Day2: Java Language Fundamentals

Java Comment Section

Before starting the implementation part, it is a convention to provide some description about our implementation, here to provide a description about our implementation we have to use the Comment Section. The description includes the author name, Objective, project details, module details, client details,

We will use comments to provide the above-specified description in the comment section.

There are 2 types of comments.

1. Single Line Comment:
   It allows the description within a single line.
   Syntax:
   // --- description------
2. Multi-Line Comment

   It allows description in more than one line

   Syntax:

   /*

   ....

   description

   ....

   */

Java Language Fundamentals

To prepare java applications, java has provided the following list of tokens.

1. Identifiers
2. Literals
3. Keywords/ Reserved words
4. Operators

1. **Identifiers:**

The identifier is a name assigned to the programming elements like variables, methods, classes, abstract classes, interfaces etc.

ex:

```
int age = 25; int height; //default to be 0

height=67;
 //0, 1, ...

here
int : data type
age : variable [identifier]
= : operator (assingment)
25 : value/constant [literal]
; : terminator
```

Note: To provide identifiers in java programming, we have to use the following rules and regulations:-

Identifiers should not be started with any number, identifiers may be started with an alphabet, '_' *symbol, '$' symbol, but, the subsequent symbols may be a number, an alphabet,* '_' symbol, '$' symbol.

//camelCase (recommendation, convention) = thisIsMe ⇒ variable names, Method names

//PascalCase = ThisIsMe⇒ classes, interfaces, abstract classes

Identifiers are not allowing spaces in the middle.

ex:

int employeeNumber = 111; // valid

String #address = "Pune"; // Invalid

String employee-Address="Hyderabad"; // Invalid

getInputStream(); // valid
get Input Stream(); // Invalid

In java applications, it is suggestible to provide identifiers with a particular meaning.

2. **Literals**

Literal is a constant/value assigned to the variable.

To prepare java programs, JAVA has provided the following set of literal:

1. ***Integer/Integral Literals:***

byte, short, int, long :- 10, 20, 30,....
char :- 'A','B',.....

2. ***Floating Point Literals:***

   float : 10.22f, 23.345f,.....
   double :- 11.123, 456.345,....

3. ***Boolean Literals:***

   boolean:- true, false

4. ***String Literals:***

   String :- "welcome", "Hello",.....

3. Keywords/ Reserved Words

Keywords in Java convey a special meaning to the compiler therefore, these cannot be used as identifiers.

Among the list of keywords list mentioned below the keywords **goto** and **const** are currently not in use. They are reserved words (for future use)..

ex:-

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

4. Operators:

An operator is a symbol, it will perform a particular operation over the provided operands.
To prepare java applications, JAVA has provided the following list of operators.

1. Arithmetic Operators:

```
+, -, *, /, %, ++, --
```

2. Assignment Operators:

```
=, +=, -=, *=, /=, %=
```

3. Comparison Operators:

```
==, !=, <, >, <=, >=
```

4. Boolean Logical Operators:

```
&, |, ^
```

5. Bitwise Logical Operators:

```
&, |, ^, <<, >>
```

6. Short-Circuit Operators:

```
&&, ||
```

7. Ternary Operator:

```
Expr1? Expr2: Expr3;
```

I Problem:

example 1 :

```
package com.masai;
public class Main{
```

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

      int a=10;
      System.out.println(a); //10
      System.out.println(a++);//10  //post-increment => (print, increase)
      System.out.println(++a);//12
      System.out.println(a--);//12
      System.out.println(--a);//10
      System.out.println(a);//10

    }

}
```

Output:-

10

10

12

12

10

10

Example 2:

```
package com.masai;
public class Main{

    public static void main(String[] args){

      int a=5;
      System.out.println(++a - ++a);
    }
}
```

Output: -1

Example 3 :

```
package com.masai;
public class Main{

    public static void main(String[] args){

      int a=5;
      System.out.println((--a+--a)*(++a-a--)+(--a+a--)*(++a+a++));
    }
}
```

Output : 16

explanation : (4 + 3) * (4 - 4) + (2 + 2) * (2 + 2)

Example: Boolean Logical Operator

This operator can also be applied the boolean value as well as an integer.

```
A B A&B A|B A^B
-----------------------
T T  T   T   F
T F  F   T   T
F T  F   T   T
F F  F   F   F


A B A&B A|B A^B
--------------------
0 0  0   0  0
0 1  0   1  1
1 0  0   1  1
1 1  1   1  0
```

Example 1:

```
package com.masai;
public class Main{

  public static void main(String[] args{

     boolean b1=true;
     boolean b2=false;

     System.out.println(b1&b1);//true
     System.out.println(b1&b2);//false
     System.out.println(b2&b1);//false
     System.out.println(b2&b2);//false

     System.out.println(b1|b1);//true
     System.out.println(b1|b2);//true
     System.out.println(b2|b1);//true
     System.out.println(b2|b2);//false

     System.out.println(b1^b1);//false
     System.out.println(b1^b2);//true
     System.out.println(b2^b1);//true
     System.out.println(b2^b2);//false
     }
  }
```

```
package com.masai;
public class Main{
```

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

   int a=10;
   int b=2;

   System.out.println(a&b);
   System.out.println(a|b);
   System.out.println(a^b);
   System.out.println(a<<b);
   System.out.println(a>>b);
   }
 }
```

```
int a=10;// 1010
int b=2;// 0010
a&b--->10&2--> 0010-----> 2
a|b--->10|2--> 1010-----> 10
a^b--->10^2--> 1000-----> 8

a<<b ---> 10<<2 -----> 00001010
00101000---> 40
--> Remove 2 symbols at left side and append 2 0's at right side.
a>>b ---> 10>>2 -----> 00001010
00000010
--> Remove 2 symbols at right side and append 2 0's at left side.

Note: Removable Symbols may be o's and 1's but appendable symbols must be 0's.
```

Example: Short-Circuit Operators

The main intention of Short-Circuit operators is to improve java applications performance.

&&

||

| vs ||

In the case of Logical-OR operator, if the first operand value is true then it is not required to check second operand value, directly, we can predict the result of overall expression is true.

In the case of '|' operator, even first operand value is true , still, JVM evaluates second operand value then only JVM will get the result of overall expression is true, here evaluating second operand value is unnecessary, it will increase execution time and it will reduce application performance.
In the case of '||' operator, if the first operand value is true then JVM will get the overall expression result is true with out evaluating second operand value, here JVM is not evaluating second operand expression unnecessarily, it will reduce execution time and it will improve application performance.
Note: If the first operand value is false then it is mandatory for JVM to evaluate second operand value in order to get overall expression result.

Example:

```
package com.masai;
public class Main{

    public static void main(String[] args {

         int a=10;
         int b=10;

         if( (a++ == 10) | (b++ == 10) )
         {
          System.out.println(a+" "+b);//OUTPUT: 11 11
         }

         int c=10;
         int d=10;

         if( (c++ == 10) || (d++ == 10) )
         {
         System.out.println(c+" "+d);//OUTPUT: 11 10
         }
    }
}
```

& vs &&

In the case of Logical-AND operator, if the first operand value is false then it is not required to check second operand value, directly, we can predict the result of overall expression is false.
In the case of '&' operator, even first operand value is false , still, JVM evaluates second operand value then only JVM will get the result of overall expression is false, here evaluating second operand value is unnecessary, it will increase execution time and it will reduce application performance.
In the case of '&&' operator, if the first operand value is false then JVM will get the overall expression result is false with out evaluating second operand value, here JVM is no evaluating second operand expression unnecessarily, it will reduce execution time and it
will improve application performance.
Note: If the first operand value is true then it is mandatory for JVM to evaluate second operand value in order to get overall expression result.

```
package com.masai;
public class Main{

   public static void main(String[] args) {

         int a=10;
         int b=10;

         if( (a++ != 10) & (b++ != 10) )
         { }
           System.out.println(a+" "+b);//OUTPUT: 11 11


        int c=10;
        int d=10;
```

```
          if( (c++ != 10) && (d++ != 10) )
          { }
            System.out.println(c+" "+d);//OUTPUT: 11 10

}
}
```

Data Types:

Java is strictly a typed programming language, where in java applications before representing data first we have to confirm which type of data we representing. In this context, to represent type of data we have to use "data types".

In java applications , data types are able to provide the following advantages.

1. We are able to identify memory sizes to store data.

   example :

   int i=10;--> int will provide 4 bytes of memory to store 10 value.

2. We are able to identify range values to the variable to assign.

   example:

   byte b=130;---> Invalid

   byte b=125;---> Valid

Reason: 'byte' data type is providing a particular range for its variables like -128 to 127, in this range only we have to assign values to byte variables.

To prepare java applications, JAVA has provided the following data types.

A. Primitive Data types :

1. Numeric Data Types
   a. Integral data types/ Integer Data types:

   byte ------> 1 bytes ----> 0
   short------> 2 bytes-----> 0

int--------> 4 bytes-----> 0
long-------> 8 bytes-----> 0

b. Real Number Data Types:

float------> 4 bytes----> 0.0f
double-----> 8 bytes----> 0.0

2. Non-Numeric Data types:

char ---------> 2 bytes---> ' ' [single space]
boolean-------> 1 bit-----> false

B. User defined data types :

All classes, all abstract classes, all interfaces, all arrays, etc.

Wrapper classes:

Classes representation of primitive data types are called as Wrapper Classes.

Primitive Data Types	:	Wrapper Classes
byte	:	java.lang.Byte
short	:	java.lang.Short
int	:	java.lang.Integer
long	:	java.lang.Long
float	:	java.lang.Float
double	:	java.lang.Double
char	:	java.lang.Character
boolean	:	java.lang.Boolean

To identify "min value" and "max value" for each and every data type, JAVA has provided the following two constant variables from all the wrapper classes.
MIN_VALUE and **MAX_VALUE**

```
package com.masai;
public class Main{

    public static void main(String[] args){

       System.out.println(Byte.MIN_VALUE+"----->"+Byte.MAX_VALUE);
       System.out.println(Short.MIN_VALUE+"----->"+Short.MAX_VALUE);
       System.out.println(Integer.MIN_VALUE+"----->"+Integer.MAX_VALUE);
       System.out.println(Long.MIN_VALUE+"----->"+Long.MAX_VALUE);
       System.out.println(Float.MIN_VALUE+"----->"+Float.MAX_VALUE);
       System.out.println(Double.MIN_VALUE+"----->"+Double.MAX_VALUE);
       System.out.println(Character.MIN_VALUE+"----->"+Character.MAX_VALUE);
       //System.out.println(Boolean.MIN_VALUE+"----->"+Boolean.MAX_VALUE);  Error
```

```
    }
}
```

Note:- All the above wrapper classes belongs from “java.lang” package.

There are many other uses of wrapper classes as well.

Type Casting

The process of converting data from one data type to another data type is called as "Type Casting".

There are two types of primitive data types type castings.

1. Implicit Type Casting
2. Explicit Type Casting

1. **Implicit Type Casting:**

   The process of converting data from lower data type to higher data type is called as Implicit Type Casting.

   example :

```
byte b=10;
int i = b;
System.out.println(b+" "+i);
```

   Note: Type Checking is the responsibility of compiler and Type Casting is the responsibility of JVM.

```
int i=10;
byte b=i;
System.out.println(i+" "+b);

Status: Compilation Error, Possible loss of precision.
```

```
byte b=65;
char c=b;
System.out.println(b+" "+c);

Status: Compilation Error
```

```
char c='A';
int i=c;
System.out.println(c+" "+i);

Status: No Compilation Error
OUTPUT: A 65
```

Reason: char internal data representation is compatible with int.

```
byte b1=60;
byte b2=70;
byte b=b1+b2;
System.out.println(b);

Status: Compilation Error, Possible loss of precision.
```

Reason : If we perform any arithmetic operation on primitive data types variables then the result data type will be according to the following formula:

max (int, type1, type2, type3,...)

according to the above formula the result data type of the above expression will be “int”

max (int, byte, byte) ——> int

2. **Explicit Type Casting:**

   The process of converting data from higher data type to lower data type is called as Explicit Type Casting.

   To perform explicit type casting we have to use the following pattern.

   P a = (Q) b;

   (Q) → Cast operator

   Where P and Q are two primitive data types, where Q must be either same as P or lower than P .

   Example :

```
int i = 10;
byte b = (byte)i;
System.out.println(i+" "+b);
```

```
byte b1=30;
byte b2=30;
byte b=(byte)(b1+b2);
System.out.println(b);
```

```
int i=130;
byte b=(byte)i;
System.out.println(b);
// -128, -127, -126 .... 0, 1, 2..... 127
// 130 = 127+3

Status: No Compilation Error
OUTPUT: -126
```

Note:- In java any decimal point number will be by default treated as double.

example

```
float f= 10.55;

Status: Compilation Error, Possible loss of precision.

In order to downcast this double value to the float we can use either

float f = (float)10.55;
or
float f = 10.55f;
```

Java Statements:

Statement is the collection of expressions.

To design java applications JAVA has provided the following statements.

1. **General Purpose Statements**

   Declaring variables, methods, classes, etc.
   Creating objects, accessing variables, methods, etc.

2. **Conditional Statements:**

   if, if-else, switch

3. **Iterative Statements:**

   for, while, do-while

4. **Transfer statements:**

   break, continue, return

5. **Exception Handling statements:**

   try-catch-finally, throw, throws

6. **Synchronized statements:**

   synchronized method, synchronized blocks

Conditional Statements:

These statements are able to allow to execute a block of instructions under a particular condition.

example : **if, if-else, switch**

if, if-else:

Syntax1:

```
if(condition)
{
---instructions----
}
```

Syntax2:

```
if(condition)
{
---instructions----
}
else
{
----instructions----
}
```

Syntax3:

```
if(condition)
{
---instructions----
}
else if(condition)
{
```

---instruction----
}

else if(condition)
{
---instructions----
}

....

....

else
{
----instructions----
}

Examples:

```
package com.masai;
public class Main{

    public static void main(String[] args) {

          int i=10;
          int j;

          if(i==10)
          {
            j=20;
          }
          System.out.println(j);
          }
}

Status: Compilation Error, Variable j might not have been innitialized.
```

Reason:

In java applications, only class level variables are having default values, local variables are not having default values. If we declare local variables in java applications then we must provide initializations for that local variables explicitly, if we access any local variable with out having initialization explicitly then compiler will rise an error like "Variable x might not have been initialized".

```
package com.masai;
public class Main{

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

```
         int i=10;
         int j;

         if(i==10)
         {
           j=20;
         }
         else
        {
         j=30;
        }
         System.out.println(j);
    }
}

Status: No Compilation Error
OUTPUT: 20
```

```
package com.masai;
public class Main {
 public static void main(String[] args)
 {
     int i=10;
     int j;

     if(i==10)
     {
     j=20;
     }
     else if(i==20)
     {
      j=30;
     }
     else
     {
       j=40;
     }
     System.out.println(j);
  }
}

Status: no Compilation Error
OUTPUT: 20
```

switch:

'if' is able to provide single condition checking by default, but, switch is able to provide multiple conditions checking.

Syntax:-

```
switch(var)
{
```

case 1:
-----instructions-----
break;
case 2:
----instructions------
break;

case n:
----instructions-----
break;
default:
----instructions-----
break;
}

Note: We will utilize switch programming element in "Menu Driven" Applications.

```
package com.masai;
public class Main {
 public static void main(String[] args){

     int i=10;

     switch(i)
     {
       case 5:
         System.out.println("Five");
         break;
       case 10:
         System.out.println("Ten");
         break;
       case 15:
         System.out.println("Fifteen");
        break;
      case 20:
         System.out.println("Twenty");
       break;
       default:
           System.out.println("Default");
       break;
     }
   }
 }
```

Rules to write switch:

1. switch is able to allow the data types like byte, short, int ,char and String.
2. In switch, all cases and default are optional, we can write switch without cases and with default, we can write switch with cases and without default, we can write switch without both cases and default also.

Iterative Statement:

These statements are able to allow JVM to execute a set of instructions repeatedly on the basis of a particular condition.

EX: for, while, do-while

1. **for loop:**

   In general, we will utilize for loop when we aware no of iterations in advance before writing loop.

   Syntax:

   for(Expr1; Expr2; Expr3)
   {
   ----instructions-----
   }

   Example:

```
package com.masai;
public class Main {
 public static void main(String[] args) {

  for(int i=0;i<10;i++) {
     System.out.println(i);
   }
  }
}
```

```
package com.masai;
public class Main {
 public static void main(String[] args) {

   System.out.println("Before Loop");

   for(int i=0; true ;i++){

   System.out.println("Inside Loop");
 }
 System.out.println("After Loop");
 }
}


Status: Compilation Error, Unreachable Statement
```

2. **while loop:**

   In java applications, when we are not aware the no of iterations in advance before writing loop there we should utilize 'while' loop.

   Syntax:
   while(Condition)
   {
   ---instructions-----
   }

```
package com.masai;
public class Main {
 public static void main(String[] args)
 {
   int i=0;
   while(i<10)
   {
     System.out.println(i);
     i=i+1;
   }
 }
}
```

```
package com.masai;
public class Main
 {
   public static void main(String[] args)
   {
     System.out.println("Before Loop");
     while(true)
      {
        System.out.println("Inside Loop");
      }
     System.out.println("After Loop");
 }
}

Status: Compilation Error, Unreachable Statement.
```

3. **do-while loop:**

   Difference between while loop and do-while loop:

   1. While loop is not giving any guarantee to execute loop body minimum one time.do-while loop will give guarantee to execute loop body minimum one time.

2. In case of while, first, conditional expression will be executes, if it returns true then only loop body will be executed. In case of do-while loop, first loop body will be executed then condition will be executed.
3. In case of while loop, condition will be executed for the present iteration. In case of do-while loop, condition will be executed for the next iteration.

Syntax:

do
{
---instructions---
}
while(Condition);

```
package com.masai;
public class Main
 {
 public static void main(String[] args)
 {
 int i=0;
 do
 {
 System.out.println(i);
 i=i+1;
 }
 while (i<10);
 }
}
```

Example 2:

```
package com.masai;
public class Main
 {
 public static void main(String[] args)
 {
  System.out.println("Before Loop");
 do
 {
 System.out.println("Inside Loop");
 }
 while (true);
 System.out.println("After Loop");
 }
}

Status: Compilation Error, Unreachable Statement
```

Transfer Statements:

These statements are able to bypass flow of execution from one instruction to another instruction.

1. **break;**
2. **continue;**
3. **return**

1. break statement

The break statement will bypass flow of execution to outside of the loops by skipping the remaining instructions in the current iteration.

Example1:

```
package com.masai;
public class Main{

    public static void main(String[] args){

       for(int i=0;i<10;i++){

        if(i==5)
          break;
        System.out.println(i);
      }
   }
}
```

Example2:

```
package com.masai;
public class Main{

  public static void main(String[] args){

    for(int i=0; i<10; i++)// Outer loop
    {
       for(int j=0; j<10; j++)// Nested Loop
       {
        if(j==5)
           break;

        System.out.println(i+" "+j);
      } // end of nested loop
```

```
    }//end of outer loop

  }
}


OUTPUT:
0 0
0 1
0 2
0 3
0 4
-------
-------
-------
9 0
9 1
9 2
9 3
9 4
```

Note: If we provide "break" statement in nested loop then that break statement is applicable for only nested loop, it will not give any effect to outer loop.

2. continue statement

This transfer statement will bypass flow of execution to starting point of the loop by skipping all the remaining instructions in the current iteration in order to continue with next iteration.

```
package com.masai;
public class Main{

    public static void main(String[] args){

       for(int i=0;i<10;i++){

        if(i==5)
          continue;
        System.out.println(i);
      }
   }
}

OUTPUT:
0
1
2
3
4
6
7
```

```
8
9
```

Example2:

```
package com.masai;
public class Main{

    public static void main(String[] args){

      for(int i=0;i<10;i++)
      {
          if(i %2 == 0)
              continue;

          System.out.println(i);
      }
   }
}

output:
1
3
5
7
9
```

Example3:

```
package com.masai;
public class Main{

    public static void main(String[] args){

      System.out.println("before Loop");

      for(int i=0;i<10;i++)
      {
        if(i == 5)
        {
          System.out.println("Inside Loop, before continue");
          continue;
          System.out.println("Inside Loop, After continue");
        }
      }

      System.out.println("After loop");
   }
```

```
}

Status: Compilation Error, Unreachable Statement.

Reason: If we provide any statement immediately after the continue statement then,
it will be an unreachable statement, where the compiler will raise an error.
```

Reason: If we provide any statement immediately after continue statement then that statement is unreachable statement, where compiler will rise an error.

Example3:

```
package com.masai;
public class Main{

    public static void main(String[] args){

    for(int i=0;i<5;i++)
    {
      for(int j=0;j<5;j++)
      {
        if(j==2)
          continue;

        System.out.println(i+" "+j);
      }
    }

   }
}

Output:
0 0
0 1
0 3
0 4
1 0
1 1
1 3
1 4
2 0
2 1
2 3
2 4
3 0
3 1
3 3
3 4
4 0
4 1
4 3
4 4

Reason: If we provide a continue statement in the nested loop then the continue statement will give an effect
to nested loop only, it will not give effect to the outer loop.
```