##**JAVA NOTES**##

. Java is a case sensitive language
. ';' is necessary at every line end

BASIC JAVA CODE

```
public class Testing
{
	public static void main(String[] args)
	{
		System.out.println("Hello World");
	}
}
```

DATA TYPES AND VARIABLES

Types of Data Types

. byte	**.** short
. int	**.** long
. float	**.** double
. char	**.** Boolean

VARIABLES
SYNTAX TO DEFINE A VARIABLE

```
data_type variable_name=value;
```

eg: *int var=10;*

TYPECASTING
. assigning the value of one data type to another type

TYPES OF TYPECASTING:
. Widening - Automatic Type Casting: In this the compiler types casts the variable automatically**.** This typecasting is done when the user follows the following order:

byte -> short -> char -> int -> long -> float -> double

. Narrowing - Manual Type Casting: In this the user has to typecast the variable himself**.** This typecasting is done when the user follows the reverse order of the Widening typecasting'.

SYNTAX FOR NARROWING TYPECASTING:

```
data_type1 var_name1 = value;
data_type2 var_name2 = (data_type2)var_name1;
new_data_type var_name2 = (new_data_type)var_name1;
```

*** in this data_type1 is the initial data type of the variable var_name1.**
*** data_type2 is the data type in which the variable var_name1 is to be converted from data type data_type1.**
*** we write the new data type in the parenthesis '()' with the name of variable which is to be converted.**

TAKING USER INPUT
SYNTAX FOR TAKING USER INPUT

```
import java.util.Scanner;
public class ScannerUserInput
{
	public static void main(String[] args)
	{
		Scanner sc = new Scanner(System.in) //Scanner object_name = new Scanner(System.in)
		int x = sc.nextInt(); //data_type var_name = object_name.function_name();
		System.out.println(x);
	}
}
```

*** Scanner** - class imported for taking user input
*** nextInt()** - function of class Scanner**.** Takes 'integer' input
*** nextDouble()** - function of class Scanner**.** Takes 'double' input
*** nextFloat()** - function of class Scanner**.** Takes 'float' input
*** nextLine()** - function of class Scanner**.** Takes 'string' input

OPERATORS IN JAVA

ARITHMETIC OPERATORS: works on actual numbers
. + : Addition Operator **. - :** Subtraction Operator **. / :** Division Operator
. * : Multiplication Operator **. % :** Modulus Operator **. ++ :** Increment by 1
. -- : Decrement by 1

BITWISE OPERATORS: works on bits
. AND : all bits must be 1 or TRUE for AND to be TRUE**.** Represented by : &
WORKING OF BITWISE AND:
Starts comparing from the right-most bit and goes to the left-most bit.
eg: *5 -> 101*
6 -> 110
5 & 6 -> (1 & 1)(0 & 1)(1 & 0) -> 100

. NOT : Reverses the bits from 1 to 0 or 0 to 1
. OR : atleast one bit must be 1 or TRUE for OR to be TRUE**.** Represented by |
WORKING OF BITWISE OR:
Starts comparing from the right-most bit and goes to the left-most bit.
eg: *10 -> 1010*
13 -> 1101
10 | 13 -> (1 | 1)(0 | 1)(1 | 0)(0 | 1) -> 1111

. XOR :

. Right-Shift: Removes the right-most bit and shifts all the remaining bits to right and adds 0 to the left-most bit**.**
Represented by: >>**.**
Divides the number by 2.
SYNTAX:
data_type var_name2 = var_name1 >> No of times Right-shift is to be done.
eg: *applying Right-Shift on 1101 -> 0110 -> 0011 -> 0001 -> 0000*

. Left-Shift: Shifts all the bits to left and adds 0 to the right-most bit**.**
Represented by: <<**.**
Multiplies the number by 2.
SYNTAX:
data_type var_name2 = var_name1 << No of times Left-shift is to be done.
eg: *applying Left-Shift on 1101 -> 11010 -> 110100 -> 1101000*

ASSIGNMENT OPERATORS: used for assigning values to variables.
. = : assignment operator
. += : adds value to the variable before assigning it
. -= : subtracts value from the variable before assigning it
. operator= : uses the particular operator on variable with the value before assigning it

COMPARISION OPERATOR: used for comparing two operands.
. == : checks if the operands are equal or not
. < : checks if the first value is less than the second or not
. > : checks if the first value is greater than the second or not
. <= : checks if the first value is less than or equal to the second or not
. >= : checks if the first value is greater than or equal to the second or not
. != : checks if the first value is not equal to second or not

LOGICAL OPERATORS: used for logical operations.
. && : Returns TRUE when both the operands are TRUE
. || : Returns TRUE when atleast one of the operands is TRUE
. ! : Returns TRUE when both the operands are Different
***** &(bitwise AND) and &&(logical AND) both give same result. The difference between these two operators is & checks both the condition while && checks only 1 condition if the first is FALSE***.

CONDITION STATEMENTS

if STATEMENTS:

SYNTAX FOR if:

```
if(condition)
{
    //statements
}
```

if-else STATEMENTS:

SYNTAX FOR if-else:

```
if(condition)
{
    //statements
}
else
{
    //statements
}
```

elseif STATEMENTS:

SYNTAX FOR elseif:

```
if(condition1)
{
    //statements
}
else if(condition2)
{
    //statements
}
.
. n times
.
else
{
    //statements
}
```

SHORTHAND METHOD FOR if-else USING TERNARY OPERATOR
SYNTAX:

data_type var_name = (condition) ? expressionTRUE : expressionFALSE;

NESTED if-else
SYNTAX FOR NESTED if-else:

```
if(condition1)
{
    if(condition2)
    {
        //statements
    }
    else
    {
        //statements
    }
}
```

SHORTHAND METHOD FOR NESTED if-else USING TERNARY OPERATOR
SYNTAX:

data_type var_name =
(condition1)?(condition2)?expressionTRUE:expressionFALSE:(condition3)?expressionTRUE:expressionFALSE

SWITCH STATEMENTS:
SYNTAX FOR SWITCH STATEMENTS:

```
switch(var_name/expression)
{
    case value1:
    {
        //statements
        break;
    }
    .
    . n-times
    case valuen:
    {
        //statements
        break;
    }
    default:
    {
        //statements
        break;
    }
}
```

LOOPS IN JAVA
. used to execute a set of statements repeatedly until a particular condition is satisfied

TYPES OF LOOP
for LOOP
SYNTAX FOR for LOOP:

```
for(initialization; condition; increment/decrement)
{
    //statements
}
```

INFINITE LOOP:

```
for(;;)
{
    //statements
}
```

***** This loop will run for infinite number of times***.
NESTED for LOOP:
SYNTAX FOR NESTED for LOOP:

```
for(initialization; condition; increment/decrement)
{
    for(initialization; condition; increment/decrement)
    {
        //statements
    }
}
```

break STATEMENT: break is used when we want to terminate the loop before the specified condition becomes false

SYNTAX FOR break:

```
for(initialization; condition; increment/decrement)
{
    if(condition)
    {
        //statement
        break;
    }
}
```

continue STATEMENT: continue is used when we want to skip a particular iteration and move to the next part of loop

SYNTAX FOR continue:

```
for(initialization; condition; increment/decrement)
{
    if(condition)
    {
        //statement
        continue;
    }
}
```

while LOOP

. also known as indefinite loop
. used when we don't know for how much time loop is to be runned

SYNTAX FOR while LOOP:

```
initialization;
while(condition)
{
    //statements
    increment/decrement
}
```

do-while LOOP

. used when we want the loop to run atleast once.

SYNTAX FOR do-while LOOP:

```
initialization;
do
{
    //statement
    increment/decrement
} while(condition);
```

ARRAYS IN JAVA

. data structure used to store multiple data of same type into a single variable.
. it is beneficial as there is no need to declare different variable for different values.

SYNTAX FOR DECLARING ARRAY

```
data_type array_name[size];
array_name = new data_type[size];
        OR
data_type array_name[] = new data_type[size];
```

SYNTAX FOR USER INPUT ARRAY

```
data_type array_name[] = new data_type[size];
array_name[i] = sc.nextInt[];
```

2-D ARRAY

. combination of multiple 1-D arrays.
. Representation: array_name[row][col]

SYNTAX FOR DECLARING 2-D ARRAY

```
data_type array_name[][] = new data_type[row_size][col_size];
```

SORTING IN ARRAYS

BUBBLE SORT

. in this we check if the element on the right is greater than the left element or not.
. if the element on right is lesser then both the elements are swapped.
. in this sorting takes place from right to left.
. in this we get the sorted array after n-1 iterations.

eg: [3,2,-1,5,4] -> [2,3,-1,5,4] -> [2,-1,3,5,4] -> [2,-1,3,5,4] -> [2,-1,3,4,5]
^ ^ ^ ^ ^ ^ ^ ^

[2,-1,3,4,5] -> [-1,2,3,4,5] -> [-1,2,3,4,5] -> [-1,2,3,4,5] -> [-1,2,3,4,5]
^ ^ ^ ^ ^ ^ ^ ^

SELECTION SORT

. in this we first traverse the array and find the smallest element and swap it with the first element.

eg: [5,2,-1,6,3] -> [-1,2,5,6,3] -> [-1,2,5,6,3] -> [-1,2,3,6,5] -> [-1,2,3,5,6]
^ ^ * ^ ^ * * ^ ^ * * * ^ ^ * * * * *

STRINGS IN JAVA

. non-primitive data type
. used to store words and sentences.
. Representation: "String" <- string 'S' <- character

SYNTAX FOR DECLARING A STRING:

String var_name1 = "value"; ***<u><- initialization by literals</u>***

OR

String var_name2 = new String("value"); ***<u><- initializing by making object of class</u>***

<u>**initialization by literals means initializing using double quotes "".</u>

STRING INITIALIZATION BY LITERALS AND 'new'

To optimize the string usage, JAVA developers made a new memory area to store strings called the 'String pool area'**.** String pool area uses the older strings which are stored in it**.** When we initialize a string, compiler first searches for the string in the String pool area**.** If the string is found in the pool area, then the program will not create the string again**.** Instead, it will use the string which is present in the pool area**.** In other words String pool area optimizes the string usage by using the older strings which are present.

When we initialize the strings using literals, the compiler searches for the string in the pool area**.** If the string is present, then it uses the older string and does not create a new one, but if the string is not present it will create a new one and when the string is used again then it is already present in the pool area, so it will not be created.

When we initialize the strings by using 'new', it will create a new string in the heap area and will also create its copy in the string pool area**.** Its copy will only be created in the pool area if the string is not present in it**.** If the string is used again using 'new' then the string will again be created in the heap area but not in pool area.

eg*: String name1 = "Anurag";*
String name2 = "Anurag";
String name3 = new String("Anurag");

name1 == name2 -> true -> because both the variables use the string present in pool area
name1 == name3 -> false -> because name1 uses the string present in the pool area while name3 uses the string present in the heap area

FUNCTIONS IN STRING

SYNTAX:

var_name.function(arguments(if required));

. char charAt(int index) : returns the character present at the index given
. int lenght() : returns the length of the string
. String substring(int beginIndex) : returns a substring from a string
. String substring(int beginIndex, int endIndex) : returns substring from a string
. boolean contains(CharSequence s) : check if the string contains particular string or not
. boolean equals(var_name2) : check if the strings are same or not

eg: *String name1 = "Anurag";*
String name2 = new String("Anurag");

name1.equals(name2) -> true -> because it checks the values not object refrences.

. boolean isEmpty() : checks if string is empty or not
. String concat(String str) : concatenate strings
. String replace(char old, char new) : replaces the old character with new one
. String[] split(String regex) : returns the array of string splitted from a expression

eg:
```
String cars = "Hyundai,Maruti,BMW,Audi,Ferrari,Lamborghini";
String allCars[] = cars.split(",");
for(String car : allCars)
{
    System.out.println(car);
}
```

. int indexOf(int ch) : returns the first index of the particular character
. String toLowerCase() : converts the string to lower case
. String toUpperCase() : converts the string to upper case
. String trim() : removes the trailing(from front and back) spaces from the string

ANAGRAMS IN STRINGS:
. Two strings are said to be anagram of each other if they have the same length, same characters and each repeat for the same number of times.
eg: 'silent' and 'listen' are the anagram of each other.

APPROACH TO SOLVE ANAGRAMS:
1**.** check if they have same length.
2**.** check if they have same characters.
3**.** check if they have each character repeating for same no of time.