Java Lexical Structure

Basic Structure of Java Programme

```
Package list;
public Class <class name>
{
    Data members;
    Constructor functions;
    User-defined methods;

    public static void main (String arguments[])
    {
        Block of statements;
    }
}
```

Computer languages, like human languages, have a lexical structure. A source code of a Java program consists of tokens. Tokens are atomic code elements. In Java we have **comments, identifiers, literals, operators, separators, and keywords**.Java programs are composed of characters from the Unicode character set.

Java comments

Comments are used by humans to clarify source code. There are three types of comments in Java.

Comment type	Meaning
// comment	Single-line comments
/* comment */	Multi-line comments
/** documentation */	Documentation comments

If we want to add some small comments we can use single-line comments.

For more complicated explanations, we can use multi-line comments.

The documentation comments are used to prepare automatically generated documentation. This is generated with the javadoc tool.

```
//Single line comment
public class CommentExample1 {
public static void main(String[] args) {
   int i=10;//Here, i is a variable
   System.out.println(i);
   }
}
```

```
//Multiline Comment
public class CommentExample2 {
public static void main(String[] args) {
/* Let's declare and
```

@Author: Er. Prabeen Soti

```
 print variable in java. */
    int i=10;
    System.out.println(i);
    }
}

/** The Calculator class provides methods to get addition and subtraction of given 2
numbers.*/
public class Calculator {
/** The add() method returns addition of given numbers.*/
public static int add(int a, int b){return a+b;}
/** The sub() method returns subtraction of given numbers.*/
public static int sub(int a, int b){return a-b;}
}
```

//Java program to illustrate frequently used

// Comment tags

/**

*** <h1>Find average of three numbers!</h1>**

*** The FindAvg program implements an application that**

*** simply calculates average of three integers and Prints**

*** the output on the screen.**

*****

*** @author Prabeen Soti**

@Author: Er. Prabeen Soti

```
* @version 1.0
* @since   2020-01-20
*/
public class FindAvg  {
   /**
   * This method is used to find average of three integers.
   * @param numA This is the first parameter to findAvg method
   * @param numB  This is the second parameter to findAvg method
   * @param numC  This is the second parameter to findAvg method
   * @return int This returns average of numA, numB and numC.
   */
   public int findAvg(int numA, int numB, int numC)
   {
      return (numA + numB + numC)/3;
   }

   /**
   * This is the main method which makes use of findAvg method.
   * @param args Unused.
   * @return Nothing.
   */

   public static void main(String args[])
   {
      FindAvg obj = new FindAvg();
```

```
        int avg = obj.findAvg(10, 20, 30);

        System.out.println("Average of 10, 20 and 30 is :" + avg);
    }
}
```

javadoc FindAvg.java

Java white space

White space in Java is used to separate tokens in the source file. It is also used to improve readability of the source code.

```
int i = 0;
```

White spaces are required in some places. For example between the int keyword and the variable name. In other places, white spaces are forbidden. They cannot be present in variable identifiers or language keywords.

```
int a=1;
int b = 2;
int c  =  3;
```

The amount of space put between tokens is irrelevant for the Java compiler. The white space should be used consistently in Java source code.

Java identifiers

Identifiers are names for variables, methods, classes, or parameters. Identifiers can have alphanumeric characters, underscores and dollar signs ($). **It is an error to begin a variable name with a number. White space in names is not permitted.**

Identifiers are case sensitive. This means that Name, name, or NAME refer to three different variables. Identifiers also cannot match language keywords.

There are also conventions related to naming of identifiers. The names should be descriptive. We should not use cryptic names for our identifiers. If the name consists of multiple words, each subsequent word is capitalized.

```
String name23;

int _col;

short car_age;
```

These are valid Java identifiers.

```
String 23name;

int %col;

short car age;
```

These are invalid Java identifiers.

The following program demonstrates that the variable names are case sensitive. Event though the language permits this, it is not a recommended practice to do.

Java literals

A *literal* is a textual representation of a particular value of a type. Literal types include boolean, integer, floating point, string, null, or character. Technically, a literal will be assigned a value at compile time, while a variable will be assigned at runtime.

```
int age = 29;

String nationality = "Hungarian";
```

Here we assign two literals to variables. Number 29 and string "Hungarian" are literals.

Java separators

A *separator* is a sequence of one or more characters used to specify the boundary between separate, independent regions in plain text or other data stream.

```
[ ]   ( )   { }   ,   ;   .   "

String language = "Java";
```

The double quotes are used to mark the beginning and the end of a string. The semicolon ; character is used to end each Java statement.

```
System.out.println("Java language");
```

Parentheses (round brackets) always follow a method name. Between the parentheses we declare the input parameters. The parentheses are present even if the method does not take any parameters. The System.out.println() method takes one parameter, a string value. The dot character separates the class name (System) from the member (out) and the member from the method name (println()).

```
int[] array = new int[5] { 1, 2, 3, 4, 5 };
```

The square brackets [] are used to denote an array type. They are also used to access or modify array elements. The curly brackets {} are used to initiate arrays. The curly brackets are also use enclose the body of a method or a class.

```
int a, b, c;
```

The comma character separates variables in a single declaration.

Java keywords

A keyword is a reserved word in Java language. Keywords are used to perform a specific task in the computer program. For example, to define variables, do repetitive tasks or perform logical operations.

Java is rich in keywords. Some are:

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**	**interface**	**static**	**var**

class	**finally**	**long**	**strictfp**	**void**
const	**float**	**native**	**super**	**volatile**
While				

Java Naming conventions

Conventions are best practices followed by programmers when writing source code. Each language can have its own set of conventions. Conventions are not strict rules; they are merely recommendations for writing good quality code. We mention a few conventions that are recognized by Java programmers. (And often by other programmers too).

- Class names begin with an uppercase letter.
- Method names begin with a lowercase letter (**CamelCase**).
- The **public** keyword precedes the **static** keyword when both are used.
- The parameter name of the main() method is called args.
- Constants are written in uppercase.
- Each subsequent word in an identifier name begins with a capital letter

Java command line arguments

Java programs can receive command line arguments. They follow the name of the program when we run it.

```
public class CommandLineArgs {

    public static void main(String[] args) {

        for (String arg : args) {
```

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

        }

    }

}
```

Command line arguments can be passed to the main() method.

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

The main() method receives a string array of command line arguments. Arrays are collections of data. An array is declared by a type followed by a pair of square brackets []. So the String[] args construct declares an array of strings. The args is an parameter to the main() method. The method then can work with parameters which are passed to it.

```
for (String arg : args) {

    System.out.println(arg);

}
```

We go through the array of these arguments with a for loop and print them to the console. The for loop consists of cycles. In this case, the number of cycles equal to the number of parameters in the array. In each cycle, a new element is passed to the arg variable from the args array. The loop ends when all elements of the array were passed. The for statement has a body enclosed by curly brackets {}. In this body, we place statements that we want to be executed in each cycle. In our case, we simply print the value of the arg variable to the terminal. Loops and arrays will be described in more detail later.

```
$ java CommandLineArgs.java 1 2 3 4 5
```

```
1

2

3

4

5
```

We provide four numbers as command line arguments and these are printed to the console.

When we launch programs from the command line, we specify the arguments right after the name of the program.

Java Data Types

A data type is a set of values and the allowable operations on those values. Java programming language is a statically typed language. It means that every variable and every expression has a type that is known at compile time.

There are two fundamental data types in Java: *primitive types* and *reference types*. Primitive types are:

- boolean
- char
- byte
- short
- int
- long
- float

- double

There is a specific keyword for each of these types in Java. Primitive types are not objects in Java. Primitive data types cannot be stored in Java collections which work only with objects. They can be placed into arrays instead.

The reference types are:

- class types
- interface types
- array types

There is also a special **NULL** type which represents a non-existing value.

Integers

Integers are a subset of the real numbers. They are written without a fraction or a decimal component. Integers fall within a set Z = {..., -2, -1, 0, 1, 2, ...} Integers are infinite.

In computer languages, integers are (usually) primitive data types. Computers can practically work only with a subset of integer values, because computers have finite capacity. Integers are used to count discrete entities. We can have 3, 4, or 6 humans, but we cannot have 3.33 humans. We can have 3.33 kilograms, 4.564 days, or 0.4532 kilometers.

Type	Size	Range
byte	8 bits	-128 to 127
short	16 bits	-32,768 to 32,767

char	16 bits	0 to 65,535
int	32 bits	-2,147,483,648 to 2,147,483,647
long	64 bits	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

Float

`float` is represented in 32 bits, with 1 sign bit, 8 bits of exponent, and 23 bits of the significand (or what follows from a scientific-notation number: $2.33728*10^{12}$; 33728 is the significand).

Double

`double` is represented in 64 bits, with 1 sign bit, 11 bits of exponent, and 52 bits of significand.

By default,

Java uses **`double`** to represent its floating-point numerals (so a literal `3.14` is typed `double`). It's also the data type that will give you a much larger number range, so I would strongly encourage its use over **`float`**.

@Author: Er. Prabeen Soti

Arrays

Array is a complex data type which handles a collection of elements. Each of the elements can be accessed by an index. All the elements of an array must be of the same data type.

```
public class ArraysEx {

    public static void main(String[] args) {

        int[] numbers = new int[5];

        numbers[0] = 3;

        numbers[1] = 2;

        numbers[2] = 1;

        numbers[3] = 5;

        numbers[4] = 6;

        int len = numbers.length;

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

            System.out.println(numbers[i]);

        }

    }

}
```

In this example, we declare an array, fill it with data and then print the contents of the array to the console.

```
int[] numbers = new int[5];
```

We create an integer array which can store up to 5 integers. So we have an array of five elements, with indexes 0..4.

```
numbers[0] = 3;

numbers[1] = 2;

numbers[2] = 1;

numbers[3] = 5;

numbers[4] = 6;
```

Here we assign values to the created array. We can access the elements of an array by the array access notation. It consists of the array name followed by square brackets. Inside the brackets we specify the index to the element that we want.

```
int len = numbers.length;
```

Each array has a length property which returns the number of elements in the array.

```
for (int i = 0; i < len; i++) {

    System.out.println(numbers[i]);

}
```

We traverse the array and print the data to the console.

```
$ java ArraysEx.java
```

```
3

2

1

5

6
```

This is the output of the program.

Some Literals Constant are

Decimal literals (Base 10) : In this form the allowed digits are 0-9.

int x = 101;

Octal literals (Base 8) : In this form the allowed digits are 0-7.

// The octal number should be prefix with 0.

int x = 0146;

Hexa-decimal literals (Base 16) : In this form the allowed digits are 0-9 and characters are a-f. We can use both uppercase and lowercase characters. As we know that java is a case-sensitive programming language but here java is not case-sensitive.

// The hexa-decimal number should be prefix

// with 0X or 0x.

int x = 0X123Face;

Binary literals : From 1.7 onward we can specify literals value even in binary form also allowed digits are 0 and 1. Literals value should be prefixed with 0b or 0B**.**

int x = 0b1111;

Boolean literals: Only two values are allowed for Boolean literals i.e. true and false.

```
boolean b = true;
```

Char literal

For char data types we can specify literals in 4 ways:

Single quote : We can specify literal to char data type as single character within single quotes.

```
char ch = 'a';
```

Char literal as Integral literal : we can specify char literal as integral literal which represents Unicode value of the character and that integral literals can be specified either in Decimal, Octal and Hexadecimal forms. But the allowed range is 0 to 65535.

```
char ch = 062;
```

Unicode Representation : We can specify char literals in Unicode representation '\uxxxx'. Here xxxx represents 4 hexadecimal numbers.

```
char ch = '\u0061';// Here \u0061 represent a.
```

Escape Sequence : Every escape character can be specified as char literals.

```
char ch = '\n';
```

Constant	Meaning
\t	Insert a tab in the text at this point.
\b	Insert a backspace in the text at this point.
\n	Insert a newline in the text at this point.
\r	Insert a carriage return in the text at this point.
\f	Insert a formatted in the text at this point.
\'	Insert a single quote character in the text at this point.
\"	Insert a double quote character in the text at this point.
\\	Insert a backslash character in the text at this point.

String literal

Any sequence of characters within double quotes is treated as String literals.

```
String s = "Hello";
```

Standard Java Naming Conventions

The list below outlines the standard Java naming conventions for each identifier type:

Packages: Names should be in lowercase. With small projects that only have a few packages it's okay to just give them simple (but meaningful!) names:

package softhover

package mycalculator

In software companies and large projects where the packages might be imported into other classes, the names will normally be subdivided. Typically this will start with the company domain before being split into layers or features:

package com.mycompany.utilities

package org.xyzcompany.application.userinterface

Classes: Names should be in CamelCase. Try to use nouns because a class is normally representing something in the real world:

class Customer

class Account

Interfaces: Names should be in CamelCase. They tend to have a name that describes an operation that a class can do:

interface Comparable

interface Enumerable

Note that some programmers like to distinguish interfaces by beginning the name with an "I":

interface IComparable

interface IEnumerable

Methods: Names should be in mixed case. Use verbs to describe what the method does: void calculateTax() string getSurname()

Variables: Names should be in mixed case. The names should represent what the value of the variable represents:

string firstName

int orderNumber.

Only use very short names when the variables are short lived, such as in for loops: for(int i=0; i<20;i++) {

//i only lives in here

}

Constants: Names should be in uppercase.

static final int **DEFAULT_WIDTH**

static final int **MAX_HEIGHT**

CamelCase: each new word begins with a capital letter. Eg: **JOptionPane**

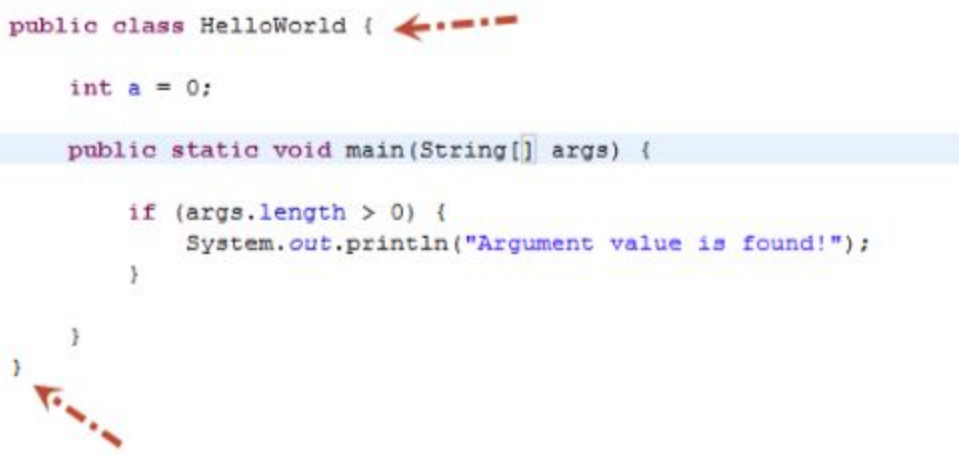
Mixed case (also known as Lower CamelCase) is the same as CamelCase except the first letter of the name is in lowercase. Eg: showMessageDialog

```
public class MyClass {
  public String firstName, lastName;

  public String fullName() {
    String name =
      firstName + " " + lastName;
    return(name);
  }
}
```

Formatting Conventions : .

You should always open a bracket at the end of a statement. You should always close the bracket at the beginning of a line.



Java operators

An *operator* is a symbol used to perform an action on some value. Operators are used in expressions to describe operations involving one or more operands.

+(addition) -(subtraction) *(multiplication) /(division) %(modulus)

&(bitwise AND) |(bitwise OR) ~(bitwise complement) ^(bitwise XOR)

=(assignment) += -= *= /= %= ^=

++(increment) --(decrement) ==(equal to)

!=(not equal to) > < (greater than and smaller than)

&= >>= <<= >= <=

||(logical or) &&(logical and) !(logical Not)

>>(right shift) <<(left shift) >>> (zero fill right shift)

?:(conditional operator)

This is a partial list of Java operators.

Java Unary Operator

The Java unary operators require only one operand. Unary operators are used to perform various operations i.e.:

- incrementing/decrementing a value by one
- negating an expression
- inverting the value of a boolean

Java Unary Operator Example: ++ and --

```
class OperatorExample{
   public static void main(String args[]){
        int x=10;
        System.out.println(x++);//10 (11)
        System.out.println(++x);//12
        System.out.println(x--);//12 (11)
        System.out.println(--x);//10
   }
}
```

Output:

```
10
12
12
10
```

Java Unary Operator Example 2: ++ and --

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

```
        int a=10;
        int b=10;
        System.out.println(a++ + ++a);//10+12=22
        System.out.println(b++ + b++);//10+11=21
    }
}
```

Output:

```
22
21
```

Java Unary Operator Example: ~ and !

```
class OperatorExample{
    public static void main(String args[]){
        int a=10;
        int b=-10;
        boolean c=true;
        boolean d=false;
        System.out.println(~a);//-11 (minus of total positive value which starts from 0)
        System.out.println(~b);//9 (positive of total minus, positive starts from 0)
        System.out.println(!c);//false (opposite of boolean value)
        System.out.println(!d);//true
    }
}
```

Output:

```
-11
9
false
true
```

Java Arithmetic Operators

Java arithmetic operators are used to perform addition, subtraction, multiplication, and division. They act as basic mathematical operations.

Java Arithmetic Operator Example

```
class OperatorExample{
    public static void main(String args[]){
        int a=10;
        int b=5;
        System.out.println(a+b);//15
        System.out.println(a-b);//5
        System.out.println(a*b);//50
        System.out.println(a/b);//2
        System.out.println(a%b);//0
    }
}
```

Output:

```
15
5
50
2
0
```

Java Left Shift Operator

The Java left shift operator << is used to shift all of the bits in a value to the left side of a specified number of times.

Java Left Shift Operator Example

```
class OperatorExample{
      public static void main(String args[]){
          System.out.println(10<<2);//10*2^2=10*4=40
          System.out.println(10<<3);//10*2^3=10*8=80
          System.out.println(20<<2);//20*2^2=20*4=80
          System.out.println(15<<4);//15*2^4=15*16=240
    }
}
```

Output:

```
40
80
80
240
```

Java Right Shift Operator

The Java right shift operator >> is used to move left operands value to right by the number of bits specified by the right operand.

Java Right Shift Operator Example

```
class OperatorExample{
   public static void main(String args[]){
       System.out.println(10>>2);//10/2^2=10/4=2
       System.out.println(20>>2);//20/2^2=20/4=5
       System.out.println(20>>3);//20/2^3=20/8=2
   }
}
```

Output:

2

5

2

Java Shift Operator Example: >> vs >>>

```
class OperatorExample{
   public static void main(String args[]){
       //For positive number, >> and >>> works same
       System.out.println(20>>2);
       System.out.println(20>>>2);
       //For negative number, >>> changes parity bit (MSB) to 0
        System.out.println(-20>>2);
        System.out.println(-20>>>2);
   }
}
```

Output:

5

5

-5

1073741819

Java AND Operator Example: Logical && and Bitwise &

The logical **&&** operator doesn't check second condition if first condition is **false**. It checks the second condition only if the first one is **true**.

The bitwise **&** operator always checks both conditions whether the first condition is **true** or **false**.

```
class OperatorExample{
   public static void main(String args[]){
      int a=10;
      int b=5;
      int c=20;
      System.out.println(a<b&&a<c);//false && true = false
      System.out.println(a<b&a<c);//false & true = false
   }
}
```

Output:

```
false
false
```

Java AND Operator Example: Logical && vs Bitwise &

```
class OperatorExample{
    public static void main(String args[]){
       int a=10;
       int b=5;
       int c=20;
       System.out.println(a<b&&a++<c);//false && true = false
       System.out.println(a);//10 because second condition is not checked
       System.out.println(a<b&a++<c);//false && true = false
       System.out.println(a);//11 because second condition is checked
   }
}
```

Output:

false

10

false

11

Java OR Operator Example: Logical || and Bitwise |

The logical || operator doesn't check second condition if first condition is true. It checks the second condition only if the first one is false.

The bitwise | operator always checks both conditions whether the first condition is true or false.

```
class OperatorExample{
    public static void main(String args[]){
        int a=10;
        int b=5;
        int c=20;
        System.out.println(a>b||a<c);//true || true = true
        System.out.println(a>b|a<c);//true | true = true
        //|| vs |
        System.out.println(a>b||a++<c);//true || true = true
        System.out.println(a);//10 because second condition is not checked
        System.out.println(a>b|a++<c);//true | true = true
        System.out.println(a);//11 because second condition is checked
    }
}
```

Output:

true

true

true

10

true

11

Java Ternary Operator

Java Ternary operator is used as one liner replacement for if-then-else statement and used a lot in java programming. it is the only conditional operator which takes three operands.

Java Ternary Operator Example

```
class OperatorExample{
      public static void main(String args[]){
        int a=2;
        int b=5;
        int min=(a<b)?a:b;
        System.out.println(min);
      }
}
```

Output:

2

Another Example:

```
class OperatorExample{
    public static void main(String args[]){
        int a=10;
        int b=5;
        int min=(a<b)?a:b;
```

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

Output:

5

Java Assignment Operator

Java assignment operator is one of the most common operators. It is used to assign the value on its right to the operand on its left.

Java Assignment Operator Example

```
class OperatorExample{
    public static void main(String args[]){
        int a=10;
        int b=20;
        a+=4;//a=a+4 (a=10+4)
        b-=4;//b=b-4 (b=20-4)
        System.out.println(a);
        System.out.println(b);
    }
}
```

Output:

14

16

Java Assignment Operator Example

```
class OperatorExample{
    public static void main(String[] args){
        int a=10;
```

```
        a+=3;//10+3
        System.out.println(a);
        a-=4;//13-4
        System.out.println(a);
        a*=2;//9*2
        System.out.println(a);
        a/=2;//18/2
        System.out.println(a);
    }
}
```

Output:

```
13
9
18
9
```