4 Data Types, Variables and Operators

This fairly long section must be read and understood before you proceed. It also contains a considerable amount of syntax information that you will want to refer to throughout the course.

4.1 Introduction

Computer programs are based on the manipulation of different types of data, held in *variables*, by using various types of arithmetic and logical *operations*. This section defines the various types of data and the basic operations available to you in JAVA.

4.2 Data Types

The basic data types are:

a) Integer: Positive or negative whole numbers normally specified in *Decimal* for example `10`, `-9`, `854`.

b) Boolean: being either *true* or *false*.

c) Floating Point: Positive or negative rational numbers held in mantissa plus exponent form, ($0.nnnnnnnn \times 10^{mm}$). These are specified with a decimal point or optional exponent, for example,

`10.0`	=	10.0
`-3.14`	=	-3.14
`4.26e8`	=	4×10^{8}
`-10.335e-15`	=	-1.0335×10^{-14}

d) Character: Upper and lower case alphabet, the ten digits, symbols such as $+ - \%\&$ etc and a range of non printable control characters such as TAB, LINEFEED, BELL etc. These are specified by the character in single quotes, for example:

Character	Meaning
`'0'`	The letter '0'
`'1'`	The letter '1'
`'A'`	The letter 'A'
`'b'`	The letter 'b'

In addition there are a range of characters that are either not directly available from the keyboard or are used for other things. These are represented by pairs of symbols but are actually *single* characters. These are:

Character	Action
`\0`	Null
`\b`	Backspace
`\t`	TAB (Horizontal Tab)
`\n`	Linefeed (New line)
`\v`	VT (Vertical Tab)
`\f`	Formfeed (New page)
`\r`	Carriage Return (Beginning of line)
`\"`	Double quote
`\'`	Single quote

Characters can also be specified by their (Octal) numerical value, detail in advanced books.

e) Strings: Which are lists of *characters* surrounded by "double quotes", for example

```
"Hello World !"
```

is a *String* containing 13 characters. Note the `" "` are not part of the *String*.

4.3 Variables

To manipulate the above data types and be able to perform arithmetic or logical operations they must be stored as *variables* in computer memory. A *variable* is a location in computer memory that your program can *read* and *write* to. Each *variable* has **three** properties, these being:

1. its *name*, which you choose,
2. its *value*, which you set,
3. its *address* (or location) in the computer memory, which is chosen for you.

In JAVA the most basic *variable* types are:

`boolean` which can take the values `true` and `false` only.

`char` 16-bit Unicode character. (not used in this course).

`int` Used to hold an integer value. In this implementation the `int` is 32-bits long and can hold integer values from -2147483648 to 2147483647.

`double` Used to hold a *floating point* number in mantissa plus exponent form, ($0.nnnnnnnn \times 10^{mm}$). The `double` is 64-bits long. The mantissa is 14 decimal places, and maximum and minimum is $\pm 1.79769 \times 10^{308}$.

In addition to normal numerical values, `double` has three special non-numerical values to represent $\pm\infty$ and "Not-a-Number" normally used for error conditions. See later in this section.

`String` Used to hold a list of characters. Technically a `String` is a list of `char`s and is not really a separate variable type but is used so extensively it is easier, at this point, to consider it as a separate variable type.

There are additional basic variable types, being `byte`, `short`, `long` and `float` which shall not be used in this course.

Clearly the use of a single *variable* is a bit limiting; we will see how to group these into *arrays* later.

4.4 Variable Names and Declaration

Before we can use a *variable* it must be *declared*. This both defines the *name* **and** allocates a memory location in which its *value* can be stored. JAVA has a very flexible set of rules for the variable names, these being:

1. Any combination of letters[1] and numbers including _ (underscore).
2. Must start with a letter. (By convention a lower case letter).
3. Upper and lower case letters *are* different.
4. Must not clash with any of the JAVA *keywords* in Table 1 which have special pre-defined meanings.

abstract	default	if	private	this
boolean	do	implements	protected	throw
break	double	import	public	throws
byte	else	instanceof	return	transient
case	extends	int	short	try
catch	final	interface	static	void
char	finally	long	strictfp	volatile
class	float	native	super	while
const	for	new	switch	
continue	goto	package	synchronized	

Table 1: Table of keywords

When you are deciding on the *name* of a variable make the name describe its purpose. This makes the program much easier to read. Typical declarations may be:

```
int counter;            // Declare an integer
double errorValue;      // Declare a double
String myName;          // Declares a string
```

You will learn more about declaration later.

4.5 Assignment

Once we have declared a *variable* we can set its *value* which is done with the very misleading "=" sign. For example if we have:

```
double xValue;
xValue = 5.34e-6;
```

This means "`xValue`" is the *name* of a `double` variable, and we have set its *value* to be 5.34×10^{-6}.

Note that each statement in the program must be terminated with a ";".

The assignment can contain arithmetic expressions, for example

```
double xValue;
xValue = 10.0 + 13.8 + 56.6;
```

[1]Due to the use of the Unicode character the definition of 'letters' and 'numbers' is very wide. Best stick to normal keyboard characters, the use of Japanese, Tibetan, Hebrew etc. can result in confusion and problems with printing and viewing !

which sets the *value* of `xValue` to 80.4. See more details of this below.

Multiple variables can be defined as for example;

```
double xValue, yValue;
xValue = 10.0;
yValue = xValue + 25.0;
```

declares two `double` variables. The *value* of `xValue` is first set to 10.0, then the *value* of `yValue` is set to the *value* of `xValue` plus 25.0, being 35.0.

Note: The order of the statements is essential. Also subsequent changes to the value of "`xValue`*" do* **not** *change the value of "*`yValue`*".*

Or multiple occurrences of the same variable, for example:

```
double xValue;
xValue = 10.0;
xValue = xValue + 25.0;
```

which declares one `double` variable `xValue`. The *value* of `xValue` is first set to 10.0. The second statement then says,

1. Take the current *value* of `xValue`. (10.0)
2. Add 25.0, (to get 35.0)
3. Set the new *value* of `xValue` to be 35.0,

so the "old" *value* of `xvalue` has been lost.

This idea is somewhat confusing on first encounter, but remember the "=" sign means "*set value to*" and is **not** the algebraic equals.

4.6 Basic Arithmetic

The basic arithmetic operators are,

Symbol	Operator
`*`	Multiply
`/`	Divide
`%`	Modulus (integer only)
`+`	Plus
`-`	Minus
`++`	Increment (integer only)
`--`	Decrement (integer only)

which are evaluated in "left-to-right" and in the normal order of precedence (Multiply/divide (and modulus) before addition/subtraction), with round () brackets being used to alter this precedence if required.

Floating Point Arithmetic

When all the variables (or constants) are *floating point*, typically `double`, then the arithmetic obeys the normal arithmetic rules. For example;

$$y = \frac{2(3x+6)}{(4.3x^2+6.9)}$$

is coded as:

```
double x,y;
x = 10.0;
y = 2.0*(3.0*x + 6.0)/(4.3*x*x + 6.9);
```

Note there is no "power" operator. It is implemented as the function `Math.pow` *detailed in the next section.*

In arithmetic expressions SPACES, TAB(S) and NEWLINEs are ignored but the whole expression *must* be terminated by "`;`".

It is good practice to space out arithmetic expressions with SPACE(s) and NEWLINE(s). Also *extra* brackets `( )` *are* permitted. This makes them *much* easier to read, and consequently *much* easier to spot a mistake. A few extra keystrokes while writing a complex arithmetic expression takes about 3 or 4 seconds. If this results in you getting it *right* it can save *hours* of frustration and mental anguish!

Integer Arithmetic

The basic rules are as for *floating point* except for three additional operators and divide "/" being different.

Divide: Integer divide gives the "integer part" of the division only, so that :

`4/3 = 1` and `4/5 = 0`

Modulus: (`%`). This gives the "remainder" following integer division, so that

`4%3 = 1` and `4%5 = 4`

Increment/Decrement: (`++` and `--`) add and subtract 1 (one) from an integer. Further details of these will be discussed later in the course. These operators are mainly used in loops to make programs more efficient but can (unfortunately) result in unreadable code.

Mixed Integer/Floating Point Arithmetic and Casting

In JAVA conversion from one type of data type to another requires thought, and the idea of "casting". At this stage we will only consider conversion between `int`s and `double`s.

The "cast" operators are

`(double)` ⇒ convert to `double`
`(int)` ⇒ convert to `int`

where the `()` are required.

Integer to Double Conversion: is the simplest with,

```
int iValue = 12;
double xValue;
xValue = (double)iValue;
```

which will result in `xValue` being set to value `12.0` exactly as expected.

This casting can also be used in arithmetic expressions so that,

```
double xValue;
int iValue = 10, jValue = 3;
xValue = (double)iValue/(double)jValue;
```

this will result in a *floating point* divide and so `xValue = 3.333..` as expected since it will convert value of `iValue` and `jValue` to `double` *before* the division.

However if you wrote the above example with,

```
double xValue;
int iValue = 10, jValue = 3;
xValue = (double)(iValue/jValue);
```

then the value of `xValue` would be `3.0`, since we have an integer division, the result of which will be converted to a `double`.

Aside: In many occasions the `(double)` *cast is optional and will occur automatically, however it is "good programming practice" to include it, it also help to make the code easier to read and understand.*

Double to Integer Conversion: Clearly since `int` can only represent whole number we have to loose the decimal fractions. Thus

```
double xValue = 5.674;
int iValue;
iValue = (int)xValue;
```

will result `iValue` being set to `5`. Note: the value is truncated and **not** rounded. See the section on Mathematical Function if you want to round.

Since we "loosing something" in this conversion the `(int)` cast is *never* optional. You will get a compile time error if you miss it out.

4.7 Arithmetic Assignment Operators

There are many instances when we want to perform an arithmetic operation on a *variable* and overwrite its *value* with the new calculated *value*, for example to add 10 to an `int` called `val` we can use,

```
val = val + 10;
```

However this construction occurs so frequently there are assignment versions of the arithmetic operators allowing this to be written as

```
val += 10;
```

which means:

1. take the *value* stored in `val`
2. add 10 to it to obtain a new *value*,
3. set `val` to this new *value*.

There are also assignment versions of the other operators, these being

Symbol	Operator
`*=`	Multiply by
`/=`	Divide by
`%=`	Modulus of (integer only)
`+=`	Add to
`-=`	Subtract from

These assignment operators should be used with care since they can easily result in your code being difficult to understand.

4.8 Strings

This short section on `Strings` is sufficient for the initial part of the course. There is a more detailed discussion later. Strings are lists of characters that can be declared, set to a value and "added" to. Note: unlike most other computer languages `Strings` in JAVA are **not** of fixed length.

To declare a `String` simply use

```
String aName = "Fred Smith";
```

which creates a `String` of 10 characters including the Space, but excluding the `"` which mark the beginning and end.

The "addition" of `Strings` "adds" or concatenates one on to the end to the end of the other exactly as expected, so for example we can re-write the basic "Hello World!" program as:

```
//
//           Hello World by adding strings.
//
public class StringExample{
    public static void main(String args[])
     {
        String firstWord = "Hello";
        String secondWord = "World";
        String sentence;

        sentence = firstWord + " " + secondWord + "!";
```

```
        System.out.println(sentence);
        System.exit(0);
    }
}
```

Here `sentence` is a `String` made up from the "addition" of **four** `String`s.

Conversion of other data types to Strings

All of the other basic data types, `boolean`, `int`, `double` and `char` can also be converted to `String`s by simple "adding" them to an existing `String`. For the data types used in this course we get.

Type	String format
`int`	Decimal value
`double`	Decimal value in fixed point format
`boolean`	Either "true" or "false"
`char`	The printable character

This gives us our first method for output of a value since the following code fragment

```
int iValue = 256;
String answer = "The value of iValue is : " + iValue;
System.out.println(answer);
```

will output `The value of iValue is :  256`.

This method of "adding" variable to `String`s is the main method of formatted output in JAVA. It works well for all but `double` where we tend to get huge numbers of unwanted significant figures. We will see how to control that in the next section.

Examples

Source code for the following on-line examples are available,

- Floating point arithmetic in DoubleExample.
- Use of brackets in floating point expressions in BracketExample.
- Use of integer arithmetic in IntegerExample.
- Use of casting between data types in CastingExample.
- Simple of of Strings in StringExample.

What Next?

You will need to read the next chapter about basic input/output before you are ready to write a real program.