# **Theory**

**- JAVA** was developed by James Gosling at **Sun Microsystems** in the year **1995**, later acquired by Oracle Corporation.

- write once run anywhere (WORA)

- Java is the name of an **island** in Indonesia where the first coffee(named java coffee) was produced

-

# **identifiers**

- name in java program is called identifiers

- identifiers used for identification purpose

- it can be method name, class name, variable name, etc.….

- identifiers are case-sensitive

**Rules for defining identifiers**

- a-z, A-Z, 0-9, \_, $ are allowed

- identifiers can’t not start with digit (0-9)

- there is no length limit for identifiers, but recommended to use small name

- we can’t use keywords or reserved words as identifiers

- all pre-defined class names and interface names also we can use as identifiers, but not recommended to use

# **Keywords**

- keywords also called reserved words

- these words have fixed meaning

- keywords contain only lower-case alphabets

- 53 keywords are there (50 key words + 3 literals)

- 3 literals -true, false, null

- 50 keywords (48 used + 2 unused)

- 2 unused are goto, const

Keywords for Data types

- byte, short, int, long, float, double, Boolean, char

Keywords for flow control

- if, else, switch, case, default, while, do, for, break, continue, return

Keywords for modifiers

- public, private, protected, static, final, abstract, synchronized, native, strictfp, transient, volatile

Keywords for exception handling

- try, catch, finally, throw, throws, assert

Class related keywords

- class, interface, extends, implements, package, import

Object related keywords

- new, instanceof, super, this

Return type keywords

- void

Extra keyword

- enum

Unused keywords

- goto creates some problems .so, we don’t use

- const also we don’t use because we have final keyword for same purpose

- if we try to use unused keywords we get error

- null is default value for object reference

- we can use enum to define a group of named constants

# **Data types**

primitive data types

- byte, short, int, long, float, double, char, Boolean

- except char, Boolean remaining are signed data types because we can represent them in both positive, negative values

- negative numbers are stored in 2’s complement form in memory that’s why in signed bit we get one extra number in negatives when we calculate range of data type, positive numbers represents directly in memory

**- byte**

- byte (size 1Byte) (range -128 to +127)(default value=0)

- byte is mostly useful if data is in streams either from files or network

**- short**

- short(size 2Bytes)(range -32768 to +32767 or -2^15 to (2^15)-1) (default value=0)

- it is most rarely used data type in java

- short is best suitable for 16bit processors like 8085 which are old and outdated

**- int**

- int (size 4bytes)(range -2^31 to (2^31)-1 or -2147483648 to 2147483647) (default value=0)

- most commonly used data type is int

**- long**

- long (size 8bytes)(range -2^63 to (2^63)-1) (default value=0)

**- float**

- float(size 4bytes)(range -3.4e38 to 3.4e38) (default value=0.0)

- if we want 4-5 decimal places of accuracy then we use float

- it is single precision

- less accuracy

**-double**

- double(size 8bytes)(range -1.7e308 to 1.7e308) (default value=0.0)

- if we want 14-15 decimal places of accuracy then we use double

- It is double precision

- more accuracy

**- Boolean**

- Boolean (size is not applicable it based on virtual machine)(range not aplicable but allowed values are true, false) (default value=false)

**- char**

- char(size 2bytes)(range 0 to 65535) (default value=0 means space)

# **Literals**

- constant value which is assigned to a variable is called literal

**- literals for integral data types**

- if we type number prefixed with zero is considered as octal value by java compiler

Ex: int x=025;

- if we type number prefixed with 0x (or) 0X is considered as hexadecimal value by java

Ex: int x=0x25; or int x=0X25;

- In hexadecimal number 10-15 means a-f is in any case

- if we type in octal or hexadecimal while we are printing we get result only in decimal form

- if we use long as data type number is suffixed with l (or) L .it is optional

- by default every integral literal is int type

- for byte, short we assign normally

**- floating point literals**

- by default every floating point literal is double

- if we use float as data type number must suffixed with f (or) F. it is compulsory

- if we use double as data type number is suffixed with d (or) D .it is optional

- floating point literals don’t support octal, hexadecimal

**- char literals**

- we can specify this with single character within single quotes

- we use single quote for char. If we use double quote for char it give error

- char we specify using integral literal in form decimal, octal, hexadecimal

- if char output value font not installed in our system then at that time always it give ? symbol

- we can specify char value in Unicode format also. Ex: ‘\uxxxx’ , where xxxx is four digit hexadecimal number

- every escape sequence is a valid char literal

**- string literal**

- any sequence of characters within double quotes is treated as string literal

**- binary literal**

- number prefixed with 0B (or) 0b is treated as binary number by java compiler.

Ex: int x=0B111;

- allowed values are 0,1

**Usage of \_ (underscore) with integral literals**

- if we have large numbers then to reduce confusion we can use \_ symbol in between digits to separate digit places

- it is only for readability

Ex: int x=10\_01\_999; which is same as x=1001999

- at time of compilation these underscore symbols will be removed automatically

- we can use more than one underscore symbol also between the digits

Ex: int x=1\_ \_23\_54\_ \_ \_912;

- we can use \_ symbol only between digits if we use anywhere else we get error

# **Escape characters**

- \n is new line

- \t is horizontal tab

- \r is carriage return

- \b is back space

- \f is form feed

- \’ is single quote

- \” is double quote

- \\ is back slash

# **Flow controls**

- it controls in which order statements are to be executed

- types

- Selection statements

- Iterative statements

- control statements/ transfer statements

- selection statements are if-else, switch

- iterative statements are while, do-while, for, for-each

- transfer statements are break, continue, return, try-catch-finally, assert

**If-else**

- syntax

If(condition) {

Action if condition true

}

else {

Action if condition false

}

- condition should be Boolean type only

- else is optional

- curly braces are optional if there is only one line of code (if only one statement in loop)

- without curly brackets we can take one statement which should not be declarative statement

- if(true); is executes. but it doesn’t give any output. It is empty statement. We can add any no. of semicolons it doesn’t give error

- when we don’t use curly brackets in some cases. At that time else is mapped to nearest if statement

**Switch**

- switch is recommended to use in place of nested if else if

- syntax

switch(condition) {

case 1:

action-1;

break;

case 2:

action-2;

break;

case n:

action-n;

break;

default:

default action;

}

- allowed types for switch statements are byte, short, int, char, Byte, Short, Integer, Character, enum, String

- curly brackets are mandatory

- case and default both are optional

- empty switch is valid java syntax

- every statement in switch is must be in under some case

- every case label should be constant

- switch argument and case label may be expression. But case label should be constant expression

- duplicate case label not allowed

- the value in the range of switch argument type

- if we don’t use break for any label then it cause execution of next case also until break or end up of switch it is called fall-through

- if none of case matched then default case will executes

- fall-through is useful for code reusability. If we want to execute same code for multiple cases then it is useful

- within the switch we can take default case anywhere we want. but, it is recommended to write as last case

**While**

- if we don’t know no. of iterations in advance then we use while loop

- syntax

while(condition) {

statements ;

}

- condition should must be Boolean type

- without curly brackets we can able to write one statement which should not be declarative

- while(condition); is valid and it is empty while loop.

- it is entry control loop

**do-while**

- if we want to execute loop for at least once then we use this

- syntax

do {

statements;

} while(condition);

- semicolon in last is compulsory

- without curly brackets we can able to write one statement which should not be declarative

- do;

while(condition);

….it is empty do-while statement

**for loop**

- it is most commonly used loop in java

- if we know no. of iterations in advance then for loop is best choice

- syntax

for (initialization; condition; increment/decrement) {

statements;

}

- initialization, condition, increment/decrement all three parts are optional and independent of each other

- initialization part executes only once in for loop life cycle

- in initialization section we can declare any no. of variables of same type

- without curly brackets we can able to write one statement which should not be declarative

- for (initialization; condition; increment/decrement); is a valid loop. it is empty for loop

- in initialization part we can take any valid java statement

- in condition part we can take any java expression which should be of type Boolean

- in condition part if we don’t take anything then compiler keep true in that place

- in increment/decrement part we can take any valid java statement

- for ( ; ; ); is valid infinite loop

**for-each (enhanced for loop)**

- it is best choice to retrieve elements of arrays and collections

- syntax

for (type var : target)

{

statements using var;

}

- target should be iterable object

**Iterable (I)**

- iterable interface contain only one method that is iterator()

- a object is said to be iterable if and only if corresponding class implements java.lang.iterable interface

**- iterator**

- it is related to collections

- we can use to retrieve the elements of collections one by one

- it is present in java.util package

- 3 methods are there .they are hasNext(), next(), remove()

**- iterable**

- it is related to for-each loop

- the target element in for-each loop should be iterable

- it is present in java.lang package

- one method that is iterator()

**break**

- break is used in inside switch, inside loops, inside labeled blocks

- it stops execution of loops and comes out of loop

**continue**

- continue used in inside loops, loops with labels

- It skips current iteration and continue next iteration

# **Coding standards**

- when writing java code, it is highly recommended to follow coding standards

- when ever we writing any component its name should reflect the purpose of that component(functionality)

**- coding standards for classes**

- usually, class names are nouns

- class name should start with upper case character and if contains multiple words every inner word should starts with upper case character

**- coding standards for interfaces**

- usually, interface names are adjectives

- interface name should start with upper case character and if contains multiple words every inner word should starts with upper case character

**- coding standards for methods**

- usually, method names are either verbs or verb-noun combination

- method name should start with lower case character and if contains multiple words every inner word should starts with upper case character

**- coding standards for variables**

- usually, variable names are nouns

- variable name should start with lower case character and if contains multiple words every inner word should starts with upper case character

**- coding standards for constants**

- usually, constant names are nouns

- constant names should contain only upper-case characters and if it contains multiple words then these words are separated with underscore symbol

- usually, we can declare constants with public static and final modifiers

# **Types of variables**

- based on position and behavior variables are divided into three types they are instance variables, static variables, local variables

- other division types are primitive variables, reference variables

**Instance variables**

- instance variables also known as object level variables or attributes

- if value of variable is varied from object to object such type of variables is called instance variables

- for every object a separate copy of instance variables will be created

- instance variable should be declared within a class directly but outside of any method or block or constructor

- at the creation of object creation instance variables will be created and destroyed at the time of object destruction hence the scope of instance variable is exactly same as scope of object

- instance variables will be stored in the heap memory as a part of object

- we can’t access instance variable directly from static area. But we can access by using object reference

- we can access instance variable directly from instance area

- for instance variables jvm will always provide default values and we are not required to perform initialization explicitly

**Static variables**

- it is also known as class level variables or fields

- if the value of a variable is not varied from object to object then it is not recommended to declare variable as instance variable. We have to declare such type of variables at class level by using static modifier

- in the case of instance variables for every object a separate copy will be created. But, In the case of static variables a single copy will be created at class level and shared by every object of the class

- static variable should be declared within a class directly but outside of any method or block or constructor

- static variables will be created at the time of class loading and destroyed at the time of class unloading. Hence scope of static variable is exactly same as scope of .class file

- static variables will be stored in method area

- we can access static variables either by object reference or class name. but, recommended to use class name

- within the same class it is not required to use class and we can access directly

- we can access static variables directly from instance and static areas

- for static variables jvm will always provide default values and we are not required to perform initialization explicitly

**Local variables**

- sometimes to meet temporary requirement of programmer we can declare variable inside a method or block or constructor such type of variables is called local variables

- local variables also called as temporary variables or stack variables or automatic variables

- local variables will be stored inside stack memory

- local variables also known as thread safe because for every thread a separate copy of local variable will be created

- local variables will be created while executing the block in which we declared it. Once block execution completes automatically local variable will be destroyed

- scope of local variable is the block in which we declared it

- for local variables jvm don’t provide default values. Compulsory we should perform initialization explicitly before using that variable. If we are not using then it is not required to perform initialization. Only for arrays it provide default values

- it is not recommended to perform initialization for local variables inside logical blocks there is no guaranty for execution of these block always at run time

- it is highly recommended to initialization local variables at the time of declaration at least with default values

- the only applicable modifier for local variables is final . if we try to use any other modifier then we get error

- if we are not declaring with any modifier then by default it is default. but, this rule is applicable only for instance and static variables but not for local variables

**- conclusions for types of variables**

- for instance and static variables jvm will provide default values and we are not required to perform initialization explicitly. But, for local variables jvm won’t provide default values compulsory we should perform initialization explicitly before using the variable

- instance and static variables can be accessed by multiple threads simultaneously and hence these are not thread safe. But, local variables are thread safe

# **Var-args (variable arguments)**

- it is used to declare a method which can take variable number of arguments

- syntax

Method\_name (int… x) { }

Method\_name (int …x) { }

Method\_name (int…x) { }

- we can call this method by passing any number of int values including zero number

- these values stored in 1D array

- internally var-arg parameters will converted into 1D array. Hence within var-arg method we can differentiate values using index

- we can mix var-arg parameter with normal parameter

- if we mix normal parameter with var-arg parameter then var-arg parameter should be last parameter

- in side var-arg method we can take only one var-arg parameter and we can’t take more than one var-arg parameter

- inside a class we can’t declare var-arg method and corresponding 1D array method simultaneously otherwise we get error

- within normal method and var-arg method in both normal method have high priority

- in general var-arg method will get least priority. That is if no other method matched then only var-arg method will get chance

- equivalence between var-arg parameter and 1D array

- > where ever 1D array present we can replace with var-arg parameter

- > where ever var-arg parameter presents we can’t replace with 1D array

# **Main() method**

- weather class contain main method or not and weather main method is declared according to requirement or not these things won’t be checked by compiler at run time. jvm is responsible to check these things. If jvm unable to find main method then we get error

- at runtime jvm always search for main method

- order of modifiers is not important

- syntax

-> public static void main(String[ ] args)

-> public static void main(String [ ]args)

-> public static void main(String args[ ])

-> public static void main(String[ ] siva)

-> public static void main(String… args)

- the following changes are allowed

-> instead of public static we can take static public also

-> we can declare string array in any acceptable form

-> instead of args we can take any valid java identifier

-> we can replace string array with var-arg parameter

- we can also declare main method with modifiers final, synchronized, strictfp

- main method overloading is possible. But, always jvm calls main method with string array argument main method only. Other overloaded method we have to call explicitly like normal method call

- inheritance concept applicable for main method. Hence, while executing if child doesn’t contain main method then parent class main method will be executed

- it seems overriding concept applicable for main method. But, it is not overriding and it is method hiding

- for main method inheritance and overloading concepts are applicable, but overriding concept is not applicable. Instead of overriding, method hiding is applicable

**Command line arguments**

- the main objective of command line arguments is we can customize behaviour of main method

# **Operators**

- increment and decrement operators

- arithmetic operators

- string concatenation operator

- relational operator

- equality operators

- instanceof operator

- bitwise operators

- short circuit operators

- type cast operator

- assignment operators

- conditional operator

- new operator

- [ ] operator

**- increment and decrement operators**

- increment operator two types they are pre-increment operator(++x), post-increment operator(x++)

- decrement operator two types they are pre-decrement operator(--x), post-decrement operator(x--)

- we use only for variables. But, not for constants

- nesting of increment/decrement operators not allowed

- for final variable we can’t apply increment/decrement operators

- we can apply increment/decrement operators for all primitive types except Boolean

-> in case of increment/decrement operators internal type casting performed automatically

- x++ = (type of x) (x+1)

**arithmetic operators**

- arithmetic operators are +, -, \*, /, %

- when we use any arithmetic operator on two variables a and b then the result type is max(int, type of a, type of b)

- 10/0 = error because there is no infinity in int type

- 10/0.0 = infinity because infinity available in double type

- in int, long there is no way to show undefined. At this time we get error

- in float, double we have way to show undefined that is NaN (not a number)

- if we compare NaN with anything it shows false. If while comparing ! symbol is there then it shows true as result

**string concatenation operator**

- string concatenation operator is +

- the only overloaded operator in java is + operator. Sometimes It acts as arithmetic addition operator and sometimes it acts as string concatenation operator.

- if at least one argument is string type then + acts as concatenation operator and if both arguments are number type then + acts as addition operator

**Relational operators**

- relational operators are <, <=, >, >=

- we can apply relational operator for every primitive type except Boolean

- relational operators we can’t apply for object types

- nesting of relational operators not allowed in java. Otherwise we get error

**Equality operators**

- Equality operators are == , !=

- when we comparing smaller type and bigger type then smaller type automatically converted to bigger type

- we can apply Equality operators for every primitive types

- we can apply Equality operators for object types also

- if we apply Equality operators for object types then compulsory there should be some relation between argument types (either child to parent or parent to child or same type). otherwise we get error

- difference between == operator and .equals() method

- > == operator went for reference comparison and .equals() went for content comparison

- for any object reference r, r==null is always false

- null == null is always true

**Instanceof operator**

- Instanceof operator used to check weather a object is a particular type of not

**Bitwise operators**

- Bitwise operators are &, |, ^

- & is called AND operator. Returns true if both arguments are true

- | is called OR operator. Returns true if at least one argument is true

- ^ is called X-OR operator. Returns true if both arguments are different

- we can apply these operators for Boolean types and integral types also.

- if we apply for Boolean types then result is Boolean. If we apply for integral types then result is number

**bitwise complement operator (~)**

- we can apply this operator only for integral types only

- if we try to apply for Boolean type then we get error

**Boolean complement operator(!)**

- we can apply this operator only for Boolean type

**Short-circuit operators**

- Short-circuit operators are &&, ||

- while using && if first condition is false then it don’t check second condition

- these operators applicable only for Boolean type

- these almost same as bitwise operators

**Type-cast operator**

- there are two types of type casting they are implicit type casting and explicit type casting

**implicit type casting**

- compiler converts smaller type to bigger type it is called implicit type casting

- there is no loss of information in implicit type casting

- it is also known as up casting or widening

- in below cases implicit type casting possible

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

**explicit type casting**

- ex: int x= (int)y;

- when we are assigning bigger data type value to smaller data type then we require explicit type casting

- in this there maybe a loss of information

- it is also known as narrowing or down casting

- programmer is responsible to perform explicit type casting

**Assignment operator**

- assignment operator types are simple assignment, chained assignment, compound

- simple(a=10), chained(a=b=c=10), compound(a+=10)

- we can’t perform chained assignment directly at the time of declaration

- assignment operator mixed with some other operator is called compound operator

- in case of compound operator internal type casting will performed automatically

**conditional operator**

- ( )?: is called conditional operator

- it is ternary operator. It is only possible ternary operator in java

- nesting of conditional operator is possible

**new operator**

- new operator is used to create an object

- after creating an object constructor will be executed to perform initialization of object, hence constructor is not for construction of object. And it is for initialization of object

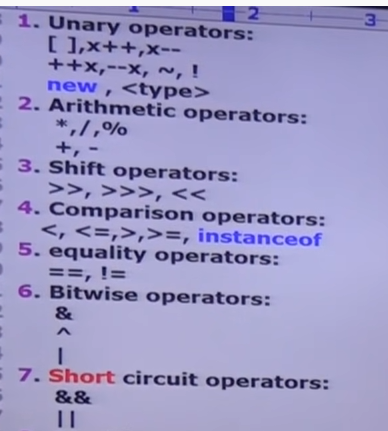
- in java we have only new keyword but not delete keyword because destruction of useless objects is the responsibility of garbage collector

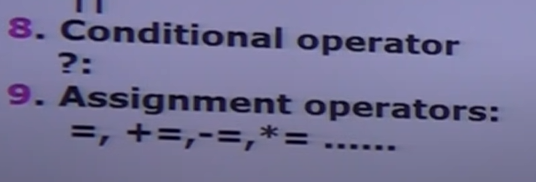
**[ ] operator**

- we can use this operator to declare and create arrays

**Operator precedence**

- unary > binary > ternary





- evaluation order of java operands

- before applying operator precedence first all operands will be evaluated from left to right

- no specific precedence for operands

# **Arrays**

- array is an indexed collection of homogeneous data elements

- every array in java is an object. So, we use new keyword to create array

- using arrays, we can represent huge number of values using single variable. It improves readability

- it is legal to have an array with size 0 in java

- we can able to specify size using data types byte, short, int, char

- array always expect size in int

- maximum allowed array size in java is 2147483647

- when we are assigning one array to another array internal elements wont be copied just reference variables will be reassigned.

- declaration of array (1D)

int[ ] x=new int[size]; -recommended

int [ ]x=new int[size];

int x[ ]=new int[size];

- declaration of array (2D)

int[ ][ ] x=new int[rowsize][columnsize];

int x[ ][ ]=new int[rowsize][columnsize];

int [ ][ ]x=new int[rowsize][columnsize];

int[ ] [ ]x=new int[rowsize][columnsize];

int[ ] x[ ]=new int[rowsize][columnsize];

int [ ]x[ ]=new int[rowsize][columnsize];

- declaration of array (3D)

int[ ][ ][ ] x=new int[size] [size] [size];

int [ ][ ][ ]x=new int[size] [size] [size];

int x[ ][ ][ ]=new int[size] [size] [size];

int[ ] [ ][ ]x=new int[size] [size] [size];

int[ ] x[ ][ ]=new int[size] [size] [size];

int[ ] [ ]x[ ]=new int[size] [size] [size];

int[ ][ ] [ ]x=new int[size] [size] [size];

int [ ][ ]x[ ]=new int[size] [size] [size];

int [ ]x[ ][ ]=new int[size] [size] [size];

- we must know size at the time of array creation

- we can’t specify size at the time of declaration. We specify size only at the time of creation

- arrays only hold homogeneous data type elements (all elements must same data type)

- when we create array every element initialize with default values

- disadvantages

- we cannot change size once array created (fixed in size)

- wastage of memory sometimes

- array initialization

- we can declare, create and initialize an array in single line

ex: int[ ] x={10,20,30};

**Array of arrays**

- it is also called jagged array In java

- array of arrays is used to reduce memory wastage in multi-dimensional arrays

- creating array of arrays using 2D

Int[ ][ ] x=new int[2][ ];

x[0] = new int[2];

x[1] = new int[3];

# **Strings**

- ways to assign value to string variable

- ex: String s = new String(“siva”); or

String s = ”siva”;

- strings are immutable

- ways to concatenate two strings

- ex: “siva”+”koti”;

“siva”.concat(“koti”);

**String methods**

- s.length();

- it returns the no.of characters In a string

- s.charAt(i);

- Returns the character at ith index.

- s.substring(i,j);

- Returns the substring from i to j-1 index.

- if we mention only one value then it give string from that index to end

- s.indexOf("va",1);

- Returns the index within the string of the first occurrence of the specified string, starting at the specified index.

- if index not mentioned then it search in whole string start to end

- s.lastIndexOf("a");

- Returns the index within the string of the last occurrence of the specified string.

- s.replace(“a”,”k”);

- Returns new string by replacing all occurrences of oldChar with newChar.

- s.toLowerCase();

- Converts all the characters in the String to lower case.

- s.toUpperCase();

- Converts all the characters in the String to upper case

- s.split(“ “);

- This method splits this string against given condition

- s.codePointAt(index);

- This method returns the Unicode value of the character at the specified index in a string.

- s1.compareTo(s2);

- method compares two strings lexicographically.

- The comparison is based on the Unicode value of each character in the strings.

- The method returns 0 if the string is equal to the other string. A value less than 0 is returned if the string is less than the other string and a value greater than 0 if the string is greater than the other string.

- s1.compareToIgnoreCase(s2);

- method compares two strings lexicographically, ignoring lower case and upper case differences.

- The comparison is based on the Unicode value of each character in the string converted to lower case.

- s1.contains(“siva”);

- method checks whether a string contains a sequence of characters.

- Returns true if the characters exist and false if not.

- s.endsWith(“va”);

- method checks whether a string ends with the specified character(s).

- s.startsWith(“si”);

- method checks whether a string starts with the specified character(s).

- s1.equals(s2);

- method compares two strings, and returns true if the strings are equal, and false if not.it do normal check only weather both value same or not

- s1.equalsIgnoreCase(s2);

- method compares two strings, ignoring lower case and upper case differences.

- s.toCharArray();

- Converts this string to a new character array

- s.trim();

- method removes whitespace from both ends of a string.

- s.isEmpty()

- method checks whether a string is empty or not.

- This method returns true if the string is empty and false if not.

- ch.valueOf()

- Returns the string representation of the specified value

- s.**getBytes();**

- method does the encoding of string into the sequence of bytes and keeps it in an array of bytes.

- s.**getChars();**

- method copies the content of this string into a specified char array. There are four arguments passed in the getChars() method. The signature of the getChars() method is given below:

- getChars(int srcBeginIndex, int srcEndIndex, char[] destination, int dstBeginIndex)

- **join();**

- method returns a string joined with a given delimiter. In the String join() method, the delimiter is copied for each element.

- join(CharSequence delimiter, CharSequence... elements);

- ex: String.join("/","25","06","2018");

# **Java source file structure**

- java program can contain any no.of classes. but at most one class can be declared as public.

- If there is a public class then name of program and name of public class must be matched otherwise we get error.

- If there is no public class then any class name we can use for java program (file name)

- whenever we are compiling a java program for every class present in the program a separate .class will be generated

- we can compiler a java program (java source file) but we can run a java .class file

- whenever we are executing a java class the corresponding class main method will be executed

- if class doesn’t contain main method then we will get error

- if the corresponding .class file not available then we will get error

- it is not recommended to declare multiple classes in a single source file

- it is highly recommended to declare only class for source file and name of program we keep same as class name. the main advantage is readability and maintainability of the code will be improved

# **Import statement**

- import statement accepts typing shortcut

- there are two types of import statements are explicit class import, implicit class import

**explicit class import**

- ex: import java.util.Scanner;

- it is highly recommended to use explicit class import because it improves readability of code

**implicit class import**

- ex: import java.util.\*;

- not recommended to use because it reduces readability of the code

- whenever we are using fully qualified name it is not required to write import statement. whenever we are writing import statement it is not required to use full qualified name

- priority of import statements

explicit class import > classes in same current working directory > implicit class import

- while resolving class names compiler will always give the precedence in the following order

explicit class import > classes in same current working directory > implicit class import

- whenever we are importing a java package all classes and interfaces present in that package bydefault available. But not sub package classes

- if we want to use sub package class compulsory we should write until sub package level

- all classes and interfaces present in the following packages are by default available to every java program. Hence, we are not required to write import statement

-> java.lang package

-> default package(current working directory)

- import statements are totally compiler time related concept. If more no.of imports then more will be the compile time. but, there is no effect on execution time(run time)s

- difference between c language #include and java import statement

-> in the case of c language #include all input output header files will be loaded at beginning only(at translation time). Hence, it is static include. But, in the case of java import statement no .class file will be loaded at the beginning. Whenever we are using a particular class then only corresponding .class file will be loaded. This is like dynamic include or load on demand or load on fly.

**static import**

- introduced In 1.5 version

- according to some usage of static import reduces length of code and improves readability. But, according to world programming experts usage of static import creates confusion and reduces readability. Hence, if there is no specific requirement then it is not recommended to use static import

- ex: import static java.lang.math.\*;

- usually we can access static members using class name but whenever we are writing static import we can access static members directly without class name

- while resolving static members compiler will always consider the precedence in the following order

-> current class static member > explicit static import > implicit static import

- two packages contains class or interface with same name is very rare and hence ambiguity problem is also very rare in normal import

- but, two classes or interfaces contain a variable or method with same name is very common and hence ambiguity problem is also very common problem in static import

- usage of static import reduces readability and creates confusion and hence if there is no specific requirement then it is not recommended to use static import

- difference between normal import and static import

-> we can use normal import to import classes and interfaces of a particular package

-> whenever we are using normal import it is not required to use fully qualified name and we can use short names directly

-> we can use static import to import static members of a particular class or interface

-> whenever we are writing static import it is not required to use class name to access static members and we can access directly

# **Packages**

- it is an encapsulation mechanism to group related classes and interfaces into a single unit, which is nothing but package

- ex1: all classes and interfaces which are required for database operations are grouped into a single package which is nothing but java.sql package

- ex2: all classes and interfaces which are useful for file io operations are grouped into a separate package which is nothing but java.io package

- the main advantages of packages are

-> to resolve naming conflicts (i.e unique identification of our components)

-> it improves modularity of the application

-> it improves maintainability of the application

-> it provides security for our components

- there is one universally accepted naming convention for package i.e. to use internet domain name in reverse

- command to run java package file is Javac -d . javafile

- -d is destination to place generated .class files

- ex: package com.siva

- generated .class file will be placed in corresponding package structure

- if the corresponding package structure not already available then this command itself will create corresponding package structure

- as destination instead od dot we can take any valid directory name ex: javac -d F: prac.java

- if the specified directory not already available then we get error

- at the time of execution we have to use full name of package ex: java com.siva.prac.java

- in any java source file there can be at most one package statement allowed. More than one package statement is not allowed otherwise we get error

- in any java program the first non-comment statement should be package statement(if it is available). Otherwise we get error

- the following is valid java source file structure

-> package statements (at most one)

Import statements (any number)

Class/enum/interfaces declaration (any number)

- an empty source file is a valid java program.

# **Class level modifiers**

- whenever we are writing our own classes we have to provide some information about our class to the jvm like

- weather this class can be accessible from anywhere or not

- weather child class creation is possible or not

- weather object creation is possible or not , etc..

We can specify this information by using appropriate modifier

- applicable modifiers for top level classes are public, default, final, abstract, strictfp

- applicable modifiers for inner level classes are public, default, final, abstract, strictfp, private, protected, static

- access specifiers vs access modifiers

- public, private, protected, default are considered as specifier except these remaining are considered as modifiers.

- But, this rule is applicable only for old languages like c++. But not in java.

- In java all are considered as modifiers only there is no word like specifiers

**public classes**

- if a class declared as public then we can access that class from anywhere

**Default classes**

- it is also known as package level access

- if a class declared as default then we can access that class only within the current package

**Final modifier**

- final is the modifier applicable for classes, methods and variables

- the main advantage we can achieve security and we can provide unique implementation

- but the main disadvantage of final keyword is we are missing key benefits of oops

- if there is no specific requirement then it is not recommended to use final keyword

Final method

- we can’t override final method

Final class

- we can’t create child class for final parent class

- if a class declared as final we can’t extend functionality of that class

- inheritance is not possible for final classes

- every method present in final class is always final method

- every variable present inside final class need not be final

**abstract modifier**

- abstract is modifier applicable for classes and method. But not for variables

- it is highly recommended to use abstract modifier. Because it promotes several oops features like inheritance and polymorphism

Abstract method

- even though we don’t know about implementation we can declare a method with abstract modifier. i.e for abstract only declaration is available but not implementation. Hence abstract method declaration should ends with semicolon.

- child class is responsible for implementation of parent class abstract method

- if a class contain at least one abstract method then we must declare class as abstract class

- if any child class extends abstract class then child class must need to implement all abstract methods available in abstract class

- by declaring abstract method in the parent class we can provide guidelines to the child classes such that which methods compulsory child class has to implements

- we can’t use final, native, synchronized, static, private, strictfp along with abstract

Abstract class

- for any java class if we are not allowed to create an object (because of partial implementation) such type of class we have to declare with abstract modifier. i.e for abstract classes instantiation is not possible

- abstract class vs abstract method

- if a class contains at least one abstract method then compulsory class as abstract. otherwise we get error (reason: if a class contains at least one abstract method implementation is not complete and hence it is not recommended to create object.to restrict object instantiation compulsory we should declare class as abstract)

- even though class doesn’t contain any abstract method still we can declare class as abstract if we don’t want instantiation. i.e abstract class can contain zero no.of abstract methods also

- if we are extending abstract class then for each and every abstract method of parent class we should provide implementation. Otherwise we have to declare child class as abstract .in this case next level child is responsible to provide implementation

- final vs abstract

- abstract class can contain final method

- final class can’t contain abstract method

**Strictfp (strict floating point)**

- we can use strictfp for classes and methods. But not for variables

- usually the result of floating point arithmetic is varied from platform to platform if we want platform independent results for floating point arithmetic then we should go for strictfp modifier

strictfp method

- if a method declared as strictfp all floating-point calculations in that method has to follow IEEE754 standard. So, that we will get platform independent results

strictfp class

- if a class declared as strictfp then every floating-point calculation present in every concrete method has to follow IEEE754 standard. So, that we will get platform independent results

- we can declare abstract strictfp combination for classes.i.e abstract strictfp combination is legal for classes. But, illegal for methods

# **Extra points**

- java is case-sensitive language

- in java return type is mandatory

- java is strongly typed programming language(data type checking is strongly checks)

- compared to other programming languages java has more object-oriented features

- 1Byte = 8bits

- by default every integral value type is int

- in java we don’t use 0 as false and other positive number as true in while loop it gives error

- java is Unicode based(**Universal Character Encoding**)

- Unicode contain 65536 characters total

- null is default value for string type

- null is default value for object reference

- null is not applicable for primitives (we can’t apply for primitives)

- integral data types are byte, short, int, long

- char is unsigned data type

- x.getClass().getName() is used to get class name of required object in java

- when we are trying to print any reference internally tostring method will be called which is implemented bydefault to return the string in the form “claasname@hashcodeinhexadecimalform”

- if we try to perform any operation on null we get error

- every final variable will be replaced by the value at compile time only

- if every argument is a final variable(compile time constant) then that operation should be performed at compile time only

- when we use arithmetic operator on any two variables a and b then the result type is max(int, type of a, type of b)

- 0/0 = undefined

- order of modifiers is not important in java

# **Questions ?**

**Arithmeticexception?**

- arithmeticexception is runtime error

- arithmeticexception not possible in floating type

- arithmeticexception is cause due to / and % only

**Java is pure object-oriented language?**

No, because it don’t support multiple inheritance, operator overloading and we depend more on primitive data types which are non-objects

**MSB?**

It acts as sign bit

It is first bit used to represent sign of number

Most significant bit, 0 means positive, 1 means negative

**Incompatible type error?**

It came when we assign other data type value

**Possible loss of precision error?**

It came when number is out of range of that type

**Arrayindexoutofbounds exception?**

We get this when we try to use index out of range

# **Pending**

-javabean

- instaceof operator

- video 23 in durga soft