Lecture 3 : How is Data stored?

How is Data Stored ?

a) How are integers stored ?

The most commonly used integer type is int which is a signed 32-bit type. When you store an integer, its corresponding binary value is stored. The way integers are stored differs for negative and positive numbers. For positive numbers the integral value is simple converted into binary value and for negative numbers their 2's compliment form is stored.

Let's discuss How are Negative Numbers Stored?

Computers use 2's complement in representing signed integers because:

1. There is only one representation for the number zero in 2's complement, instead of two representations in sign-magnitude and 1's complement.

2. Positive and negative integers can be treated together in addition and subtraction. Subtraction can be carried out using the "addition logic".

Example:

int i = -4;
Steps to calculate Two's Complement of -4 are as follows:

Step 1: Take Binary Equivalent of the positive value (4 in this case)
0000 0000 0000 0000 0000 0000 0000 0100

Step 2: Write 1's complement of the binary representation by inverting the bits

1111 1111 1111 1111 1111 1111 1111 1011

Step 3: Find 2's complement by adding 1 to the corresponding 1's complement

```
 1111 1111 1111 1111 1111 1111 1111 1011
+0000 0000 0000 0000 0000 0000 0000 0001
 ..................................................
 1111 1111 1111 1111 1111 1111 1111 1100
```

Thus, integer -4 is represented by the binary sequence (1111 1111 1111 1111 1111 1111 1111 1100) in Java.

b) Float and Double values

In Java, any value declared with decimal point is by default of type double (which is of 8 bytes). If we want to assign a float value (which is of 4 bytes), then we must use 'f' or 'F' literal to specify that current value is "float".

Example:

```
float float_val = 10.4f;            //float value
double val = 10.4;          //double value
```

c) How are characters stored

Java uses Unicode to represent characters. As we know system only understands binary language and thus everything has to be stored in the form binaries. So for every character there is corresponding code – Unicode/ASCII code and binary equivalent of this code is actually stored in memory when we try to store a char.

Unicode defines a fully international character set that can represent all the characters found in all human languages. In Java, char is a 16-bit type. The range of a char is 0 to 65,536.

Example code:

```
public static void main(String[] args) {
        char ch1, ch2;
        ch1 = 88;      //ASCII value for 'X'
        ch2 = 'Y';
        System.out.println(ch1 +" " +ch2);
}
```

Output:
X Y

Adding int to char

When we add int to char, we are basically adding two numbers i.e. one corresponding to the integer and other is corresponding code for the char.

Example code:

```
public static void main(String[] args) {
        System.out.println('a' + 1);
}
```

Output:
98

Here, we added a character and an int, so it added the ASCII value of char 'a' i.e 97 and int 1. So, answer will be 98.

Similar logic applies to adding two chars as well, when two chars are added their codes are actually added i.e. 'a' + 'b' wil give 195.

Typecasting

1. Widening or Automatic type conversion:
   In Java, automatic type conversion takes place when the two types are compatible and size of destination type is larger than source type.

2. Narrowing or Explicit type conversion:
   When we are assigning a larger type value to a variable of smaller type, then we need to perform explicit type casting.

   Example code:

```
public static void main(String[] args) {
        int i = 100;
        long l1 = i;                //automatic type casting

        double d = 100.04;
        long l2 = (long)d;          //explicit type casting
        System.out.println(i);
```

```
        System.out.println(l1);
        System.out.println(d);
        System.out.println(l2);
}
```

Output:
100
100
100.04
100

Operators

Arithmetic operators

Arithmetic operators are used in mathematical expression in the same way that are used in algebra.

OPERATOR	DESCRIPTION
+	Adds two operands
-	Subtracts second operand from first
*	Multiplies two operands
/	Divides numerator by denominator
%	Calculates Remainder of division