Chapter – 2

Data Types and Memory Organization

Dulal Acharjee

In this chapter discussions will be on different data types and its memory organizations. Different data types are: *int, float, byte, double, long* for numerical numbers which are required to define different sizes of variables. Again, another type of data are: *char* which is defined to a character variable. When we need to work with multiple characters like a name of a person, or to write a sentence, then, we need to define a variable as *String* which is a inbuilt class to work with large number of characters including white spaces. Though *String* is not a primitive data type, it is an inbuilt Class. Variables are required to declare with their data type. The reason behind is that, different data types has different sizes of memory as discussed later. Once, variables are declared, then, only these can be used by their names as x, y, z etc. Declaration of data type should be done before uses of the variables. Some examples of declarations of variables are shown below:

```
int x;
char y;
float salary;
double distance_of_Sun
```

Like that, for working with many characters, an object, like a variable, can be created as:

```
String address;
```

In other words, it can be described as ‘address- has a data type as String’ and address can hold a stream of characters known as string. On a string different types of operations are done like changing its attribute to bold, italic, underline, or other string operations like: concatenation of two strings, convert to uppercase or lowercase, cutting string from left side, right side or from middle for a fixed number of characters, string insert etc. All these types of string operations are available under the Class *String* by name of different methods.

Different numerical data types:

In different applications of programs, numbers of different sizes are required. When a number is assigned or input into a variable, it occupies some memory of the computer. Memories are organized as a block of 8 bits (Byte) known as smallest block of memory of any standard computer as discussed in next section. To avoid occupation of unnecessary memory-blocks, and to avoid processing complexities, numbers are grouped as required for 1 byte, 2 bytes, 4 bytes, 8 bytes etc. sizes. These are defined as different data types known as variable identifiers as:

```
byte x=25;//one byte size or 8 bits or one block of memory
short y=25;//two bytes size or 16 bits or two blocks of memory
int z=25;//four bytes are occupied
float p = 25;// four bytes
double q = 25;// eight bytes are occupied from the memory
long r=25; //eight bytes are occupied from the memory
char ch= 'a';// two types are occupied.
boolean  YES= true;//size is not one bit, it has other compiler based size.
```

By these examples it is shown that in all cases, the same number 25 is assigned into the variables, but, the lengths of occupied memories are different. So, question arises, why are we declaring a variable as short, int, long, float or as double when only one byte space is sufficient to store 25? Answer is that if we are confirmed that within this program, value of x will not be replaced by a larger value, then we should declare it as 'byte x=25'. But for large and larger numbers, we can't hold within *byte* type of data. For that reason, according to the sizes of numbers we need to declare as int, long, double etc.
Assigning numbers in other number formats using prefix 0x for hexa and 0b for binary numbers:

```
int nbase10 = 526;
//The number, 526, in hexadecimal is written as an example,
int nbase16 = 0x32E;
// The number, 526, in binary is written as an example,
int binVal = 0b1000001110;
```

Literal representations:

With the declaration of float and double, by default 'f / F'(32 bit float literal) or 'd / D'(64 bit double literal) are there if we mention or not.

```
double x = 523.9;
The same value can be written as in scientific notation
double y = 5.239e2;
float z  = 523.9f;
```

For '*char x*', 'x' is of two byte size and it can represent 2^{16} or 0-65535, total 65536 types of character codes. These code numbers are of unsigned , i.e., MSB, Most Significant Bit, is not sign bit for characters but also considered as 16 bits space for generating code numbers of characters of different countries and for producing special characters.

Problem: using Unicode of Java, demonstrate to print characters of other countries indifferent ways.

```
//ChDemo.java: demo of unicode to print characters of other Country.
//different ways of working with characters.
//unicode uses 16 bits to form characters.
import java.lang.*;
class ChDemo
{
  public static void main(String agrs[])
  {
        //String str = "\u0041";
        String str = "\u00F6";
        char c = '\u00E6';
        Character ch='\u00E2';
        System.out.println("Unicode str Ch = " + str);
        System.out.println("Unicode c = " + c);
        System.out.println("Unicode Ch = " + ch);
  }
}
OUTPUT:
Unicode str Ch = ö
Unicode c = æ
Unicode Ch = â
```

Discussion: in java, '\u' is denoted as escape sequence of unicode and it is represented as BCD(Binary Code for Hexadecimal) form. For 16 bits(2bytes of char type) it is written as the maximum value: 'FFFF' where 'F' is the highest digit in hexadecimal number system. There are 65536 possibilities to generate different characters by 2 byte of *char* data types. Within these range, all character alphabets of different countries are possible to generate and in this example that is demonstrated.

Note: at the time of declaration of variable, we should guess what could be the maximum size of the data and what type of data should be required for that. We should not occupy unnecessary memory by declaring a larger data type for a smaller data.

Data Type casting

Size of data type 'short' is of two bytes (16 bits). Eight bits make one byte, 16 bits is of two bytes. The first bit holds the sign symbol of the number, so, rest of 15 bits can holds '1' in 15 positions. So, the numbers it can hold are -32768 to 32767.
• int is of four bytes (32 bits). Data range is -2147483648 to 2147483647.

• long is of eight bytes (64 bits). Data range is -9223372036854775808 to 9223372036854775807.

To understand deeply, it is required to study computer architecture, memory/ register architecture of the computer. Then number system is also related with this concept.

```
int x=100;
byte y= (byte) x;
long z=(long) x;
```

Sometimes, due to data casting data may be dost, i.e, a larger data can't be casted within a smaller data type variable. As example, a long or double type of variable can't be casted within an 'int' type, then data will be lost. It has some concept of sizes of computer register and bit size of int, double, long, short etc. types of data.

Automatic conversion is performed automatically as example:

```
byte x =45;
int y=x;
```

Here, y is an integer variable which converts byte type of data of x; so, now value of y is 45 and it is an integer data type.

```
//TypeCast.java: data type casting or redefining data type inline.
class TypeCast
{
	public static void main(String[] args)
	{
		//assigning values to variables
		byte bb=68;
		int ii=123456789;
		long jj=1234567654321L;
		short ss1=(short)bb;
		short ss2=(short)ii;
		float nn=(float)jj;
		//printing values and check what happens
		System.out.println(" value of short bb : " + ss1);
		System.out.println(" value of short ii : " + ss2);
		System.out.println(" value of float jj : " + nn);
	}
}
```

```
E:\example>javac TypeCast.java
E:\example>java TypeCast
 value of short bb : 68   // it is correct output
 value of short ii : -13035  // data is lost; original value was ii=123456789
 value of float jj : 1.23456769E12 // data is lost; original jj=1234567654321L
```

Note: data casting should be chosen carefully, the science is that within a larger pod you can put a smaller one, but can't put a larger pod within a smaller one. To understand this point, architecture of the Microprocessor which is the main computing processor of the computer, should be learnt.

Enumerated data Type:

It is a special type of data to create list of indexes over a list of variables. Sometimes, it is required to give indexes as 1,2,3…n serial numbers to a list as days of the week, month-names of the year etc.

```
//ExEnum.java: Command line argument: to run the program we use
// java progname; if we put extra words after progname, these
//words are inserted within args[] arrays.
class ExEnum
{
	//declaration of new data type 'Day' having 7 elements.
	//numbering starts from zero, to first Day 'SUN' and 6 to 'SAT'
	enum Day { SUN, MON, TUES, WED, THURS, FRI, SAT }
	public static void main(String[] args)
	{
		Day dd, ddd;
		dd= Day.THURS;
		System.out.println(dd);
		System.out.println("We know:"+dd+" is the " + dd.ordinal() +" th
								day of the Week.)
	}
}

E:\example>javac ExEnum.java
E:\example>java ExEnum
THURS
 We know: THURS is the 4 th day of the Week
```

Note: enum counts starting from zero, 0.
Discussion: using ‘enum’ any weightage value can be assigned to a set of variables as names of months, serial number of class rooms etc.

Built-in keywords:

In JAVA, there are some inbuilt keywords which have been used by JAVA compiler for their internal uses; so, these keywords must not be used as user defined variables, class /object names, method names, file names etc. These words are reserved by the compiler and known as different identifiers and tokens of compiler of JAVA.

Table: 2.1 Reserved words or tokens of JAVA

abstract	assert	boolean	break
byte	case	catch	char
class	const	continue	default
do	double	else	enum

extends	false	final	finally
float	for	goto	if
implements	import	instanceof	int
interface	long	native	new
null	package	private	protected
public	return	short	static
strictfp	super	switch	synchronized
this	throw	throws	transient
true	try	void	volatile
while			

Testing-1: To proof the theoretical concept of using tokens: ‘tokens can’t be used as a variable’, we have written some lines as declaring a variable by name ‘for’ and assigned value 10 within that variable, we compile the program and found five errors which are printed below to validate the theoretical concept.

```
//TokenEx.java
class TokenEx
{
        public static void main(String[] args)
        {
                int for=10;// define & assign value to a token
                System.out.println(" value=  " + for);
        }
}
```

```
OUTPUT:
C:\example>javac TokenEx.java
TokenEx.java:10: error: not a statement
        int for=10;
        ^
TokenEx.java:10: error: ';' expected
        int for=10;
           ^
TokenEx.java:10: error: '(' expected
        int for=10;
              ^
TokenEx.java:13: error: illegal start of expression
        System.out.println(" value=  " + for);
                                          ^
TokenEx.java:13: error: '(' expected
        System.out.println(" value=  " + for);
                                             ^
5 errors
```

Testing-2: In this test case design, we have created a class by name ‘for’ and saved the program as name ‘for.java’ and compiled to check our theoretical concept that ‘*a class name can’t be given as a name of reserved token name*’. We get the error ‘<identifier > expected’; so, in both test cases, it is proved that user defined variables can’t be given as the name of reserved tokens.

```
//for.java : to demonstrate that 'for' word can't be used as name of a program.
class for
{
        public static void main(String[] args)
        {
                //int for=10;// define & assign value to a token
                //System.out.println(" value=  " + for);
        }
}
OUTPUT:
C:\example>javac for.java
for.java:3: error: <identifier> expected
class for
    ^
1 error
```

Problem: wap to demonstrate different mathematical operators like + - * / % and show the use of Math class method 'sqrt()'

```
//MulDiv.java using math operators like +,*,/ etc.
// using math function sqrt() square root.
//mod operator % returns remainder after division.
import java.lang.Math;
class MulDiv
{
   public static void main(String args[])
  {
        int p=3;
        double x=5;
        double y;
        y = Math.sqrt(x);
        double z=x*y;
        double q=z/2;
        int r=(int)x%p; //data type casting
        System.out.println("y = " + y);
        System.out.println("\n z= " +z);
        System.out.println("\n q= " +q);
        System.out.printf(" Value of r is: %d",r);
     }
}
OUTPUT:
y = 2.23606797749979
 z= 11.180339887498949
 q= 5.5901699437494745
 Value of r is: 2
```

Discussion: simple mathematical operations are shown in this example. Operator symbol, %, is called 'mod' which gives remainder after division. But, the control symbol, %d, will print integer value in that place. *Java.io* package has two similar methods, *printf* and *format*, having same roles. Here is an example of using '*format*' like *printf*:

```
System.out.format("Any string: " + "put float data: " + "%f, " + "Any integer data: %d, " + "put any string: %s", float_x, int_y, string_st);
```

Here, the first ‘%’ is known as format specifier and the second character is known as ‘converter’, both works together. Like ‘printf()’, ‘format()’ prints but cursor stays at the same like at end. If we need to move cursor to the next line, then at end ‘%n’, new line-specifier should be mentioned.

Precedence of Operators:

Like algebraic operations of mathematics, computer languages also follow some rules of precedence of using operators. We can say this rules are the basic of computing process. JAVA has defined rules of precedence of operators and is mentioned from highest precedence to lowest. In computing process, operators of highest precedence will be computed first and so on. Some examples:

```
X=(5-3)*4+5; //x=13
Y=5-3*4/2-1; //y= -2
Z=9%2*5+12/3*2;//z=13
```

Manufacturer of JAVA has mentioned about the precedence of operators within their literature which is non-changeable, non-editable; only we are describing as the requirements of readers of preliminary stages. Operators mentioned in the same line of the table, mentioned below, have similar precedence indices. Assignment operators, = and another operators attached with = symbol, (as +=, >=, != etc.) works from right to left, i.e., it starts computing from right most variables, gathers all values as a data value and then assigns within the left variable. But, all binary operators(&, ^, | etc.) work from left to right.

Table: 2.2 List of operators as per highest to lowest precedence

Operators precedence (highest to lowest)	Name of operators with descriptions
. [ ] ( )	Dot operator is used to access variables and methods of an object, [] is used for arrays and ( ) first brackets are used for grouping.
x-- *x++*	Postfix and unary (works on single variable)
--*x* ++x -*x* +*x* ~x !x	Prefix and unary(works on single variable)
* / %	Multiplication, division and mod(two operands are required)
- +	Add, sub(two operands are required)
<< >> >>>	Left, right and shift with carry types of Shift operators
< > <= >= instanceof	Relational operators(less, greater, less than and equal, greater than and equal)
== !=	Comparative equal, not equal(two operands are required)

&	Bitwise AND (two operands are required)
^	Bitwise EOR(two operands are required)
|	Bitwise OR(two operands are required)
&&	Logical AND(two operands are required)
||	Logical OR(two operands are required)
? :	Ternary operator(two operands are required)
= += -= *= /= %= &= ^= |= <<= >>= >>>=	All assignment operators(at right end only one data value)

Problem: wap to demonstrate role of unary operators and its uses

```
//OpPrece.java: demonstration of precedence of operators.
class OpPrece {
   public static void main(String[] args) {
         int x = 12, y = 4, z = 2;
         //int output = x - ++z - ++y;//x-(3)-(5) = 4
            int output = x - ++z + --y;//x-(3)+(3)= 12
            System.out.println(" Final result: " + output);
            //now the value of z is z+1 and ++z= (z+1) +1;//z=3, ++z=4
            System.out.printf("Value of z= %d and z++ is = %d %n",z,++z);
   }
}
```

OUTPUT:
Final result: 12
Value of z= 3 and z++ is = 4

Discussion: in this example, two unary operators are used as ++z and –y; unary operator works on single variable. First, value is increased by one and then, take part in operation. Contrary of that, z++ and y-- are known as post increment by one, i.e., the value of variable will take part in operation and after that it will increase by one.

Problem: wap to input an year number from keyboard and then test for whether it is Leap year or not.

```
//OpPrece.java: demonstration of precedence of operators.
import java.util.Scanner;
class LpYr {
   public static void main(String[] args) {
            Scanner ob= new Scanner(System.in);
            System.out.printf("Input any year: ");
            int yr=ob.nextInt();
            if((((yr % 4) == 0) && ((yr % 100) != 0)) || ((yr % 400) == 0))
              System.out.println(" It is Leap year : "+yr);
            else
              System.out.println(" It is not Leap year : "+yr);
```

```
    }
}
OUTPUT:
Input any year: 2020
 It is Leap year: 2020

C:\example>java LpYr.java
Input any year: 2021
 It is not Leap year : 2021
```

Discussion: in this program if the logic is true then it will print for yes Leap year else not. Logic of testing Leap year is given within if(..), when the inside logic will satisfy the condition it will generate true else false. Logics of testing Leap year are:
(a)Must be divided by 4 and should not be divided by 100. or
(b)Should be divided by 400

Increment-Decrement Operators:

Two plus signs, + +, it is used for increment by one and known as unary operator. Example, x = x + 1; it increased value of x by one. This can be written as x += 1 ; another advanced notation is x++ meaning that x will be used in operation, then, value of x will be increased by one and then the increased value will be used with next operation. It has another use: ++x ; meaning is that previous value of x will be increased by one first, then, increased value will be used in operation.
- - (two minus sign): it is decreasing value by one.
x--; value of x will be used in operation, then, it will be decreased by one.
--x : first value of x will be decreased by one then take part in operation.

```
//NumberXpp.java: to demonstrate ++ operator.
class NumberXpp
{
        public static void main(String args[])
        {
                int x=5;
                int y;
                //int z = ++x;
                int z = x++;
                System.out.println(" Result = " + x + " value of z = " +z);
        }
}
E:\example>java NumberXpp
 Result = 6    value of z = 5
```

Discussion: in theory, it is mentioned that , in case of x++ , first of all previous value of x will take part in operation and the it will be increased. In the example of above, x value was 5, and that was transferred to z by the assignment, z = x++ , so, when we printed value of z it shows 5, after transferring its own value, then, it has increased own value by one and goes to next line of program. When we print output of x it shown 6 and for z it shows 5. Out theory is proved. But, if we use z= ++x , in that case first x value will be increased by one to 6 and will be assigned to z. In that case output of both x and z will be 6 and 6. In the same way, the decrement operator can be proved also.

Ternary Operator:

There are three operators in one statement as shown in this example as:

```
Test-condition? True-result:false-result
int larger = x>y ? x : y ;
```

Meaning of this statement is that if x is greater than y, then value is x else y. The value of x or y is assigned within the left variable ‘larger’; this ternary statement is the shortest form of if..else statement of decision taking.

Data Types and Ranges of Numbers

JAVA has eight primitive data types to define a variable within a fixed size of memory. As these are primitive data types, meaning that these are predefined. When any program is written with variables, according to the requirements of sizes of numbers, data type should be decided. Reasons of learning of it are that a variable of larger block can’t be inserted within a smaller variable. But, generally, a smaller data can be inserted within a larger data block. We can explain in that way: a larger envelope can’t be inserted within a smaller one but reverse is possible.

When a variable is declared, it occupies some memory blocks from the physical memory of the computer/electronic device. To manage memory that unnecessary extra memory are not occupied JAVA has these eight types of data types. Among these, *byte*, *short*, *int* and *long* are of integer type of signed data types. Java has no unsigned data types like C++. The other two types, *float* and *double* are for holding decimal numbers or fraction of numbers.

Rather than primitive data types, there are some other user created data types like *class*, *enum*, *interface* etc.; these are composite data types, because, within a composite data type- many variables, methods, constructors etc. may reside and as a whole it gives a name of a class type of data. A class is a data type; enumeration can create a new data type; an interface is also like a class type of data. These are known as user defined and names can be anything given by users.

Table: 2.3 List of primitive data types

Data type	Size/length	Bytes	Range of numbers
byte	8bits	1	-128 to + 127
short	16bits	2	-32768 to +32767
char	16 bits	2	Characters are represented by code numbers
int	32 bits	4	-2147483648 to +2147483647
float	32 bits	4	$\pm 10^{38}$ and 7 significant decimal digits

double	64 bits	8	$\pm10^{308}$ and 15 significant decimal digits.
long	64 bits	8	-9223372036854775808 to + 9223372036854775807
boolean	1 bit ***	***	True/false, false/true is defined by 0/1.

*** Boolean size is fictitious to say it is one bit as mentioned in some books. In computer memory, data are stored as bytes(8 bits); if for a Boolean data one bit is reserved to put 1 for YES and 0 for NO, then other bits can't be used for other data storage purpose. So, we see, minimum one byte would be required. Again, if 'True' or 'False' words are required to store then more bytes are required. So, Boolean is Java compiler dependent and can't be the size only one bit.

Memory Organization:

To understand data type and number of bits required for a data, some basic concept of computer memory or register's organization is required to learn. Generally, data are stored within RAM(Random Access Memory) or ROM(Read Only Memory) which are electronic semiconductor type of chip. Data can be stored within a capacitor in the form of charge or within a P-N type of semiconductor in the form of depositing electrons. There are other devices like magnetic disk or optical disk. Charges are stored within these devices as data. If charge is stored that is considered as 1 and if no charge is stored that is considered as zero.

A general concept of a Memory is that: 8 bits memory chip is the array of 8 cells which can hold either 0 (zero) or 1(one) as in binary number system. All characters, numbers, symbols and nonprintable keys present on the keyboard of computer have individual ASCII code values. When a key is pressed, computer translates the corresponding ASCII code to binary numbers. As example, if we type 'A' then its ASCII value 65 is translated to binary number system as: 1000001. The bit patterns within the memory are shown below:

	S	1	0	0	0	0	0	1
Index position:	7	6	5	4	3	2	1	0

Weight of bits: $2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0$ (in binary system)

Computed value: $1*2^6 + 0*2^5 + 0*2^4 + 0*2^3 + 0*2^2 + 0*2^1 + 1*2^0$ (in binary system)

Final value : 64 + 0 + 0 + 0 + 0 + 0 + 1 = 65 (in decimal system)

By this example, it is shown how a character is translated and digital bits(0 or 1) are stored within memory cell.

But, when a numerical data is input that might be of any data type such as *short, int, float, long, double* etc. which is known as signed data. In JAVA, there is no unsigned identifier like C/C++. Any numerical data can be either positive or negative. If any sign(+ -) is not prefixed with any number, that one is considered as a positive number. In memory structure last bit is the sign bit. If we want to store a positive

number then last bit should be 0, zero, and for negative number the last bit should be 1, one. In simple words: last bit, 1 means minus and 0 means positive number.
According to memory organization of computer, 8 bits = 1Byte (unit of memory).
Let us look insight of the memory organizations of different data types. A byte, means 8 cells and each cell can hold either 0 or 1.
1 Byte = 8 bits; if all positions of a byte are occupied by 1s, then, it looks like:

*	1	1	1	1	1	1	1
7	6	5	4	3	2	1	0

A Byte of memory organization. First row is the Memory and the second row is the position index within the memory. Bit positions start from zero and ends at 7 for 8 bit memory.

If a variable is defined as: *byte x*; which indicate that the range of 'x' is of 8 bits or one byte and using 'x' maximum numbers can be processed are +127 and -128. If we need to process larger numbers, then we have to take larger data types as *short*, *int*, *float*, *long*, *double* etc. Taking an example of smallest data type 'byte', let us explain the pattern of bit distributions that how it holds bits within all 7 positions(except sign bit) to process the maximum number +127.

In a 8 bit memory block, if all bits are one, then the binary number is: $(1111111)_2$; there are seven ones while the last bit is sign bit. Internally, computer works with binary number system. Though computer circuits work with signals, if a high signal is stored that is considered as 1 and for a low signal below the threshold value is considered as 0, zero. In binary number system there are only two numbers: 0 and 1. To represent contents of memory, binary number system is used. A binary number is written as subscript 2 as an example: $(01011)_2 = (11)_{10}$. It is eleven in decimal number system.
Let us convert this number from binary to decimal number system:
The bit distribution of +127 is shown below:

0	1	1	1	1	1	1	1

The above number is represented as per weight value of each positions:

$$1*2^6 + 1*2^5 + 1*2^4 + 1*2^3 + 1*2^2 + 1*2^1 + 1*2^0$$
$$= 64 + 32 + 16 + 8 + 4 + 2 + 1 = 127$$

So, it is seen that if the sign bit is 0, then the largest number can be inserted within 8 bit memory is +127.
If the sign bit is 1 then the largest number can be handled by 8 bit memory is -128 and the bit pattern is as:

1	0	0	0	0	0	0	0

$$= 2^7 + 0 + 0 + 0 + 0 + 0 + 0 + 0 = 128$$

As the sign is 1, so, this number will be considered as minus = - 128
There are two methods of representation of numbers:

a. **Signed magnitude representations**: examples mentioned above are under this method and generally not used in computer system. Sum-subtract-multiply and division are complicated in this method and need complex circuits.
b. **Two's complement representation**: commonly used method of negative number representation and used for mathematical operations within the computer circuits. To convert a number in this system steps are:
    a. Make all bits as compliment of it as e.g. for a four bit memory organization, say number is : +5; its binary is 0101, then after making 1's compliment it is : 1010 , it is known as first compliment form.
    b. Add one with the first compliment, 1010 +0001 = 1011, so, -5 is (1011).

As, this book is for beginners of JAVA program learning, we shall not describe more here about number systems.

Let us explain with an example of signed magnitude representation form:
int x = 14;
Distribution of positional weight values are as: 14= 8 + 4 +2 +0

0	0	0	0	1	1	1	0	
				8	4	2	0	= 14

If we want to assign -14 into x as:
int x= -14;
Then, within the memory- bit patters of -14 will be as:

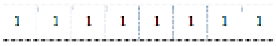
1	0	0	0	1	1	1	0	
				8	4	2	0	= -14

Or 128 + 0 + 0 + 0 + 8 + 4 + 2 + 0 = 142
As the last bit is 1, for a computer of 8 bit memory size, 142 is a negative number.

That is from zero to 0- 127 positive numbers can be processed by that type of computers. Or, if we take a variable as data type 'byte', in that case, maximum positive number can be processed is 127 and the lowest negative number can be processed is -128. Old days computers of 8085 had 8 bit registers to hold a number for processing and it could process ranges of numbers -128 to +127. Please, do not be confused, now a days computers or laptops use 32, 64 or 128 bits of memory block size; its meaning is that either 4, 8, 16 blocks of memory, as shown above, are the memory block sizes of our computers. Byte is the unit of memory and 8 bits make a byte.
Outcome of learning: we should declare a variable as a data type assuming the number size, a smallest data type is '*byte*' is of one byte and largest data types are '*long* and *double*' is of 8 bytes. Unnecessarily declaring a larger data type will occupy unwanted memory spaces.

Graphical representations of different data types

byte: 8 bits = 1 byte



short: 16 bits = 2 bytes

int = 32 bits = 4 bytes

In the same way double and long will have 8 blocks of memory.

Floating Point Representation:

Working with floating numbers needs sufficient care, because, in this type of operations after decimal point numbers are rounded after some digits. In some cases, we don't need extra digits after two digits of decimal point. Example given, price of a fruit is 30.505 USD, or 200.50789 INR. Generally, we ignore extra digits of decimal point after two digits. But, in some cases, if it is the weight of precious materials/chemicals, then we may need to count four/five digits after decimal point. Let us think, is it right to write that price= 30.50UDS instead of writing price=30.505USD? Computing system is that we take help of some machine either computer or calculator for computing purposes. In most of the cases of multiplication and division with floating numbers, some data are losses after decimal points, that is, after some digits it do rounds up as the expression of exponential, 10 to the power of written as e2, e3 etc. Let us think on this example:

PI= 3.141592653589 (approx.)
= 3.140000000000 = 314e-2; or 314 divide by 10 to the power minus two, or 314/100;
= 3.141000000000 = 3141e-3; or 3141/1000 has the same meaning

But, in this example, in both the cases as shown above, after decimal point, some digits are rounded to zero as its value is too small. Some data, may be very small, are lost in this type of computing process which is known as lossy transformation. So, we see, when we work with floating number system, we need to compromise upto some digit after floating point.
But, if we multiply 3.5 by two, result is 7.0 which is Loss less transformation where no digits are lost after this multiplication.
In decimal number system, a floating number can be written in different forms using exponential powers as:

1.75 = $0.275 \times 10^{1} = 0.0275 \times 10^{2}$ etc.
Or,
2.75 = $27.5 \times 10^{-1} = 275 \times 10^{-2}$ etc.

In all cases, we see the position of decimal point or floating point moves left or right sides and change the value of mantissa and exponent values. Movement of decimal point in left or right side is equivalent to multiply or divide by 10 as it is decimal number system.

In Binary number system, let us take an example:

$(1\ 0\ 1\ .\ 1\ 1)_2$ its value in decimal system is:

$1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0\ .\ 1 \times 2^{-1} + 1 \times 2^{-2}$

$= 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0\ .\ 1/2^1 + 1/2^2$

$= 4 \quad + \quad 0 \qquad + 1 \times 1\ .\ 0.5 + 0.25$

$= 5\ .75$

Let us shift decimal point by two left positions of the binary number as mentioned above:
101.11 after left shifting of two positions of floating point it will be:
1.0111×2^2 ; (this 2^2 is written in decimal number for understanding); we have to multiply the value of left side by 4. Now let us see what is the decimal value of (1.0111) :
In binary number system, moving decimal point left or right means multiply or divide by 2 as shown below:
1 . 0 1 1 1 in binary number system it will be as:
$= 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3} + 1 \times 2^{-4}$ in decimal number system
$= 1 + 0 + 0.250 + 0.125 + 0.0625$ in decimal
$= 1.\ 4375$ in decimal

So, 1.0111×2^2 : left part in binary and right part in decimal – for understanding
$= 1.4375 \times 4 = 5.75$ in decimal

In Java, different data types like ‘float’ and ‘double’ can handle floating point numbers or decimal numbers.

IEEE 754 standard of Floating Point representations are in binary format is shown below:
(i) Bit distributions of *float* data type is = 1 + 8 + 23 = 32 bits / 8 = 4 bytes.

S 1 bit	Exponent 8 bits	Mantissa 23 its

(ii) Bit distributions of double data type of = 1 + 11 + 52= 64 bits /8 = 8 bytes.

S 1 bit	Exponent(e) 11 bits	Mantissa(m) 52 bits

Note: 8 bits= 1 Byte.
Any decimal number of *float* Data Type is represented as normalized form by this formula:

$(-1)^{s} \text{ x m x } 2^{(e-127)}$

It is read as: minus one to power s multiple m multiple 2 to power e minus hundred twenty seven.
Rounding of any floating number is sometime important for scientific computation. For that, there are some, constants under *java.math.RoundingMode* class and these are:

HALF_EVEN: if input is 7.5, it will give output as 8.
HALF_DOWN: if input is 7.5, it will give output as 7.
HALF_UP: if input is 7.5, it will give output as 8.

Problem: wap to demonstrate processing of large floating point numbers which is larger than double/long using BigDecimal() then do rounding using rounding constants available under RoundingMode.

```
//BigFloat.java: working with very large floating numbers.
import java.math.BigDecimal;
import java.math.RoundingMode;
import java.io.*;
public class BigFloat {
   public static void main(String[] args) {
      double x = 1.29 * 52;
      System.out.println("Value of 1.29 * 52 = " + new BigDecimal(x));
      System.out.println("Rounded of 1.29 * 52= " + Math.round(x));
      BigDecimal p = new BigDecimal("4.1454");
      BigDecimal q = new BigDecimal("79");
      BigDecimal r = p.multiply(q);
      System.out.println("Multiplied value: "+ r);
      System.out.println("Rounded value: " + r.setScale(2,
                              RoundingMode.HALF_EVEN));
   }
}
OUTPUT:
C:\example>java BigFloat.java
Value of 1.29 * 52 = 67.0799999999999982946974341757595539093017578125
Rounded of 1.29 * 52= 67
Multiplied value: 327.4866
Rounded value: 327.49
```

Discussions: it is observed that data type of *x* is double, but, after decimal point there are 46 digits which is out of ranges of *double*. Here, *BigDecimal* has done this role to concatenate rest portions of the data. Finally, RoundingMode.HALF_EVEN has rounded the value for two digits after decimal point.

Details are available to the reference given below:
Note: (IEEE 754 Groups: 754_WG - Working Group for Floating-Point Arithmetic, C/MSC - Microprocessor Standards Committee, under IEEE Computer Society; this committee suggested this standard for processing of floating numbers).

Details of floating point are here: https://introcs.cs.princeton.edu/java/91float/
For rounding numbers:
https://docs.oracle.com/javase/7/docs/api/java/math/RoundingMode.html
A good video on number system: https://www.youtube.com/watch?v=9nCtaTNdAgc

Questions: JAVA Chapter-2:

2.1. Data size of 'char' type is of two bytes-why not defined it as one byte?
2.2. What is the size of 'int' data type in bits?
2.3. What is boolean data type? when is it required to declare?
2.4. What is literal representation of data type? How it works?
2.5. What is unicode? How internationl characters(rather than english) can be printed?
2.6. The keyword 'final' is used when and how?
2.7. For signed numbers, the last bit of the memery block is not considered as number formation, what are the reasons?
2.8. Data types, long and double are of 8 bytes, why are two datatypes required of same length?
2.9. As per IEEE 754 guide line what are the bit distribution of float data type?
2.10. As per IEEE 754 guide line what are the bit distribution of double data type?
2.11. Explain,what is the role of *BigDecimal(x)* ?
2.12. Explain the statement: *BigDecimal p = new BigDecimal("4.1454");*
2.13. What is the meaning of it *RoundingMode.HALF_EVEN* ?
2.14. What is the meaning of it *RoundingMode.HALF_UP* ?

2.15. Convert to binary number system:
12 =
16 =
245 =

2.16. Convert to Hexadecimal number system:
18 =
65 =
234 =

2.17. Convert binary number to decimal number system:
101 =
110011 =
10000001 =
1111111 =

2.18. Convert decimal floating numbers to decimal number system
11.001 =
1101.10101 =
111.110011 =

Link of PPT of this book and all source codes:

`https://drive.google.com/drive/folders/1bMxMCaqPe0W35COAdn_-BrnV_uzQ-tNO?usp=sharing`

(Note: to get access, copy and paste this link to your browser. Both PPT and PDF version of presentation of this book are uploaded in Google Drive.)