Module 1: OOPS Fundamentals

Introduction: History of Java, Byte code, JVM, Java buzzwords, OOP principles, Data types, Variables, Scope and life time of variables, Operators, Control statements, Type conversion and casting, Arrays.

Concepts Of Classes And Objects: Introducing methods, Method overloading, Constructors, Constructor overloading, Usage of static with data and method, Access control, this key word, Garbage collection, String class, StringTokenizer.

1.1 Introduction:

Java is a high-level programming language and a platform. it is an object-oriented programming language. The principles for creating Java programming are "Simple, Robust, Portable, Platform-independent, Secured, High Performance, Multithreaded, Architecture Neutral, Object-Oriented, Interpreted, and Dynamic".

Types of Java Applications – There are mainly 4 types of applications that can be created using Java programming:

1) Standalone Application – Standalone applications are also known as desktop applications or window-based applications. These are traditional software that we need to install on every machine. Examples of standalone application are Media player, antivirus, etc. AWT and Swing are used in Java for creating standalone applications.

2) Web Application – An application that runs on the server side and creates a dynamic page is called a web application. Currently, Servlet, JSP, Struts, Spring, Hibernate, JSF, etc. technologies are used for creating web applications in Java.

3) Enterprise Application – An application that is distributed in nature, such as banking applications, etc. is called an enterprise application. It has advantages like high-level security, load balancing, and clustering. In Java, EJB is used for creating enterprise applications.

4) Mobile Application – An application which is created for mobile devices is called a mobile application. Currently, Android and Java ME are used for creating mobile applications.

Java Platforms / Editions – There are 4 platforms or editions of Java:

1) Java SE (Java Standard Edition) – It is a Java programming platform. It includes Java programming APIs such as java.lang, java.io, java.net, java.util, java.sql, java.math etc. It includes core topics like OOPs, String, Regex, Exception, Inner classes, Multithreading, I/O Stream, Networking, AWT, Swing, Reflection, Collection, etc.

2) Java EE (Java Enterprise Edition) – It is an enterprise platform that is mainly used to develop web and enterprise applications. It is built on

top of the Java SE platform. It includes topics like Servlet, JSP, Web Services, EJB, JPA, etc.

3) Java ME (Java Micro Edition) **–** It is a micro platform that is dedicated to mobile applications.

4) JavaFX **–** It is used to develop rich internet applications. It uses a lightweight user interface API.

1.1.1 History of Java

Java was developed by Sun Microsystems (which is now the subsidiary of Oracle) in the year 1995. James Gosling is known as the father of Java. Before Java, its name was Oak.

Since Oak was already a registered company, so James Gosling and his team changed the name from Oak to Java.

Java was originally designed for interactive television, but it was too advanced technology for the digital cable television industry at the time.

Java Version History **–** After 1995 many JDK versions are released. Each and every updated version has enhanced features when compared with previous version. The list versions release are as follows.

JDK Alpha and Beta (1995)
JDK 1.0 (23rd Jan 1996)
JDK 1.1 (19th Feb 1997)
J2SE 1.2 (8th Dec 1998)
J2SE 1.3 (8th May 2000)
J2SE 1.4 (6th Feb 2002)
J2SE 5.0 (30th Sep 2004)
Java SE 6 (11th Dec 2006)
Java SE 7 (28th July 2011)
Java SE 8 (18th Mar 2014)
Java SE 9 (21st Sep 2017)
Java SE 10 (20th Mar 2018)
Java SE 11 (September 2018)
Java SE 12 (March 2019)
Java SE 13 (September 2019)
Java SE 14 (Mar 2020)
Java SE 15 (September 2020)
Java SE 16 (Mar 2021)
Java SE 17 (September 2021)
Java SE 18 (to be released by March 2022)

1.1.2 Byte code

A byte code acts as an intermediate code present between a machine code and a source code. A byte code is basically a low-level code that results from the compilation of source code that might be present in a high-level language. A virtual machine such as a JVM (Java Virtual Machine) processes a byte code. A byte code is also sometimes known as a portable code.

1.1.3 JVM – Java Virtual Machine

It is a specification that provides a runtime environment in which Java bytecode can be executed. The JVM performs the following main tasks:

- Loads code
- Verifies code
- Executes code
- Provides runtime environment

JVM provides definitions for Memory area, Class file format, Register set, Garbage-collected heap, Fatal error reporting etc.

1.1.4 Java buzzwords

The list of java features is known as Java buzzwords. These buzzwords are listed as follows.

1. ***Simple*** **–** Java is very easy to learn, and its syntax is simple, clean and easy to understand. Java has removed many complicated and rarely-used features, for example, explicit pointers, operator overloading, etc. There is no need to remove unreferenced objects because there is an Automatic Garbage Collection in Java.
2. ***Object-oriented*** **–** Java is an object-oriented programming language. Everything in Java is an object. Object-oriented means we organize our software as a combination of different types of objects that incorporate both data and behavior.
3. ***Platform Independent*** **–** Java provides a software-based platform. It has two components: Runtime Environment and API(Application Programming Interface). Java code can be executed on multiple platforms, for example, Windows, Linux, Sun Solaris, Mac/OS, etc. Java code is compiled by the compiler and converted into bytecode. This bytecode is a platform-independent code because it can be run on multiple platforms, i.e., Write Once and Run Anywhere (WORA).
4. ***Secured*** **–** Java is best known for its security. With Java, we can develop virus-free systems. Java is secured because it has

   No explicit pointer

   Java Programs run inside a virtual machine sandbox

   Class loader - Classloader in Java is a part of the Java Runtime Environment (JRE) which is used to load Java classes into the Java Virtual Machine dynamically. It adds security by separating the package for the classes of the local file system from those that are imported from network sources.

   Bytecode verifier - It checks the code fragments for illegal code that can violate access rights to objects.

   Security manager - It determines what resources a class can access such as reading and writing to the local disk.
5. ***Robust*** **–** Java is robust because It uses strong memory management. There is a lack of pointers that avoids security problems, Java provides automatic garbage collection which runs on the Java Virtual Machine to get rid of objects which are not being used by a Java application

anymore. There are exception handling and the type checking mechanism in Java.

6. ***Architecture neutral –*** Java is architecture neutral because there are no implementation dependent features, for example, the size of primitive types is fixed.
7. ***Portable –*** Java is portable because it facilitates you to carry the Java bytecode to any platform. It doesn't require any implementation.
8. ***Interpreted and High performance –*** Java is faster than other traditional interpreted programming languages because Java bytecode is "close" to native code.
9. ***Distributed –*** Java is distributed because it facilitates users to create distributed applications in Java. RMI and EJB are used for creating distributed applications. This feature of Java makes us able to access files by calling the methods from any machine on the internet.
10. ***Multi-threaded –*** A thread is like a separate program, executing concurrently. We can write Java programs that deal with many tasks at once by defining multiple threads. The main advantage of multi-threading is that it doesn't occupy memory for each thread. It shares a common memory area. Threads are important for multi-media, Web applications, etc.
11. ***Dynamic –*** Java is a dynamic language. It supports the dynamic loading of classes. It means classes are loaded on demand. It also supports functions from its native languages, i.e., C and C++. Java supports dynamic compilation and automatic memory management (garbage collection).

1.1.5 OOP principles

OOP stands for “Object-Oriented Programming”. OOP is a programming paradigm in which every program is follows the concept of object. In other words, OOP is a way of writing programs based on the object concept. The object-oriented programming paradigm has the following core concepts.

Encapsulation – Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse. One way to think about encapsulation is as a protective wrapper that prevents the code and data from being arbitrarily accessed by other code defined outside the wrapper. Access to the code and data inside the wrapper is tightly controlled through a well-defined interface.

Inheritance – Inheritance is the process by which one object acquires the properties of another object. This is important because it supports the concept of hierarchical classification.

Polymorphism – Polymorphism (from Greek, meaning “many forms”) is a feature that allows one interface to be used for a general class of actions. The specific action is determined by the exact nature of the situation.

Abstraction – An essential element of object-oriented programming is abstraction. A powerful way to manage abstraction is through the use of hierarchical classifications. This allows you to layer the semantics of complex systems, breaking them into more manageable pieces.

Class - *Collection of objects* is called class. It is a logical entity. A class can also be defined as a blueprint from which you can create an individual object. Class doesn't consume any space.

object - Any entity that has state and behavior is known as an object. For example, a chair, pen, table, keyboard, bike, etc. It can be physical or logical. An Object can be defined as an instance of a class. An object contains an address and takes up some space in memory. Objects can communicate without knowing the details of each other's data or code. The only necessary thing is the type of message accepted and the type of response returned by the objects.

1.1.6 Data types

Java defines eight primitive/simple types of data: byte, short, int, long, char, float, double, and boolean.

- Integers This group includes byte, short, int, and long, which are for whole-valued signed numbers.

Name	Width	Range
long	64	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
int	32	–2,147,483,648 to 2,147,483,647
short	16	–32,768 to 32,767
byte	8	–128 to 127

- Floating-point numbers This group includes float and double, which represent numbers with fractional precision.

Name	Width	Range
double	64	4.9e–324 to 1.8e+308
float	32	1.4e–045 to 3.4e+038

- Characters This group includes char, which represents symbols in a character set, like letters and numbers.

Name	Width	Range
char	16	0 to 65,536

- Boolean This group includes boolean, which is a special type for representing true/false values.

1.1.7 Variables

The variable is the basic unit of storage. A variable is defined by the combination of an identifier, a type, and an optional initializer. In addition, all variables have a scope, which defines their visibility, and a lifetime.

Syntax: type identifier [= value][, identifier [= value] ...];

Example: int a=2,b=8;

1.1.8 Scope and life time of variables

A block defines a scope. Thus, each time you start a new block, you are creating a new scope. A scope determines what objects are visible to other parts of your program. It also determines the lifetime of those objects.

Variables declared inside a scope are not visible (that is, accessible) to code that is defined outside that scope. Thus, when you declare a variable within a scope, you are localizing that variable and protecting it from unauthorized access and/or modification. Indeed, the scope rules provide the foundation for encapsulation.

Scopes can be nested. For example, each time you create a block of code, you are creating a new, nested scope. When this occurs, the outer scope encloses the inner scope. This means that objects declared in the outer scope will be visible to code within the inner scope. However, the reverse is not true. Objects declared within the inner scope will not be visible outside it.

In Java there are two major scopes are defined which are as follows:

Scope defined by a class – These variables must be declared inside class (outside any method). They can be directly accessed anywhere in class.

Scope defined by a method – Scope of a variable in a method begins with its opening curly brace '{'and ends with corresponding close curl brace '}'. However, if that method has parameters, they too are included within the method's scope.

Within a block, variables can be declared at any point, but are valid only after they are declared. Thus, if you define a variable at the start of a method, it is available to all of the code within that method. Conversely, if you declare a variable at the end of a block, it is effectively useless, because no code will have access to it.

Variables are created when their scope is entered, and destroyed when their scope is left. This means that a variable will not hold its value once it has gone out of scope. Therefore, variables declared within a method will not hold their values between calls to that method. Also, a variable declared within a block will lose its value when the block is left. Thus, the lifetime of a variable is confined to its scope.

Example:

// Demonstrating Scope of a Variable

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

```
int x;  //Accessible to all statements in main();
x=10;
if(x==10)
{
//starts new scope
int y=20;  // Accessible to this block
System.out,println("x and y is :"+x+" "+y);
x=y*2;
}
//y=100;    //Error, y can't be accessed here
Sysytem.out.println("x is "+x);
}
}
```

Output:

x and y is 10 and 20

// Demonstrating Lifetime of a variable

```
class DemoLifeTime
{
public static void main(String args[])
{
int x;
for(x=0;x<3;x++)
{
int y=-1;
System.out.println(""y is "+y);
y=100;
System.out.println("y now is "+y);
}
}
}
```

Output:

y is -1

y now is 100

y is -1

y now is 100

```
y is -1
y now is 100
//Demonstrating Error of a Variable
class DemoError
{
public static void main(String args[])
{
int a=10;
{
int a=20;  //Error, variable a can't be declared again
System.out.println("a is"+a);
}
}
}
```

Output:

Error

1.1.8.1 Types of Variables

There are three types of Variables. They are:

1. Instance variable
2. Static Variable
3. Local Variable

1) Instance Variable

A variable declared inside the class but outside the body of the method, is called an instance variable. It is not declared as static. It is called an instance variable because its value is instance-specific and is not shared among instances.

Example:

```
class Test
{
int i;
public static void main(String args[])
{
Test t=new Test();
System.out.println(t.i);
}
}
```

Output:

0

2) Static variable

A variable that is declared as static is called a static variable. It cannot be local. You can create a single copy of the static variable and share it among all the instances of the class. Memory allocation for static variables happens only once when the class is loaded in the memory.

Example:

```
class Test
{
int i;
static int j;
public static void main(String args[])
{
Test t=new Test();
t.i=10;
Test.j=20;
System.out.println("i="+i);
System.out.println("j="+Test.j);
}
}
```

Output:

```
i=10
j=20
```

3) Local Variable

A variable declared inside the body of the method is called local variable. You can use this variable only within that method and the other methods in the class aren't even aware that the variable exists. A local variable cannot be defined with "static" keyword.

Example:

```
class Test
{
public static void main(String args[])
{
int i;
i=10;
System.out.println(i);
}
}
```

Output:

```
10
```

Example:

```
// Using Three Types of Variables
class Test
{
int a=10;
static int b=20;
```

```
public static void main(String args[])
{
int c=30;
Test obj=new Test();
System.out.println("Instance variable is:"+obj.a);
System.out.println("Static variable is:"+Test.b);
System.out.println("Local variable is:"+c);
}
}
```

Output:

Instance variable is 10

Static variable is 20

Local variable is 30

1.1.9 Operators

Java provides a rich operator environment. Most of its operators can be divided into the following four groups***: arithmetic, bitwise, relational, and logical***.

Arithmetic Operators –

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following table lists the arithmetic operators:

Symbol	Operator Name	Example
+	Addition	A + B
-	Subtraction	A – B
*	Multiplication	A * C
/	Division	A / B
%	Modulus	A % B
++	Increment	A ++
--	Decrement	A --
+=	Addition assignment	A += 2
-=	Subtraction assignment	A -= 2
*=	Multiplication assignment	A *= 2
/=	Division Assignment	A /= 2
%=	Modulus assignment	A %= 2

The operands of the arithmetic operators must be of a numeric type. You cannot use them on boolean types, but you can use them on char types, since the char type in Java is, essentially, a subset of int.

The Bitwise Operators –

Java defines several bitwise operators that can be applied to the integer types, long, int, short, char, and byte. These operators act upon the individual bits of their operands. They are summarized in the following table:

Symbol	Operator Name	Example
~	Bitwise unary NOT	~ A
&	Bitwise AND	A & B
|	Bitwise OR	A | B
^	Bitwise exclusive OR	A ^ B
>>	Right shift	A >> B
<<	Left shift	A << B
&=	Bitwise AND assignment	A &= 2
|=	Bitwise OR assignment	A |= 2
^=	Bitwise exclusive OR assignment	A ^= 2
>>=	Right shift assignment	A >>= 2
<<=	Left shift assignment	A <<= 2
>>>=	Shift right zero fill assignment	A >>>= 2

The following table shows the outcome of each operation

A	B	A & B	A | B	A ^ B	~ A
1	0	0	1	1	0
1	1	1	1	0	0
0	0	0	0	0	1
0	1	0	1	1	1

Relational Operators –

The relational operators determine the relationship that one operand has to the other. Specifically, they determine equality and ordering. The relational operators are shown here:

Symbol	Operator Name	Example
= =	Equal to	A = = B
!=	Not Equal to	A != B
>	Greater than	A > B
<	Less than	A < B
>=	Greater than or Equal to	A >= B
<=	Less than or Equal to	A <= B

The outcome of these operations is a boolean value. The relational operators are most frequently used in the expressions that control the if statement and the various loop statements. Any type in Java, including integers, floating-point numbers, characters, and Booleans can be compared using the equality test, ==, and the inequality test, !=. Only numeric types can be compared using the ordering operators. That is, only integer, floating-point,

and character operands may be compared to see which is greater or less than the other.

Boolean Logical Operators –

The Boolean logical operators shown here operate only on boolean operands. All of the binary logical operators combine two boolean values to form a resultant boolean value.

Symbol	Operator Name	Example
!	Logical unary NOT	! A
&&	Logical AND	A && B
||	Logical OR	A || B

Ternary Operators –

Java Ternary operator(?:) is used as one line replacement for if-then-else statement and used a lot in Java programming. It is the only conditional operator which takes three operands.

Operator precedence:

( ) [ ] .
++ -- ~ !
* / %
+ -
>> >>> <<
> >= < <=
= = !=
&
^
|
&&
||
?:
= op=

1.1.10 Control statements

Java's program control statements can be put into the following categories: selection, iteration, and jump.

Selection statements allow your program to choose different paths of execution based upon the outcome of an expression or the state of a variable.

Iteration statements enable program execution to repeat one or more statements (that is, iteration statements form loops).

Jump statements allow your program to execute in a nonlinear fashion.

Selection Statements – These statements allow you to control the flow of your program's execution based upon conditions known only during run time. Java supports two selection statements: *if* and *switch*.

The if *Statement*:

It is conditional branch statement. It can be used to route program execution through two different paths.

Syntax:

```
if (condition) statement1;
else statement2;
```

Here, each statement may be a single statement or a compound statement enclosed in curly braces (that is, a block). The condition is any expression that returns a boolean value. The else clause is optional.

The if works like this: If the condition is true, then *statement1* is executed. Otherwise, *statement2* (if it exists) is executed. In no case will both statements be executed.

Nested ifs –

A nested if is an if statement that is the target of another if or else. Nested ifs are very common in programming. When you nest ifs, the main thing to remember is that an else statement always refers to the nearest if statement that is within the same block as the else and that is not already associated with an else. Here is an example:

```
if(i == 10) {
if(j < 20) a = b;
if(k > 100) c = d; // this if is
else a = c; // associated with this else
}
else a = d; // this else refers to if(i == 10)
```

The if-else-if Ladder –

A common programming construct that is based upon a sequence of nested ifs is the *if-else-if ladder*.

Syntax:

```
if(condition)
    statement;
else if(condition)
    statement;
else if(condition)
    statement;
.
.
.
else
    statement;
```

The *if* statements are executed from the top down. As soon as one of the conditions controlling the *if* is true, the statement associated with that if is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final else statement will be executed. The

final else acts as a default condition; that is, if all other conditional tests fail, then the last else statement is performed. If there is no final else and all other conditions are false, then no action will take place.

switch –

The switch statement is Java's multiway branch statement. It provides an easy way to dispatch execution to different parts of your code based on the value of an expression. As such, it often provides a better alternative than a large series of if-else-if statements.

Syntax:

```
switch (expression) {
    case value1:
    // statement sequence
    break;
    case value2:
    // statement sequence
    break;
    .
    .
    .
    case valueN:
    // statement sequence
    break;
    default:
    // default statement sequence
}
```

The expression must be of type byte, short, int, or char; each of the values specified in the case statements must be of a type compatible with the expression. (An enumeration value can also be used to control a switch statement. Each case value must be a unique literal (that is, it must be a constant, not a variable). Duplicate case values are not allowed.

The switch statement works like this: The value of the expression is compared with each of the literal values in the case statements. If a match is found, the code sequence following that case statement is executed. If none of the constants matches the value of the expression, then the default statement is executed.

However, the default statement is optional. If no case matches and no default is present, then no further action is taken. The break statement is used inside the switch to terminate a statement sequence. When a break statement is encountered, execution branches to the first line of code that follows the entire switch statement. This has the effect of "jumping out" of the switch.

The break statement is optional. If you omit the break, execution will continue on into the next case. It is sometimes desirable to have multiple cases without break statements between them.

Nested switch Statements –

You can use a switch as part of the statement sequence of an outer switch. This is called a nested switch. Since a switch statement defines its own block, no conflicts arise between the case constants in the inner switch and those in the outer switch.

Note:

- ✓ The switch differs from the if in that switch can only test for equality, whereas if can evaluate any type of Boolean expression. That is, the switch looks only for a match between the value of the expression and one of its case constants.
- ✓ No two case constants in the same switch can have identical values. Of course, a switch statement and an enclosing outer switch can have case constants in common.
- ✓ A switch statement is usually more efficient than a set of nested ifs

Iteration Statements –

Java's iteration statements are ***for, while,*** and ***do-while***. These statements create what we commonly call loops. A loop repeatedly executes the same set of instructions until a termination condition is met.

while –

The while loop is Java's most fundamental loop statement. It repeats a statement or block while its controlling expression is true.

Syntax:

```
while(condition) {
    // body of loop
}
```

The condition can be any Boolean expression. The body of the loop will be executed as long as the conditional expression is true. When condition becomes false, control passes to the next line of code immediately following the loop. The curly braces are unnecessary if only a single statement is being repeated.

The body of the while (or any other of Java's loops) can be empty. This is because a null statement (one that consists only of a semicolon) is syntactically valid in Java.

do-while –

As you just saw, if the conditional expression controlling a while loop is initially false, then the body of the loop will not be executed at all. However, sometimes it is desirable to execute the body of a loop at least once, even if the conditional expression is false to begin with. In other words, there are times when you would like to test the termination expression at the end of the loop rather than at the beginning. Fortunately, Java supplies a loop that does just that: the do-while. The do-while loop always executes its body at least once, because its conditional expression is at the bottom of the loop

Syntax:

```
do {
// body of loop
} while (condition);
```

Each iteration of the do-while loop first executes the body of the loop and then evaluates the conditional expression. If this expression is true, the loop will repeat. Otherwise, the loop terminates. As with all of Java's loops, condition must be a Boolean expression.

The do-while loop is especially useful when you process a menu selection, because you will usually want the body of a menu loop to execute at least once.

The for Loop:

There are two forms of the for loop –

General for loop: It is the traditional form that has been in use since the original version of Java.

Syntax:

```
for(initialization; condition; iteration) {
// body
}
```

Working *for* loop: When the loop first starts, the initialization portion of the loop is executed. Generally, this is an expression that sets the value of the loop control variable, which acts as a counter that controls the loop. It is important to understand that the initialization expression is only executed once. Next, condition is evaluated. This must be a Boolean expression. It usually tests the loop control variable against a target value. If this expression is true, then the body of the loop is executed. If it is false, the loop terminates. Next, the iteration portion of the loop is executed. This is usually an expression that increments or decrements the loop control variable. The loop then iterates, first evaluating the conditional expression, then executing the body of the loop, and then executing the iteration expression with each pass. This process repeats until the controlling expression is false.

For-Each Version:

A foreach style loop is designed to cycle through a collection of objects, such as an array, in strictly sequential fashion, from start to finish. The for-each style of for is also referred to as the *enhanced for loop*.

Syntax: for(type itr-var : collection) statement-block

Here, *type* specifies the type and *itr-var* specifies the name of an iteration variable that will receive the elements from a collection, one at a time, from beginning to end. The collection being cycled through is specified by collection.

Nested Loops – one loop may be inside another.

Jump Statements –

Java supports three jump statements: break, continue, and return. These statements transfer control to another part of your program.

break –

In Java, the break statement has three uses.

- It terminates a statement sequence in a switch statement.
- It can be used to exit a loop –
  By using break, you can force immediate termination of a loop, bypassing the conditional expression and any remaining code in the body of the loop. When a break statement is encountered inside a loop, the loop is terminated and program control resumes at the next statement following the loop
- It can be used as a "civilized" form of goto –
  the break statement can also be employed by itself to provide a "civilized" form of the goto statement. Java does not have a employed goto statement because it provides a way to branch in an arbitrary and unstructured manner.

  Syntax: break *label*;

continue –

Sometimes it is useful to force an early iteration of a loop. That is, you might want to continue running the loop but stop processing the remainder of the code in its body for this particular iteration. This is, in effect, a *goto* just past the body of the loop, to the loop's end. The continue statement performs such an action. In *while* and *do-while* loops, a continue statement causes control to be transferred directly to the conditional expression that controls the loop. In a *for* loop, control goes first to the iteration portion of the for statement and then to the conditional expression. For all three loops, any intermediate code is bypassed

return –

The return statement is used to explicitly return from a method. That is, it causes program control to transfer back to the caller of the method. At any time in a method the return statement can be used to cause execution to branch back to the caller of the method. Thus, the return statement immediately terminates the method in which it is executed.

1.1.11 Type conversion and casting

Conversion of data from one type to another data type. In Java there are two ways to convert the data which are as follows:

Type conversion –

When smaller type of data is assigned to larger type of variable, an automatic type conversion will take place if the following two conditions are meet:

- ✓ The two types are compatible.
- ✓ The destination type is larger than the source type.

When these two conditions are met, a *widening* conversion takes place. For example, the int type is always large enough to hold all valid byte values, so no explicit cast statement is required. For widening conversions, the numeric types, including integer and floating-point types, are compatible with each other. However, there are no automatic conversions from the numeric types to char or boolean. Also, char and boolean are not compatible with each other.

Example:

```
class TypeConversion
{
public static void main(String args[])
{
int a=10;
long b=a;
float c=a;
System.out.println("a+" "+b+" "+c);
}
}
```

Output:

10

10

10.0

Type casting –

The process of converting larger data types into smaller data types.

To create a conversion between two incompatible types, you must use a cast. A cast is simply an explicit type conversion. This kind of conversion is sometimes called a narrowing conversion, since you are explicitly making the value narrower so that it will fit into the target type.

Syntax: (target-type) value

Here, target-type specifies the desired type to convert the specified value to.

Example:

```
int a;
byte b;
b = (int) a
```

A different type of conversion will occur when a floating-point value is assigned to an integer type: *truncation*. As you know, integers do not have fractional components. Thus, when a floating-point value is assigned to an integer type, the fractional component is lost. For example, if the value 1.23

is assigned to an integer, the resulting value will simply be 1. The 0.23 will have been truncated.

Note: if the size of the whole number component is too large to fit into the target integer type, then that value will be reduced modulo the target type's range.

Example:

```
class TypeCasting
{
public static void main(String args[])
{
double m=71.5;
int a=(int)m;
System.out.println(a);
}
}
```

Output:

71

1.1.12 Arrays

An array is a group of like-typed variables that are referred to by a common name. Arrays of any type can be created and may have one or more dimensions. A specific element in an array is accessed by its index. Arrays offer a convenient means of grouping related information.

One-Dimensional Arrays –

A one-dimensional array is, essentially, a list of like-typed variables. To create an array, you first must create an array variable of the desired type.

Syntax: type var-name[ ] = new type[size];

Example:

```
import java.lang.*;
import java.util.*;
class OnedArray
{
public static void main(String args[])
{
Scanner sc=new Scanner(System.in);
System.out.println("enter size of array");
int n;
```

```
n=sc.nextInt();
int a[]=new int[n];
System.out.println("enter elements of an array");
int i;
for(i=0;i<n;i++)
{
a[i]=sc.nextInt();
System.out.println("Elements are");
for(int j:a)
{
System.out.println(j+" ");
}
}
}
```

Output:

```
enter size of array 3
enter elements of an array 20 30 40
Elements are 20 30 40
```

Multidimensional Arrays –

In Java, multidimensional arrays are actually arrays of arrays. These, as you might expect, look and act like regular multidimensional arrays. However, as you will see, there are a couple of subtle differences. To declare a multidimensional array variable, specify each additional index using another set of square brackets.

Syntax: type var-name[][] = new type[r_size][c_size];

Example:

```
import java.lang.*;
import java.util.*;
class TwodArray
{
public static void main(String args[])
{
Scanner sc=new Scanner(System.in);
System.out.println("enter row size & col size of array");
int r,c;
r=sc.nextInt();
```

```
c=sc.nextInt();
int a[][]=new int[r][c];
System.out.println("enter elements of an array");
int i,j;
for(i=0;i<r;i++)
{
for(j=0;j<r;j++)
{
a[i][j]=sc.nextInt();
}
}
System.out.println("Elements are");
for(i=0;i<r;i++)
{
for(j=0;j<r;j++)
{
System.out.print(a[i][j]+" ");
}
System.out.println(" ");
}
}
}
```

Output:

```
enter row size & col size of array 2 2
enter elements of an array 20 30 40 50
Elements are
20 30
40 50
```

1.2 Concept of Classes and Objects:

Class forms the basis for OO programming in Java. It is a logical construct upon which entire Java language built. It defines both shape and nature of an object. It defines a new data type. class is a template for an object, and an object is an instance of a class.

A *class* definition itself contains, its exact form and nature. You do this by specifying the *data* that it contains and the *code* that operates on that data. While very simple *classes* may contain only *code* or only *data*, most real-world

classes contain both. A *class* code defines the interface to its *data*. A *class* is declared by use of the ***class*** keyword. A simplified general form of a *class* definition is shown here:

```
class classname {
        type instance-variable1;
        type instance-variable2;
        type instance-variableN;
        type methodname1(parameter-list) {
          // body of method
        }
        type methodname2(parameter-list) {
          // body of method
        }
        // ...
        type methodnameN(parameter-list) {
          // body of method
        }
}
```

The data, or variables, defined within a class are called instance variables. The code is contained within methods. Collectively, the methods and variables defined within a class are called members of the class.

Variables defined within a class are called instance variables because each instance of the class (that is, each object of the class) contains its own copy of these variables. Thus, the data class for one object is separate and unique from the data for another.

Creation of a class is nothing but creating a new data type. This data type can be used to declare objects of that type. However, obtaining objects of a class is a two-step process. First, a variable of that class type must be declared. This variable does not define an object. Instead, it is simply a variable that can refer to an object. Second, to acquire an actual, physical copy of the object and assign it to that variable that can be done by using the ***new*** operator. The ***new*** operator dynamically allocates (that is, allocates at run time) memory for an object and returns a reference to it. This reference is, more or less, the address in memory of the object allocated by new. This reference is then stored in the variable. Thus, in Java, all class objects must be dynamically allocated.

Syntax: <Class_name> <Reference_variable> = ***new*** <Class_Name>***( );***

Example:

```
import java.lang.*;
class Rectangle
{
double length;
double breadth;
```

```
double area()
{
return length*breadth;
}
public static void main(String args[])
{
Rectangle rect1=new Rectangle();
Rectangle rect2=new Rectangle();
//Assign values to rect1 instance variables
rect1.length=10;
rect1.breadth=20;
//Assign values to rect2 instance variables
rect2.length=30;
rect2.breadth=40;
System.out.println("Area of rect1:"+rect1.area());
System.out.println("Area of rect2:"+rect2.area());
}
}
```

Output:

Area of rect1: 200

Area of rect1: 1200

1.2.1 Introducing methods

A method is a block of code that performs a specific task. In Java there are 2 types of methods namely –

User defined methods – These are created by user.

Standard library methods – These are built-in methods which are already available to user.

General form of a method:

```
<modifier> <static> <returned_data_type> > method_name>(<parameter-list>) {
        // body of method
}
```

Modifier – It defines access types whether the method is public, private etc.

Static – If we use static keyword then we can access method without creating objects.

Return type – It specifies type of data does the method returns. Methods that have a return type other than void return a value to the calling routine using the following form of the return statement: ***return*** <value>;

Method_name – It is an identifier that is used to identify the method.

Body of the method – It includes the programming statements that are used to perform some tasks. It is enclosed in {}.

Parameter list – These values are passed to a method.

Once a method is declared, to invoke the execution of it a method call should be established. Syntax: *Method_name(<parameter-list>);*

1.2.2 Method overloading

If two or more methods are have same name inside a class and they have different arguments then, it is called method overloading. Method overloading can be achieved by either *changing the number of argument*s or *changing type of argument*s.

Example: Write a java program to demonstrate method overloading.

```
import java.io.*;
import java.util.*;
class Addition
{
        void sum(int a, int b)
        {
                System.out.println("Sum of "+a+", "+b+" is "+(a+b));
        }
        //Method overloading by changing the types of parameters
        void sum(double a, double b)
        {
                System.out.println("Sum of "+a+", "+b+" is "+(a+b));
        }
        //Method overloading by changing the number of parameters
        void sum(int a, int b, int c)
        {
                System.out.println("Sum of "+a+", "+b+", "+c+" is "+(a+b+c));
        }
        //Method overloading by changing both type and no. of parameters
        void sum(int a,int b,double c)
        {
                System.out.println(Sum of "+a+", "+b+", "+c+" is "+(a+b+c));
        }
}
class Overload
{
        public static void main(String args[])
        {
                Addition a=new Addition();
                a.sum(10,20);
                a.sum(10,20,30);
                a.sum(10.60,12.07);
```

```
        a.sum(10,20,5.6);
    }
     }
```

Output: java Overload
Sum of 10, 20 is 30
Sum of 10, 20, 30 is 60
Sum of 10.6, 12.07 is 22.67
Sum of 10, 20, 5.6 is 35.6

Note: If only the return types are changed then it is not method overloading.

1.2.3 Constructors

A constructor initializes an object immediately upon creation. It has the same name as the class in which it resides and is syntactically similar to a method. Once defined, the constructor is automatically called immediately after the object is created, before the ***new*** operator completes.

Constructors look a little strange because they have no return type, not even void. This is because the implicit return type of a class constructor is the class type itself. It is the constructor's job to initialize the internal state of an object so that the code creating an instance will have a fully initialized, usable object immediately.

There are two rules defined for the constructor:

- ✓ Constructor name must be the same as its class name
- ✓ A Constructor must have no explicit return type
- ✓ A Java constructor cannot be abstract, static, final, and synchronized.

Default constructors:

A constructor is called "Default Constructor" when it doesn't have any parameter. When there is no explicitly defined constructor for a class, then Java creates a default constructor for the class. The default constructor automatically initializes all instance variables to zero. Once the user defines a constructor to the class, then the default constructor is no longer be used.

Example: Java program to demonstrate default constructor.

```
import java.io.*;
import java.util.*;
class Addition
{
  int x,y;
  Addition()  //default constructor
{
x=10;
y=20;
}
  void display()
{
```

```
        System.out.println("Initialized values are x = "+x+", y= "+y);
  }
}
class con
{
  public static void main(String args[])
{
        Addition b=new Addition();
        b.display();
  }
}
```

OUTPUT:

Initialized values are x = 10, y= 20

Parameterized Constructors:

Each object is initialized as specified in the parameters to its constructor. The parameterized constructor is used to provide different values to distinct objects. However, you can provide the same values also.

Example: Java program to demonstrate parameterized constructor.

```
import java.io.*;
import java.util.*;
class Addition
{
  int x,y;
  Addition(int a,int b) //parameterized constructor
{
x=a;
y=b;
}
  void display()
{
        System.out.println(“Initialized values are x = "+x+", y= "+y);
  }
}
class con
{
  public static void main(String args[])
{
        Addition b=new Addition(10,20);
        b.display();
  }
}
```

OUTPUT:

Initialized values are x = 10, y= 20

1.2.4 Constructor overloading

In java constructors also can be overloaded like methods. Constructor overloading in Java is a technique of having more than one constructor with different parameter lists. They are arranged in a way that each constructor performs a different task.

Example: Java program to demonstrate constructor overloading.

```
import java.io.*;
import java.util.*;
class Addition
{
  int x,y,z;
  Addition(int a)
{
x=a;
}
  Addition(int a,int b)
{
x=a;
y=b;
}
  Addition(int a,int b,int c)
{
x=a;
y=b;
z=c;
}
  void sum()
{
        System.out.println("sum is "+x+y+z);
  }
}
class con
{
  public static void main(String[] args)
{
        Addition a=new Addition(10);
        Addition b=new Addition(10,20);
        Addition c=new Addition(10,20,30);
        a.sum();
        b.sum();
        c.sum();
  }
}
OUTPUT:
      Sum is 10
      Sum is 30
```

Sum is 60

1.2.5 Usage of static with data and method

In java *static* keyword is used to manage the memory. *Static* keyword in java indicates that a particular member is not an instance, but rather part of a type. The *static* member will be shared among all instances of the class, so we will only create one instance of it. The static can be:

- ✓ Variable (also known as a class variable)
- ✓ Method (also known as a class method)
- ✓ Block
- ✓ Nested class

Static variable:

If we declare a variable static, all objects of the class share the same static variable. It is because static variables are also associated with the class. And, we don't need to create objects of the class to access the static variables.

Static Methods:

Static methods are also called class methods. It is because a static method belongs to the class rather than the object of a class. And we can invoke static methods directly using the class name.

Static Blocks:

In Java, static blocks are used to initialize the static variables. The static block is executed only once when the class is loaded in memory. The class is loaded if either the object of the class is requested in code or the static members are requested in code. A class can have multiple static blocks and each static block is executed in the same sequence in which they have been written in a program.

Example: Java program that demonstrates "static" keyword.

```
import java.io.*;
import java.util.*;
class Sample
{
        Scanner scan=new Scanner(System.in);
        int a,b;
        static int c;                   //Static variable
        static {                        //Static block
                int x=40;
                int y=50;
                int z= x+y;
                c=x-y;
                System.out.println("x="+x+"y="+y+"z="+z+"c="+c);
        }
        void read()
        {
                System.out.print("Enter a : ");
```

```
                a=scan.nextInt();
                System.out.print("Enter b: ");
                b=scan.nextInt();
        }
        void display()
      {
                System.out.println("a = "+a+"b= "+b+" c = "+c);
        }
        static void s_display()      //Static method
{
                System.out.println("c = "+c);
        }
}
class Sta
{
        public static void main(String args[])
{
                Sample s1=new Sample();
                Sample s2=new Sample();
                Scanner scan=new Scanner(System.in);
                        s1.read();
                        s1.display();
                        s1.c=40;
                        s2.read();
                        s2.display();
                        s1.display();
                        Sample.c=67;
                        s2.display();
                        Sample.s_display();
        }
}
OUTPUT:
                x = 40  y= 50   z = 90  c = -10
        Enter a : 10
        Enter b: 23
                a = 10  b= 23   c = -10
        Enter a : 20
        Enter b: 34
                a = 20  b= 34   c = 40
                a = 10  b= 23   c = 40
                a = 20  b= 34   c = 67
                c = 67
```

1.2.6 Access control

Access control is a mechanism, an attribute of encapsulation which restricts the access of certain members of a class or entire class to specific parts of a program.

In Java access modifier are available which are used to restrict access of members or classes. There are four access modifiers in Java that are listed as follows.

- public
- protected
- default
- private

There are two levels of the access controlling:

- At top level – public and default.
  - A class may be declared with the modifier public, in which case that class is visible to all classes everywhere.
  - If a class has no modifier (the default, also known as package-private), it is visible only within its own package
- At members' level – public, private, default and protected.
  - Access levels are as follows.

Modifier	class	package	Subclass within the package	Subclass in other packages	Any class in other packages
Public	YES	YES	YES	YES	YES
Protected	YES	YES	YES	YES	NO
Default	YES	YES	YES	NO	NO
private	YES	NO	NO	NO	NO

Example: Java program to demonstrate access controls.

```
import java.io.*;                                          //Line 1
import java.util.*;                                        //Line 2
class Sample{                                              //Line 3
        public int a=10;                                   //Line 4
        private int b;                                     //Line 5
        int c=30;                                          //Line 6
        protected int d=40;                                //Line 7
        void display(){                                    //Line 8
           System.out.println("a="+a+"b="+b+"c="+c+"d="+d);  //Line 9
        }                                                  //Line 10
}                                                          //Line 11
class Access{                                              //Line 12
     public static void main(String[] args){              //Line 13
     Sample obj= new Sample();                             //Line 14
    System.out.println("a="+obj.a+"b="+obj.b+"c="+obj.c+"d="+obj.d)//Line 15
        }                                                  //Line 16
}                                                          //Line 17
```

OUTPUT:

```
Access.java:15: error: b has private access in Sample
System.out.println("a="+obj.a+"b = "+obj.b+"c = "+obj.c+"d = "+obj.d);
                                                  ^
1 error
```

1.2.7 This keyword

In Java, this is a reference variable that refers to the current object. Usage of Java this keyword:

- ✓ *this* can be used to refer current class instance variable.
- ✓ *this* can be used to invoke current class method (implicitly)
- ✓ *this()* can be used to invoke current class constructor.
- ✓ *this* can be passed as an argument in the method call.
- ✓ *this* can be passed as argument in the constructor call.
- ✓ *this* can be used to return the current class instance from the method.

Example:

```
// displays address of class using this keyword
class A
{
void show()
{
System.out.println(this);
}
public static void main(String args[])
{
A obj=new A();
System.out.println(obj);
obj.show();
}
}
```

Output:

```
45632abcv98 //address of class holding by this keyword
45632abcv98 //address of class holding by object
```

//using this keyword for instance variable when both instance variable and local variable has same variable.

```
class A
{
int a;
A(int a)
{
this.a=a;
}
void show()
{
System.out.println(this);
}
public static void main(String args[])
{
A obj=new A(100);
obj.show();
}
```

```
}
```
Output:
100

//using this keyword to call the default constructor of its own class

```
class A
{
A()
{
System.out.println("learn coding");
}
A(int a)
{
this();
System.out.println(a);

}
public static void main(String args[])
{
A obj=new A(100);
}
}
```

Output:
learn coding
100

//using this keyword to call the parameterized constructor of its own class

```
class A
{
A()
{
this(10);
}
A(int a)
{
System.out.println(a);
}
public static void main(String args[])
{
A obj=new A();
}
}
```

Output:
10

1.2.8 Garbage collection

Since objects are dynamically allocated by using the new operator, such objects has to be destroyed and their memory released for later reallocation.

In Java the deallocation of memory for the dynamically memory allocated objects are called as garbage collection which can be achieved automatically.

During the garbage collection process, the collector scans different parts of the heap, looking for objects that are no longer in use. If an object no longer has any references to it from elsewhere in the application, the collector removes the object, freeing up memory in the heap. This process continues until all unused objects are successfully reclaimed.

Advantage of Garbage Collection:

- ✓ It makes java memory efficient because garbage collector removes the unreferenced objects from heap memory.
- ✓ It is automatically done by the garbage collector (a part of JVM) so we don't need to make extra efforts.

Phases of garbage collection:

- ✓ Mark – GC "paints" the objects which are active, leaving the dead objects unmarked.
- ✓ Sweep – GC looks for unpainted objects and collects them for removal from memory.

When the garbage collector chooses an object for garbage collection:

- ✓ Assigning null reference to the variable.
- ✓ Assigning another reference to the variable.
- ✓ Creating local variables: As soon as the method execution is complete all the variables declared in the method will be eligible for garbage collection.
- ✓ Anonymous objects.
- ✓ GC checks for this cyclic reference and removes the object groups that are referencing each other without being referenced by any active object.

Whenever the JVM finds out the heap space is getting occupied and will soon run out of space, it runs the GC thread. Apart from automatic run, we can also explicitly ask the GC to run at any point of the code using *System.gc()* or *Runtime.getRuntime().gc()*.

Example: Java program to demonstrate explicit garbage collection.

```
import java.io.*;
import java.util.*;
class Sample
{
  public void finalize()
{
  System.out.println("Objects Destroyed");
}
  Scanner scan=new Scanner(System.in);
  int a,b;
  void read()
{
```

```
            System.out.print("Enter a : ");
             a=scan.nextInt();
            System.out.print("Enter b: ");
             b=scan.nextInt();
    }
    void display()
  {
            System.out.println("a = "+a+"b= "+b+"c = "+c);
    }
  }
  class Garbage
  {
    public static void main(String args[]){
            Sample s1=new Sample();
            s1=null;
            new Sample();
            System.gc();
    }
  }
```

finalize() method in Java is a method of the Object class that is used to perform cleanup activity before destroying any object. It is called by Garbage collector before destroying the objects from memory. finalize() method is called by default for every object before its deletion. This method helps Garbage Collector to close all the resources used by the object and helps JVM in-memory optimization.

Types of Garbage Collectors in Java: Four types of Garbage collector.

- ✓ Serial GC – This GC runs with a single thread, in other words, when this GC runs it freezes all other threads and does its job.
- ✓ Parallel GC – This GC works in a multi-threaded manner. It runs multiple threads to collect the garbage, but like the previous one, it also stops other threads while running GC threads. It is the default GC of the JVM, and the number of threads, pause time, etc can be controlled through the command line argument. If not configured properly, this may have an impact on the overall performance of the application.
- ✓ CMS GC – Concurrent Mark Sweep, known as CMS garbage collector, this collector also works in multiple GC threads but this doesn't stop the application threads while itself is running. This collector is generally used in an application with a large set of long-lived data, and the hardware has two or more processors which will benefit in running GC threads along with the application threads.
- ✓ G1 GC – Garbage First, known as G1 Garbage collector is also a multi-threaded GC. This is the successor of CMS GC and is available after Java 7. This is a more optimized and powerful GC and is preferably used in multiprocessor machines that have large heap memory. This works in two phases, first, it marks the liveness of objects in heap, and

when it collects all the information of empty spaces in heap, it performs the cleanup activity.

1.2.9 String class

String is a sequence of characters. Java implements string as object of type String. When you create a String object, you are creating a string that cannot be changed i.e., immutable. That is, once a String object has been created, you cannot change the characters that comprise that string.

Syntax of String class: String obj=new String(String);

Even though string is immutable still operations can be performed on it by creating new String object whenever a modification is happened. Hence, the original string has to unmodified. For those cases in which a modifiable string is desired, Java provides two options: *StringBuffer and StringBuilder*. Both hold strings that can be modified after they are created.

Syntax1 of StringBuffer class: StringBuffer obj=new StringBuffer();

Syntax2 of StringBuffer class: StringBuffer obj=new StringBuffer(int);

Syntax3 of StringBuffer class: StringBuffer obj=new StringBuffer(String);

The *String, StringBuffer,* and *StringBuilder* classes are defined in *java.lang.* Thus, they are available to all programs automatically.

The class String includes methods for examining individual characters of the sequence, for comparing strings, for searching strings, for extracting substrings, and for creating a copy of a string with all characters translated to uppercase or to lowercase.

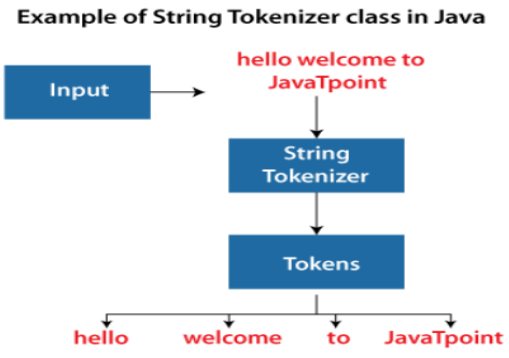
(String & StringBuffer Methods: Refer Notebook)

1.2.10 StringTokenizer

The java.util.StringTokenizer class allows you to break a String into tokens. It is simple way to break a String. It is a legacy class of Java.

It doesn't provide the facility to differentiate numbers, quoted strings, identifiers etc. like StreamTokenizer class. We will discuss about the StreamTokenizer class in I/O chapter.

In the StringTokenizer class, the delimiters can be provided at the time of creation or one by one to the tokens.



Constructors of the StringTokenizer Class

There are two constructors defined in the StringTokenizer class.

Syntax1: StringTokenizer obj=new StringToenizer(String);

Syntax2: StringTokenizer obj=new StringToenizer(String, delimeter);

Methods of the StringTokenizer Class

Methods	Description
boolean hasMoreTokens()	It checks if there is more tokens available.
String nextToken()	It returns the next token from the StringTokenizer object.
String nextToken(String delim)	It returns the next token based on the delimiter.
boolean hasMoreElements()	It is the same as hasMoreTokens() method.
Object nextElement()	It is the same as nextToken() but its return type is Object.
int countTokens()	It returns the total number of tokens.

Example:

```
import java.util.StringTokenizer;
public class Simple
{
 public static void main(String args[])
{
StringTokenizer st = new StringTokenizer("my name is khan"," ");
System.out.println("No. of Token is:"+st.countToken());
while (st.hasMoreTokens())
{
System.out.println(st.nextToken());
}
}
}
```

Output:

No. of Token is: 4

my

name

is

khan