BCA PART- III

PAPER – XIX

TOPIC: CONSTANTS, VARIABLES, AND DATA TYPES

PREPARED BY: DR. KIRAN PANDEY

(SCHOOL OF COMPUTER SCIENCE)

EMAIL ID: kiranpandey.nou@gmail.com

Introduction

A programming language is designed to process certain kinds of *data* consisting of numbers, characters and strings and to provide useful output known as *information*. The task of processing data is accomplished by executing a sequence of instructions constituting a *program*. The instructions are formed using certain symbols and words according to some rigid rules known as *syntax rules* (or *grammar*). Every program instruction must conform precisely to the syntax rules of the language.

Like any other language, Java has its own vocabulary and grammar.

Constants

Constants in Java refer to fixed values that do not change during the execution of a program. Java supports several types of constants as illustrated in Fig. 2.1.

Integer

An *integer* constant refers to a sequence of digits. There are three types of integers, namely, *decimal* integer, *octal* integer and *hexadecimal* integer.

Decimal integers consist of set of digits, 0 through 9, preceded by an optional minus sign. Valid examples of decimal integer constants are :

```
123        - 321      0          654321
```

Embedded spaces, commas, and non-digit characters are not permitted between digits. For example,

```
15 750     20.000          $1000
```

are illegal numbers.

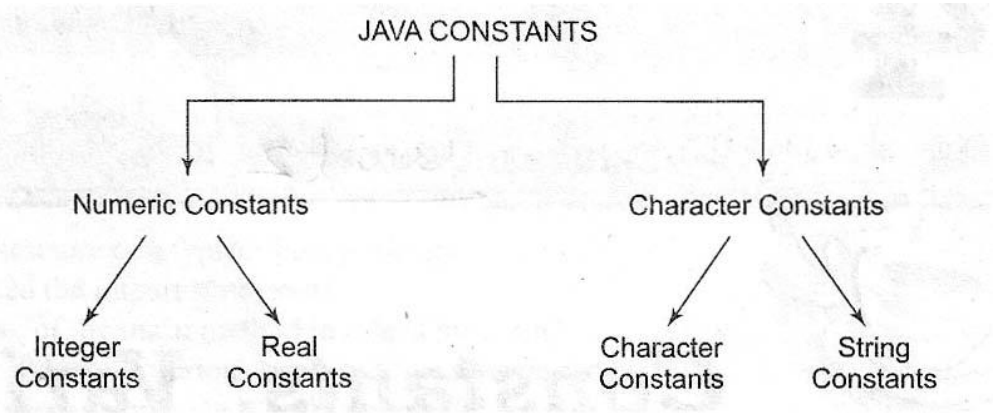

Fig. 2.1 Java Constants

An *octal* integer constant consists of any combination of digits from the set 0 through 7, with a leading 0. Some examples of octal integer are:

```
037        0        0435        0551
```

A sequence of digits preceded by 0x or 0X is considered is *hexadecimal* integer (hex integer). They may also include alphabets A through F or a through f. A letter A trough F represents the numbers through 15. Following are the examples of valid hex integers.

```
0X2        0X9F        0×bcd        0x
```

We rarely use octal and hexadecimal numbers in programming.

Real Constants

Integer numbers are inadequate to represent quantities that vary continuously, such as distances, heights, temperatures, prices, and so on. These quantities are represented by numbers containing fractional parts like 17.548. Such numbers are called *real* (or *floating point*) constants. Further examples of real constants are:

0.0083 –0.75 435.36

These numbers are shown in *decimal notation,* having a whole number followed by a decimal point and the fractional part, which is an integer. It is possible that the number may not have digits before the decimal point or digits after the decimal point. That is,

215. .95 –.71

are all valid real numbers.

A real number may also be expressed in *exponential* (or *scientific*) *notation*. For example, the value 215.65 may be written as 2.1565e2 in exponential notation. e2 means multiply by 10^2. The general form is:

mantissa e exponent

The *mantissa* is either a real number expressed in *decimal notation* or an integer. The *exponent* is an integer with an optional *plus* or *minus* sign. The letter e separating the mantissa and the exponent be written in either lowercase or uppercase. Since the exponent causes the decimal point to "float", this notation is said to represent a real number in *floating point form*. Examples of legal floating point constant are:

0.65e4 12e-2 1.5e+5 3.18E3 –1.2E-1

Embedded white (blank) space is not allowed, in any numeric constant.

Exponential notation is useful for representing numbers that are either very large or very small in magnitude. For example, 7500000000 may be written as 7.5E9 or 75E8. Similarly, –0.000000368 is equivalent to –3.68E-7.

A floating point constant may thus comprise four parts:

- a whole number
- a decimal point
- a fractional part
- an exponent

Single Character Constants

A single character constant (or simply character constant) contains a single character enclosed within a pair of single quote marks. Examples of character constants are:

'5' 'X' ';' ' '

Note that the character constant '5' is not the same as the *number* 5. The last constant is a blank space.

String Constants

A string constant is a sequence of characters enclosed between quotes. The characters may be alphabets, digit, special characters and blank spaces. Examples are:

"Hello Java" "1997" "WELL DONE" "?...!" "5+3" "X"

Backslash Character Constants

Java supports some special backslash character constants that are used in output methods. For example, the symbol '\n' stands for newline character. A list of such backslash character constants is given in Table 2.1. Note that each one of them represents one character, although they consists of two character. These characters combination are known as *escape sequences*.

Table 2.1 ***Backslash Character Constants***

Constant	*Meaning*
'\b'	back space
'\f'	form feed
'\n'	new line
'\r'	carriage return
'\t'	horizontal tab
'\"	single quote
'\" '	double quote
'\\'	backslash

Variables

A *variable* is an identifier that denotes a storage location used to store a data value. Unlike constants that remain unchanged during the execution of a program, a variable may take different values at different times during the execution of the program. In Chapter 1, we had used several variables. For instance, we used variables **length** and **breadth** to store the values of length and breadth of a room.

A variable name can be chosen by the programmer in a meaningful way so as to reflect what it represents in the program. Some examples of variable names are:

- average
- height
- total_height
- classStrength

As mentioned earlier, variable names may consist of alphabets, digits, the underscore (_) and dollar characters, subject to the following conditions:

1. They must not begin with a digit.
2. Uppercase and lowercase are distinct. This means that the variable **Total** is not the same as **total** or **TOTAL**.
3. It should not be a keyword.
4. White space is not allowed.
5. Variable names can be of any length.

Data Types

Every variable in Java has a data type. Data types specify the size and type of values that can be stored. Java language is rich in its *data types*. The variety of data types available allow the programmer to select the type appropriate to the needs of the application. Data types in Java under various categories are shown in Fig. 2.2. Primitive types (also called *intrinsic* or *built-in* types) are discussed in detail in this chapter. Derived types (also known as *reference* types) are discussed later as and when they are encountered.

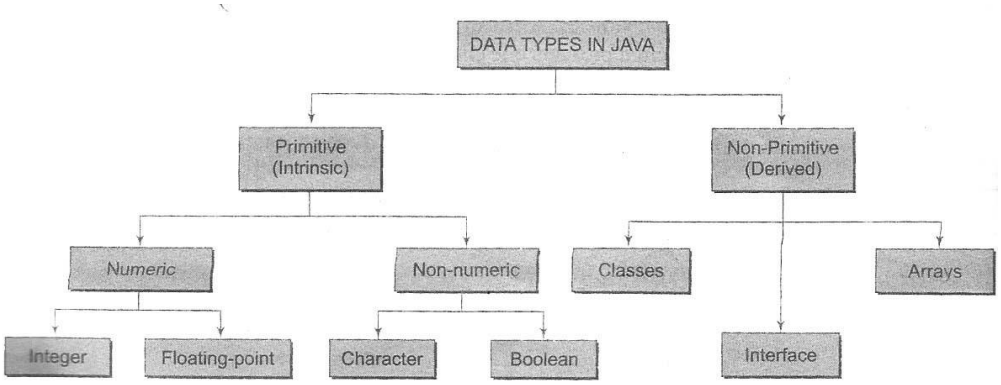

Fig 2.2 ***Data types in Java***

Integer Types

Integer types can hold whole numbers such as 123, -96, and 5639. The size of the values that can be store depends on the integer data type we choose. Java supports four types of integer. They are **bytes, short, int,** and **long**. Java does not support the concept of *unsigned* types and therefore all Java values are signed meaning they can be positive or negative. Table 2.3 shows the memory size of all the four integer data types.

Table 2.3 ***Size and Range of Integer Types***

Type	***Size***
Byte	One byte
Short	Two bytes
int	Four bytes
Long	Eight bytes

We can make integers **long** by appending the letter L or 1 at the end of the number. Examples:

```
123L    or    1231
```

Floating Point Types

Integer types can hold only whole numbers and therefore we use another type known as *floating point* type to hold numbers containing fractional parts such as 27.59 and –1.375 (known as floating point constants). There are two kinds of floating point storage in Java.

The **float** type values are *single-precision* numbers while the **double** types represent *double-precision* numbers. Table 2.4 gives the size of these two types.

Floating point numbers are treated as double-precision quantities. To force them to be in single-precision mode, we must append f or F to the number. Example:

```
1.23f
7.56923e5f
```

Table 2.4 ***Size and Range of Floating Point Types***

Type	***Size***
float	4 bytes
double	8 bytes

Double-precision types are used when we need greater precision in storage of floating point numbers. All mathematics functions, such as sin, cos and sqrt return **double** type values.

Character Type

In order to store character constants in memory, Java provides a character data type called **char**. The char type assumes a size of 2 bytes but, basically, it can hold only a single character.

Boolean Type

Boolean type is used when we want to test a particular condition during the execution of the program. There are only two values that a Boolean type can take: **true** or **false**. Boolean type is denoted by the keyword **Boolean** and uses only one bit of storage.

All comparison operators (see Chapter 5) return Boolean type values. Boolean values are often used in selection and iteration statements. The words **true** and **false** cannot be used as identifiers.

Declaration of Variables

In Java, variables are the names of storage locations. After designing suitable variable names, we must declare them to the compiler. Declaration does three things:

1. It tells the compiler what the variable name is.

2. It specifies what type of data the variable will hold.

3. The place of declaration (in the program) decides the scope of the variable.

A variable must be declared before it is used in the program.

A variable can be used to store a value of any data type. That is, the name has nothing to do with the type. Java allows any properly formed variable to have any declared data type. The declaration statement defines the type of variable. The general form of declaration of a variable is:

```
type variable1, variable2, ……, variableN;
```

Variables are separated by commas. A declaration statement must end with a semicolon. Some valid declarations are:

```
int       count ;
float     x, y ;
double    pi ;
byte      b ;
char      c1, c2, c3 ;
```

Giving Values to Variables

A variable must be given a value after it has been declared but before it is used in an expression. This can be achieved in two ways:

1. By using an assignment statement

2. By using a read statement

Assignment Statement

A simple method of giving value to a variable is through the assignment statement as follows:

```
variableName = value;
```

For example:

```
initialValue   =  0 ;
finalvalue     =  100 ;
yes            =  'x' ;
```

We can also string assignment expression as shown below:

```
x = y = z = 0 :
```

It is also possible to assign a value to a variable at the time of tis declaration. This takes the form:

```
Type variableName = value;
```

Examples:

```
int      finalValue  =   100 ;
char     yes         =   'x' ;
double   total       =   75.36;
```

The process of giving initial values to variables is known as the *initialization*. The ones that are not initialized are automatically set to zero.

The following are valid Java statements:

```
float   x, y, z;          // declares three float variables
int     m = 5, n = 10;    // declares and initializes two int variables
int     m, n = 10;        // declares m and n and initializes n
```

Read Statement

We may also give values to variables interactively through the keyboard using the **readLine( )** method as illustrated in Program 2.1.

Program 2.1 ***Reading data from keyboard***

```
Import java.io.DataInputStream;
Class Reading
{
  public static void main(String args[])
  {
    DataInputStream in = new DataInputStream(System.in);
    int intNumber = 0;
    float floatNumber = 0.0f;
    try
   {
     System.out.println("Enter an Integer; ");
      intNumber – Integer.parseInt(in.readLine());
      System.out.println("Enter a float number: ");
      floatNumber =
              Float.valueOf(in.readline()) .floatValue();
   }
   catch (Exception e) { }
   Sytem.out.println("intNumber = " + intNumber);
   System.out.println("floatNumber – " + floatNumber);
  }
}
```

The interactive input and output of Program 4.1 are shown below:

```
Enter an inter:
123
Enter a float number:
123.45
intNumber = 123
floatNumber = 123.45
```

The **readLine( )** method (which is invoked using an object of the class **DataInputStream**) reads the input from the keyboard as a string which is then converted to the corresponding data type using they data type wrapper classes.

Note that we have used the keywords **try** and **catch** to handle any errors that might occur during the readings process. Java requires the. See unit for more details details on error handling.

Scope of Variables

Java variables are classified into three kinds:

- *instance* variables,
- *class* variables, and
- *local* variables,

Instance and class variables are declared inside a class. Instance variables are created when the objects are instantiated and therefore they are associated with the objects. They take different *values* for each object. On the other hand, class variables are global to a class and belong to the entire set of objects that class creates. Only one memory location is created for each class variable. Instance and class variables will be considered in detail in Chapter 8.

Variables declared and used inside methods are called *local variables*. They are called so because they are not available for use outside the method definition. Local variables can also be declared inside program blocks that are defined between an opening brace { and a closing brace }. These variables are visible to the program only from the beginning of its program block to the end of the program block. When the program control leaves a block, all the variables in the block will cease to exist. The area of the program where the variable is accessible (i.e., usable) is called its *scope*.

We can have program blocks within other program blocks (called *nesting*) as shown in Fig. 2.3.

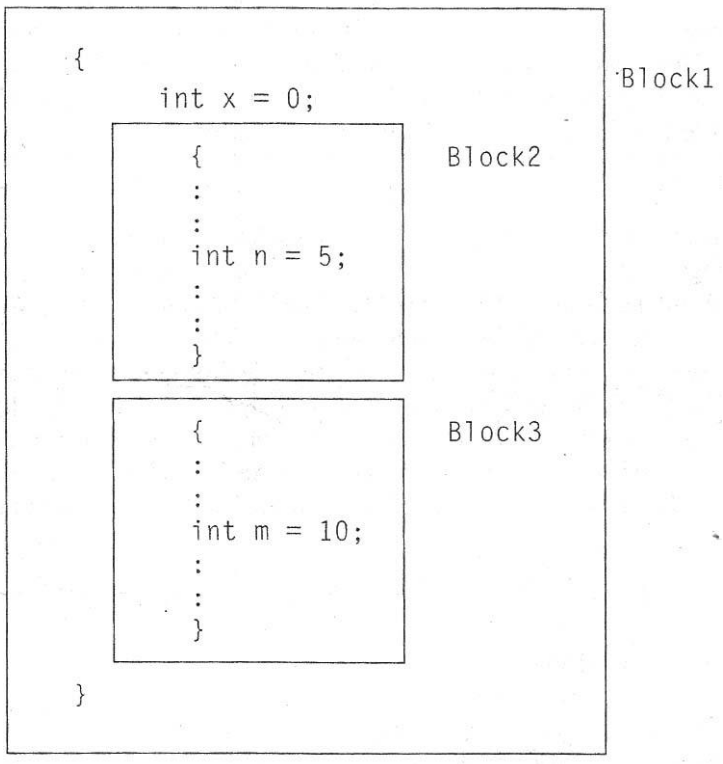

Fig. 2.3 ***Nested program blocks***

Each block can contain its own set of local variable declarations. We cannot, however, *declare* a variable to have the same name as one in an outer block. In Fig. 2.3, the variable **x** declared in Block1 is available in all the three blocks. However, the variable **n** declared in Block2 is available only in Block2, because it goes out of the scope at the end of Block2. Similarly, **m** is accessible only in Block3.

Note that we cannot declare the variable **x** again in Block2 or Block3 (This is perfectly legal in C and C++).

Symbolic Constants

We often use unique constants in a program. These constants may appear repeatedly in a number of places in the program. One example of such a constants is 3.142, representing the value of the mathematical constant "pi". Another example is the total of students whose mark-sheets are analysed by a 'test analysis program'. The number of students, say 50, may be used for calculating the class total, class average, standard deviation, etc. We face two problems in the subsequent use of such programs. They are:

1. Problem in modification of the program.

2. Problem in understanding the program.

Modifiability

We may like to change the value of "pi" from 3.142 to 3.14159 to improve the accuracy of calculations or the number 50 to 100 to process the test results of another class. In both the cases, we will have to search throughout the program and explicitly change the value of the constant wherever it has been used. If any value is left unchanged, the program may produce disastrous outputs.

Understandability

When a numeric value appears in a program, it use is not always clear, especially when the same value means different things in different places. For example, the number 50 may mean the number of students at one place and the 'pass marks' at another place of the same program. We may forget what a certain number meant, when we read the program some days later.

Assignment of a symbolic name to such constants frees us from these problems. For example, we may use the name **STRENGTH** to denote the number of students and **PASS_MARK** to denote the pass marks required in a subject. Constant values are assigned to these names at the beginning of the program. Subsequent use of the names **STRENGTH** and **PASS_MARK** in the program has the effect of causing their defined values to be automatically substituted at the appropriate points. A constant is declared as follows:

```
final type symbolic-name = value;
```

Valid examples of constant declaration are:

```
final   int     STRENGTH = 100;
final   int     PASS_MARK = 50;
final   float   PI = 3.14159;
```

Note that:

1. Symbolic names take the same form as variable names. But, they are written in CAPITAL to visually distinguish them from normal variable names. This is only a convention, not a rule.
2. After declaration of symbolic constants, they should not be assigned any other value within the program by using an assignment statement. For example, STRENGTH = 200; is illegal.
3. Symbolic constants are declared for types. This is not done in C and C++ where symbolic constants are defined using the # define statement.
4. They can NOT be declared inside a method. They should be used only as class data members in the beginning of the class.

Type Casting

We often encounter situations where there is a need to store a value of one type into a variable of another type. In such situations, we must cast the value of be stored by proceeding it with the type name in parentheses. The syntax is:

```
type variable1 = (type) variable2;
```

The process of converting one data type to another is called *casting*.

Examples:

```
int   m     =     50;
byte  n     =     (byte)m;
long  count =     (long)m;
```

Casting is often necessary when a method returns a type different than the one we require.

Four integer types can be cast to any other type except Boolean. Casting into a smaller type may result in a loss of data. Similarly, the **float** and **double** can be cast to any other type except Boolean. Again, casting to smaller type can result in a loss of data. Casting a floating point value to an integer will result in a loss of the fractional part. Table 2.5 lists those casts, which are guaranteed to result in no loss of information.

Table 2.5 ***Casts that Results in No Loss of Information***

From	*To*
byte	short, car, int, long, float, double
short	int, long, float, double
char	int, long, float, double
int	long, float, double
long	float, double
float	double

Automatic Conversion

For some types, it is possible to assign a value of one types to variable of a different type without a cast. Java does the conversion of the assigned value automatically. This is known as *automatic* type conversion. Automatic type conversion is possible only if the destination type has enough precession to store the source value. For example, **int** is large enough to hold a **byte** value. Therefore,

```
byte  b = 75;
int   a = b
```

are valid statements.

The process of assigning a smaller type to a larger one is known as *widening* or *promotion* and that of assigning a larger type to a smaller one is known as *narrowing*. Note that narrowing may result in loss of information.

Program 2.2 illustrates the creation of variables of basic types and also shows the effect of type conversions.

Program 2.2 ***Creation and casting of variables***

```
class TypeWrap
{
  public static void main(String args[])
  {
      System.out.println("Variables created");
      Chart c = 'x';
```

```
        byte b = 50;
        short s = 1996;
        int i = 123456789;
        long l = 1234567654321L;
        float f1 = 3.142F;
        float f2 = 1.2e-5F;
        double d2 = 0.000000987;

        System.out.println(" c = " + c);
        System.out.println(" b = " + b);
        System.out.println(" s = " + s);
        System.out.println(" i = " + i);
        System.out.println(" l = " + l);
        System.out.println(" f1 = " + f1);
        System.out.println(" f2 = " + f2);
        System.out.println(" d2 = " + d2);

        System.out.println(" ");
        System.out.pringln("Types converted");
        short s1 = (short)b;
        short s2 = (short)i:  // Produces incorrect result
        float n1 = (float)1;
        int m1 = (int)f1;     // Fractional part is lost

        System.out.println(" (short)b = " + s1);
        System.out.println(" (short)i = " + s2);
        System.out.println(" (float)l = " + n1);
        System.out.println(" (int)f1 = " + m1);
    }
}
```

Output of Program 2.2 is as follows:

```
        Variables created
        c = x
        b = 50
        s = 1996
        i = 123456789
        l = 1234567654321
        f1 = 3.142
        f2 = 1.2e-005
        d2 = 9.87e-007
    Types converted
        (short)b = 50
        (short)i = -13035
        (float)l = 1.23457e + 012
```

```
(int)f1 = 3
```

Note that floating point constants have a default type of **double**. What happens when we want to declare a **float** variable and initialize it using a constant? Example:

```
float x = 7.56;
```

This will cause the following compiler error;

"Incompatible type for declaration. Explicit cast needed to convert double to float."

This should be written as:

```
float x = 7.56F;
```

Getting Values of Variables

A computer program is written to manipulate a given set of data and to display or print the results. Java supports two output methods that can be used to send the results to the screen.

- `print( ) method` // print and wait
- `println( ) method` // print a line and move to next line

The **print( )** method sends information into a buffer. This buffer is not flushed until a newline (or end-of-line) character is sent. As a result, the **print( )** method prints output on one line until a newline character encountered. For example, the statements

```
System.out.print ("Hello ");
System.out.print ("Java!");
```

will display the words **Hello Java!** on one line and waits for displaying further information on the same line. We may force the display to be brought to the next line by printing a newline character as shown below:

```
System.out.print ('\n');
```

For example, the statements

```
System.out.print ("Hello");
System.out.print ("\n");
System.out.print ("Java!");
```

will display the output in two lines as follows:

```
Hello
Java!
```

The **print( )** method, by contrast, takes the information provided and displays it on a line followed by a line feed(carriage-return). This means that the statements

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

will produce the following output:

```
Hello
Java!
```

The statement

```
System.out.println ( );
```

will print a blank line. Program 2.3 illustrates the behaviour of **print( )** and **print( )** methods.

Program 2.3 ***Getting the result to the screen***

```
class Displaying
{
  public static void main(String args[])
  {
```

```
      System.out.println("Screen Display");
      for(int i = l, i <= 9; i++)
      {
         for (int j = 1; j <= i; j++)
        {
          System.out.print("  ");
          System.out.print(i);
        }
        System.out.print("\n");
       }

}
```

Program 2.3 displays the following on the screen:

```
Screen Display
      1
      2 2
      3 3 3
      4 4 4 4
      5 5 5 5 5
      6 6 6 6 6 6
      7 7 7 7 7 7 7
      8 8 8 8 8 8 8 8
      9 9 9 9 9 9 9 9 9
```