation if you eliminated `sum`.

```
import java.util.*;

public class userInput
{
    static Scanner kb = new Scanner(System.in);

    public static void main(String[] args)
    {

        int test1, test2, test3;
        double average;

        System.out.print("Enter test 1: ");
        test1 = kb.nextInt();

        System.out.print("Enter test 2: ");
        test2 = kb.nextInt();

        System.out.print("Enter test 3: ");
        test3 = kb.nextInt();

        average = (test1 + test2 + test3) / 3.0;
        System.out.println("The average is " + average + ".");
    }
}
```

Example 2: Program modified to read values from user.

Basic Input/Output

`System.out` is known as the **standard output stream object** and `println()` (read print-line) is a method of that object. For our purposes we can think of standard output as the screen. The `println()` method always moves the cursor to the beginning of the next line.

It is important to view the chronology of the program in *Example 1*. First the variables are declared, then values are assigned for the three test grades. The sum is then calculated and finally an average is determined and displayed. It would be silly to put the output statement first; what could we possibly display? Keep in mind the steps necessary to solve a problem. This example is quite rudimentary and you can expect your programming projects to be more difficult than this, but the premise is the same: determine what absolutely must happen next, then what must occur after that.

Consider what you would do if you had to read values from the user instead of assigning them directly using constants. The `Scanner` class is a simple way to read data from some input stream and is only available in Java 5.0 and higher. In particular, we will use `System.in` which is, you guessed it, the **standard input stream object**. Since we can view the standard input stream as the keyboard, we will declare our `Scanner` class object as follows:

```
static Scanner kb = new Scanner(System.in);
```

This declaration says, “`kb` is a static reference variable of the `Scanner` class that gets the value of a new `Scanner` class object associated with `System.in`”. Wow, that’s a mouthful! An experienced Java programmer might say it as, “`kb` is a `Scanner` reference to the standard input stream.” Either way, we intend to use `kb` to read input from the user. We are also setting the stage for sharing the same `Scanner` object with other user-defined methods later in the semester.

The program in *Example 2* is fundamentally the same as *Example 1* with test data read from the user with the `Scanner` class. A sample run on a Unix host, with the user-provided values in bold, is shown here:

```
$ javac userInput.java
$ java userInput
Enter test 1: 90
Enter test 2: 85
Enter test 3: 87
The average is 87.33333333333333.
$
```

There are some items to note about this program:

1. The variable `sum` was eliminated.
2. The `nextInt()` method of `kb` returns the input in the form of an integer and stores the value in the variable to the left of the assignment operator.
3. There is an alternation of prompt, read, prompt, read. This is necessary to inform the user what to do next.
4. An additional method, `print()`, is used to hold the cursor on the same line as the prompt. Recall that `println()` moves the cursor to the next line.

A very important note about the program in *Example 2* is the use of the import statement. In order to use the `Scanner` class, we must import the class from the *package* that defines the class. The package `java.util` is really where the `Scanner` class is defined, but it is common to require more than one class from `java.util` so although we could import `java.util.Scanner`, we will simply use `java.util.*` and import all that we reference.

Another package (and its associated classes and methods) is automatically imported for every Java class you write. It is known as `java.lang`.

Integers are not the only things we would want to read. What about characters, words, lines of text and precision values? Additional methods assist with this. Some common `Scanner` class methods are shown in *Table 7*.

Scanner Method	Purpose
<code>nextInt()</code>	Read next value as an integer.
<code>nextDouble()</code>	Read next value as a double.
<code>next()</code>	Read next word.
<code>nextLine()</code>	Read next complete line up to newline.
<code>next().charAt(0)</code>	Read next character. (The <code>next()</code> method returns a <code>String</code> object which then has the <code>charAt()</code> method applied to that <code>String</code> object to obtain the first character).

Table 7: A sample of Scanner methods.

So how do these methods know where one piece of data ends and the next one begins? Well, certainly because of our use of the *Enter* key. More to the point, these methods break on *whitespace*. This means the methods will end identification of the requested data type when a space, tab, carriage return or newline is discovered during processing with the exception of `nextLine()` which only breaks on the newline character. This is not particularly useful knowledge at the moment since we will be using the *Enter* key for our input handling. This will be useful, however, when file handling becomes necessary.

If you recall from our `String` class discussion, every character has a position. The `charAt()` method retrieves the first character returned by the `next()` method. This is possible because `charAt()` is a `String` method and `next()` returns a `String`. This is also necessary to get a single character from the user since no method exists in the `Scanner` class that does this naturally.

Escape

It is often necessary to print characters that are otherwise reserved for special use. For example, if double quotes are used to surround the string of characters to be displayed, then how does one print double quotes? What about representing a single quote as a character or using singles quotes in a string? What about printing multiple blank lines? Well, all of these issues are easily resolved with a series of *escape sequences*. Escape sequences are represented by a leading backslash (`\`) which changes the meaning of the very next character. Consider the following:

```
char squote;
String fiveBlankLines;

squote = '\'';
fiveBlankLines = "\n\n\n\n\n";
```

The character sequence `\'` escapes the single quote character making is simply a single quote and not part of the delimiting quotes for a character constant. The assignment statement actually stores a single quote character into `squote`. A similar concept is applied to the sequence `\n` which makes the `n` special – this `n` represents the newline character. A list of escape sequences is shown in *Table 8*.

Escape Sequence	Purpose
<code>\n</code>	Newline. Moves cursor to beginning of next line.
<code>\t</code>	Tab. Moves cursor to next tab stop.
<code>\\</code>	Backslash. Represents a backslash.
<code>\"</code>	Double quote. Represents a double quote within a string or character constant.
<code>\'</code>	Single quote. Represents a single quote withing a string or character constant.
<code>\b</code>	Backspace. Move cursor back one character.
<code>\r</code>	Return. Move cursor to beginning of current line.

Table 8: Common escape sequences.

Increment, Decrement, Concatenation and Comments

Some additional aspects of the Java language that require mention but not a great deal of detail are mentioned here:

The increment (++) and decrement (--) operators are unary operators similar to unary plus and unary minus. One important difference is ++ and -- may appear on either side of the expression element. This defines pre- or post- for the operator.

Consider the following:

```
int x=4, y=6, z, a;

x++;
--y;
z = x++;
a = ++y;
```

The variable `x` starts at 4 and `y` starts at 6. A post-increment is applied to `x` making its value 5. A pre-decrement of `y` makes its value 5. Since these statements only have single variable expressions, it really doesn't matter if they are pre or post – in this case.

Now look at the assignment statements for `z` and `a`. The variable `z` will get the value of a post-increment of `x`. This means `z` will get the value of `x` **before** it is incremented. Therefore `z` will be 5 and `x` will be 6. The variable `a` will get the value of `y` **after** it is incremented. Therefore `a` will be 6 and `y` will be 6.

Recall from our code examples where the plus (+) operator was used inside the `println()` method. Again, this is the *concatenation* operator for strings. This is also classified as an *overloaded operator* as the plus operator is also used for the sum of numeric quantities. Further examples of concatenation are shown here:

```
String name, greeting, statement;

name = "Bill";
greeting = "Good morning";

statement = greeting + ", " + name + ".";
```

The concatenation operator is use to take the parts of the verbal statement and put them together in the `String` object `statement` as “Good morning, Bill.” complete with punctuation.

Lastly, comments are the favorite tool of any good programmer to document their program, especially complex or non-intuitive solutions. Comments come in two forms – the `//` form and the `/* */` form.

```
int x;    //comment to end of line describing purpose of x

/*
This comment is allowed to span lines.
It is quite useful when the necessary programming note
requires much more detail than a single line will allow.
*/

x = 10;
```

As you can see, the `//` comment is only a comment until the end of the line. and the `/*...*/` comment may span multiple lines. Note that there are no spaces between any of the comment characters.

The Dot Operator

You may have noticed at this point that there is a period or *dot* between class names or reference variables and the methods we wish to access such as in:

```
System.out.println("Hello!");
```

and

```
x = kb.nextInt();
```

The dot between `kb` and `nextInt()` is known as the *member access operator*. This operator indicates that we wish to access or *call* the `nextInt()` method which is a member of `kb`. More to the point, since `kb` is a `Scanner` class object, we are actually accessing the `nextInt()` method of the `Scanner` class.

The dot is required otherwise we would have code that looks like:

```
Systemoutprintln("Hello!");
```

and

```
x = kbnextInt();
```

Both of these are completely legal identifiers and could represent actual methods created by the user, but certainly not any predefined methods.