NOTES ON

OCA/OCP Java SE 6 & 7

(Exams 1Z0-851 , 1Z0-803 , 1Z0-804)

Book 1

BY

Sandeep J. Gupta

8. 8th Sept (thu)
1:30 - 5:30 pm
(OCJP)

(or)

Sept (wed)
2-6 pm
(OCJP)

1) Interfaces dont have constructors
2) cannot override a method marked final. It is invalid. compilation fails
3) method is private / static → No overriding But valid
4) constructor or static method → cannot be declared abstract
5) method → final or abstract
6) method → private or abstract
7) interface members → ✗ private ✗ protected They are implicitly public
8) variables of interface must be left uninitialized
9) That subclass method which overrides the abstract method of interface must be kept public

Table of Contents

- Interfaces
- Is a, has a relship
- Upcasting, downcasting
- Polymorphic reference concept

abstract → 1) no body
method 2) compulsory override in subclass

static var → allocated mem only once

final var → cannot be modified.

Sandeep J. Gupta (9821882868)

Q.1 Describe in brief about history of Java.

1990	Sun Microsystems decided to develop a special software that could be used to operate consumer electronic devices like TV, VCR, toaster etc. This task was assigned to a team of scientists headed by James Gosling and Patrick Naughton. This team was known as "Green Project Team"
1991	After exploring the possibility of using C++ for their software, the team decided to develop an altogether new language which they named "Oak"
1992	The Green Project Team demonstrated the application of their new language Oak to control home appliances using a hand held device. Since Oak was to be used on different devices with different architecture, the inventors of Oak always wanted their language to be ***platform independent.***
1993	While Oak was being further developed, there came into existence the World Wide Web or the Internet. Because of its platform independence, the inventors of Oak realized that their language could be used on the Internet on a large scale instead of being confined to consumer electronic appliances. Accordingly, they came up with an idea of developing web applets (tiny programs) using Oak. These applets would run on all types of computers connected to Internet and provided a variety of animation and interaction which greatly enhanced user experience.
1994	The team developed a web browser called ***Hot Java*** which could run applets on the Internet. Because of its efficiency, Hot Java became instantly popular among Internet users.
1995	Because of some legal problems, Oak was renamed Java. Java is just a name, not an acronym. Popular companies like Netscape and Microsoft announced their support for java in their browsers.
1996	Finally Sun Microsystems released Java Development Kit 1.0. Since then it is popularly known as the ***Language of the Internet.***

Q.2 Write a note on features of Java OR Write a note on Java buzzwords.

Given below is a list of Java buzzwords.

Object Oriented

1 Java is a true object oriented language. Everything in Java is an object. All program code and data reside within classes.
2 Java offers a set of classes, arranged in packages which we can use in our programs through inheritance.
3 The object model in Java is simple and easy to extend.

Platform Independent or Portable

1 The most important feature of Java is that it is highly portable. Java programs can be moved anywhere from one computer system to another.
2 Changes in operating system or processor will not force any changes in Java programs.
3 This is the reason why Java has become so popular for Internet programming since different kinds of systems throughout the world are connected to the Internet.

Safe and Secure

1 Security becomes an important issue when one is connected to the Internet. Every time information is downloaded from the Internet, we risk our computer systems to viral attacks. These viruses may gather confidential information like passwords, bank account nos. etc or may even damage the data present on the hard disk.

Java enabled Machine means it has built in JVM

2 Java addresses this issue by making sure that no virus is accompanied with an applet. Not only that, the Java execution environment sees to it that the applet it is going to run does not access those parts of a computer which may result into a security breach.

3 Moreover, the absence of pointers in Java makes sure that malicious programs don't access memory locations without proper authorization.

The Bytecode

1 If we want to work on a language like C, we need C compiler which will convert source code(C statements which the programmer writes) into executable code(machine code for that particular machine). Hence, it becomes necessary to install C compiler on the computer if one intends to execute programs in C. This is the case with almost all languages.

2 Java works somewhat differently. Execution of a Java program takes place in two stages. Java compiler does not directly produce machine code for a specific machine. Rather, the output of Java compiler is an intermediate code called ***bytecode.***

3 This bytecode is then interpreted(translated) by Java interpreter called ***Java Virtual machine(JVM).*** The JVM takes the bytecode as input and converts it to machine code for a particular platform(computer or machine).

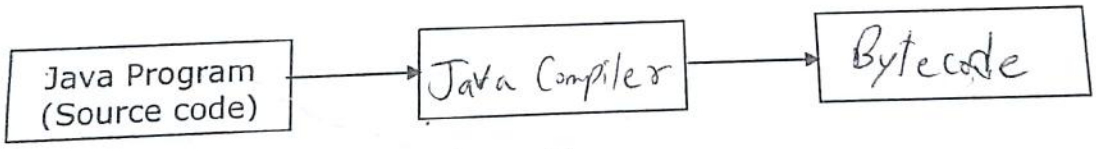
4 Hence, there will be different JVM's for different machines. Although all will take the same bytecode as input but will produce different outputs(machine code) for different machines. Given the bytecode, one now needs to install only the JVM for a particular computer which is much easier than installation of the entire Java compiler.

5 This kind of setup is especially useful for the Internet where the JVM is a part of web browser and the server sends the bytecode instead of sending the Java source code. Because of this mechanism, it is now possible to execute Java programs(for example ***applets***) on a computer which does not have a Java compiler. All we need is a *Java compatible web browser* (a web browser with JVM in it).

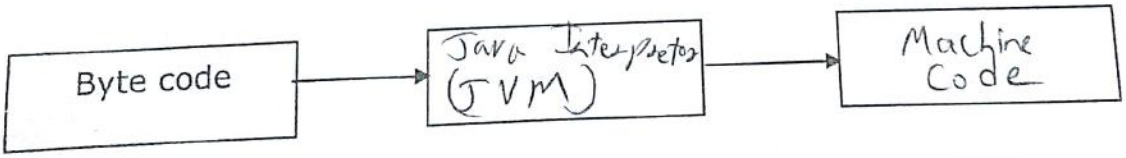
6 The only disadvantage of this mechanism is that programs run a little slower if they are interpreted instead of being directly compiled to executable code. However, the designers of Java have been successful to keep this difference at its minimal.

7 The following diagrams depict the compilation and interpretation of a Java program.

Stage I



Stage II



Robust

1. Since Java is widely used for the Internet it must execute reliably in a variety of systems. Thus the ability to create robust programs was given high priority in the design of Java.
2. Normally a program fails because of the following two main reasons:
   a) Memory management mistakes
   b) Mishandled exceptional conditions(i.e. Run-time Errors)
3. Talking about memory management, in C++ the programmer must manually allocate and free(de-allocate) all the dynamic memory. This could lead to problems

since the programmer may forget to free the unused memory or may even free that memory which is currently being used by some other part of program.

4. Java overcomes these problems by managing memory allocation and de-allocation itself. Infact, memory de-allocation is completely automatic since Java provides garbage collection mechanism for unused objects.
5. When a program executes, we often face run-time errors because of problems like "division by zero", "file not found" etc. Java provides well-defined exception handling mechanism which allows the programmer to mange almost all the possible run-time errors.
6. Moreover, java forces the user to find mistakes in the program during early stages of program development. For example, in C++ the compiler does not report an error if a variable is not initialized. When the program executes that uninitialized variable is assigned a garbage value. However, in Java a variable cannot be kept uninitialized. In other words, one cannot execute a Java program if a particular variable is kept uninitialized.

Distributed

1. Java is designed for the distributed environment of the Internet where it may be required to share both data and programs.
2. Java applications can open and even work on remote objects on Internet as easily as they do in a local system.
3. This enables multiple programmers at multiple remote locations to work ***together*** on a single project.

Multithreaded

1. Multithreading means handling multiple tasks at the same time. Java supports multithreaded programming which means we need not wait for one task(program) to get over before starting the new one.
2. For example, we can listen to music and at the same time surf the Internet.

High performance

1. Programs run a little slower if they are interpreted instead of being directly compiled to executable code. Although Java is interpreted, its designers took great care about its performance issues. According to Sun Microsystems, Java speed is almost comparable to that of C/C++.
2. Moreover, the feature of multithreading also increases the overall speed of execution.

Dynamically Linked

1. Java programs carry with them substantial amount of run-time information which is used to verify and resolve conflicts at run-time.
2. This makes it possible to dynamically link code in a safe and convenient manner.
3. This type of mechanism is important for applets where several decisions may be taken during run-time after interacting with the user.

Simple

1. Java was designed to be an easy language for a professional programmer. If one has already worked on C++, then mastering Java is not a difficult task since Java is also object oriented.
2. To make things simpler, Java has many features which are syntactically same as C++. Not only that, some of the confusing features of C++ are either left out or dealt with in a better way.

Q.3 How is Java strongly associated with the Internet? OR
Why Java is known as the "Language of the Internet"?

Java is known as the "language of the internet" because of its two main properties:

A) Platform Independence:

1. Java was originally designed for the development of software for consumer electronic devices like TV, VCR, toaster etc. Since Java was to be used on different devices with different architecture, the inventors of Java always wanted their language to be *platform independent.*
2. While Java was being further developed, there came into existence the World Wide Web or the Internet. With the emergence of Internet, the programmers realized the importance of platform independent programs.
3. This is because the Internet connects different types of computers across the world having different operating systems. Although the platforms are different, all the Internet users will naturally expect a particular program to behave in the same way at all the times.
4. It now became obvious to the Java design team that the problems of portability which they encountered while writing programs for consumer electronic devices are also found in programs meant for Internet. Because of its platform independence, the inventors of Java realized that their language could be used on the Internet on a large scale instead of being confined to consumer electronic appliances.
5. This realization caused the focus of Java to switch from consumer electronics to Internet programming.
6. In 1994, the Java design team developed a web browser called ***Hot Java*** which could run applets on the Internet. Because of its efficiency, ***Hot Java*** became instantly popular among Internet users and since then it is popularly known as the ***Language of the Internet.***

B) ByteCode: *Refer the earlier answer.*

Q.4 Explain creation and implementation of a Java program.

```
/* This is the first Java program. The output of this program will be
   welcome to Java */
class Example
{
   public static void main(String[] args) //Execution Starts here
   {
      System.out.println("Welcome to Java");
   }
}
```

→ Everything enclosed in a class even main()
→ Execution starts at main
→ main takes one argument -args (array of strings) → [String args[]]
→ System → class, out → Object, println → method

1) After installing Java, go to bin, copy its Path and change the command prompt Path to bin's Path using 'cd' command. Then do the following :-

2) Editing the Program
→ Use a text editor - Notepad
→ Type and save the program as .java file
→ Filename same as classname (having main)
→ 'E' in upper case not only, but a Convention

3) javac Example.java
→ Invokes the Compiler
→ Creates a file Example.class
→ Example class has bytecode

4) Java Example
→ Invokes the Interpreter → Converts Bytecode to machine code → gives output

5) Let's say we have stored Example.class in "C:\d" Now using command prompt, first go to "C:\d", then give the following Command :
"set Path = C:\Program Files\Java\jdk1.7.0_50\bin"
Now we can compile and Execute in a normal way

Q.5 What does Java Environment, Java Development Kit and Application Programming Interface (API) mean?

1. The Java environment includes a large number of development tools and hundreds of classes and methods.
2. The development tools are part of the system known as Java Development Kit (JDK) and the classes and methods are part of the Java Standard Library(JSL), also known as the Application Programming Interface(API).

A) Java Development Kit (JDK)

The Java Development Kit comes with a collection of tools that are used for developing and running Java Programs. They include

Tools	Meaning
appletviewer	Enables us to run Java Applets
java	Java Interpreter, which runs applets and applications by reading and interpreting bytecode files.
javac	The Java Compiler, which translates Java source code to bytecode files that the interpreter can understand.
javadoc	creates HTML format documentation from Java source code files
javah	Produces header files for use with native methods.
javap	Java Disassembler, which enables us to convert bytecode files into program description.
jdb	Java debugger, which helps us to find errors in our programs.

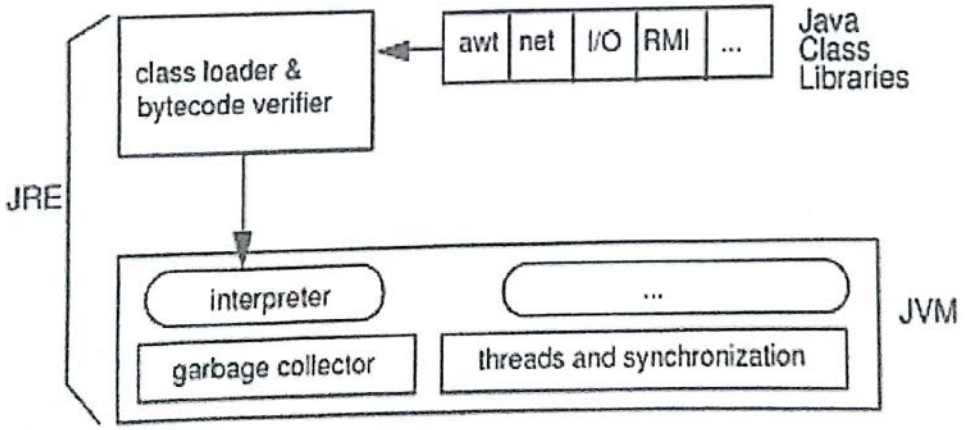
B) Application Programming Interface

The Java Standard Library(JSL), also known as the Application Programming Interface(API)contains hundreds of classes and methods grouped into several packages. Most commonly used packages are:

Language Support Package	This package contains classes and methods required for implementing basic features of Java. Eg. java.lang
Utilities Package	This package contains classes and methods required to implement utility functions such as date and time. Eg. java.util
Input/Output Package	This package contains classes and methods required for performing Input/Output operations. Eg. java.io
Networking Package	This package contains classes and methods required for communicating with other computers via the internet.
Abstarct Window Toolkit (AWT) Package	This package contains classes and methods required for implementing platform independent graphical user interface. Eg. java.awt
Applet Package	This package contains classes and methods required for creating java applets. Eg. java.applet

Q.6 What does Java Run-time Environment mean ?

1. The Java Runtime Environment (JRE) is a combination of
   a) Java Virtual Machine, and
   b) Class Libraries and other components
2. The JVM in JRE runs the program with the help of class libraries, and other supporting components provided in JRE.
3. Java Runtime Environment is a must install on machine in order to execute pre compiled Java Programs. However, the JRE does not contain tools and utilities such as compilers or debuggers required for developing applets and applications. These tools and utilities are present in JDK which is a superset of the JRE.
4. The following diagram shows the JRE and its components:



Q.7 Describe the history of Java Version Numbers.

1. The Java Platform name and version numbering has changed a few times over the years.
2. Java was first released in January 1996 and was named Java Development Kit, abbreviated JDK.
3. Version 1.2 brought many significant changes in Java and was therefore rebranded as Java 2, the full name being Java 2 Standard Edition abbreviated as J2SE. Infact, J2SE is used for all versions from 1.2 to 1.5. That's why these versions are collectively known as Java 2.
4. Version 1.5 was released in 2004 as J2SE 5.0, dropping the "1." from the official name.
5. Finally, with release of version 1.6 in 2006, Sun replaced the name "J2SE" with **Java SE** and dropped the ".0" from the version number.
6. The following table shows version number, release date and new features added:

Version	Release Date	New Features Added
JDK 1.0	23 January, 1996	First Release of Java
JDK 1.1	19 February, 1997	Inner Classes, JavaBeans, JDBC, RMI,
J2SE 1.2	8 December, 1998	strictfp keyword, Swing, Collections Framework, JIT Compiler
J2SE 1.3	8 May, 2000	-
J2SE 1.4	6 February, 2002	assert keyword (assertion)
J2SE 5.0	30 September, 2004	Generics, Autoboxing/Unboxing, Enumerations, Varargs, static import, for each loop
Java SE 6	11 December, 2006	Scripting Language Support, JDBC 4.0 Support
Java SE 7	7 July, 2011	Strings in switch, Allowing underscores in numeric literals, New file I/O library adding support for multiple file systems
Java SE 8	18 March 2014	Language-level support for lambda expressions, a JavaScript runtime which allows developers to embed JavaScript code within applications, New date / time APIs

Q.8 What is meant by Java SE, Java EE and Java ME?

Java SE, Java EE and Java ME are three different Java platforms. Each platform is used to create applications for different types of systems

1. Java SE (Java Platform, Standard Edition)

It is used for developing applications and applets which work on desktops or servers.

2. Java EE (Java Platform, Enterprise Edition)

Java EE extends the Java SE. It is used to develop distributed and multi-tier applications. It assumes that your application spans across some network, talks to different database systems and supports several types of clients like desktops or mobiles.

3. Java ME (Java Platform, Micro Edition)

It is used to develop applications running on mobile and embedded devices like mobile phones, set-top boxes, Blu-ray Disc players, digital media devices, M2M modules, printers and more.

Java ME technology was created to deal with the constraints associated with building applications for small devices with limited memory, display and power capacity.

Q.9 Write a note on Command line arguments / parameters.

1 There may be occasions when we may like our program to behave in a particular way depending on the data provided at the time of execution. This is achieved in Java programs by using mechanism of command line arguments.

2 Command line arguments are arguments which are supplied to the program at the time of invoking it for execution.

3 If we have a Java program stored in a file Example.java then to execute it we give the command

java Example

4 This line is called command line since ***java Example*** acts as a command to execute the application.

5 Instead of simply writing ***java Example,*** We may also write something like

java Example Jack Jill Tim

where "Jack", "Jill" and "Tim" are command line arguments.

6 When we write any Java application the main method starts with the line

public static void main(String[] args)

where args is an array of strings(actually String objects)

7 Now, for the command ***java Example Jack Jill Tim***

```
args[0]="Jack"
args[1]="Jill"
args[2]="Tim"
```

8 Now these arguments can be used in the program in any way we want.

WAJP to add one integer and one floating point no using command line arguments

The Command from command line will be something like

Java ComAdd 2 4.5

```
class ComAdd
{public static void main(String [] args)
 { String s1 = args[0];
   String s2 = args[1];
 } int n = Integer.parseInt(s1);
   double d = Double.parseDouble(s2);
   System.out.println("Sum is "+(n+d));
   // Parentheses across n+d Compulsory
}
```

O/P :-

Sum is 6.5

Sandeep J. Gupta (9821882868)

Q.10 Describe 2 types of programs which can be developed in Java

Java can be used to develop the following 2 types of programs:

Applications(or Stand-Alone Programs)

1 Applications are stand-alone programs which are written in Java to carry out a certain task on a stand-alone computer.

2 In other words, an application in Java is almost same as a program written in C or C++. In fact, Java can be used to develop any program which can be developed using C/C++.

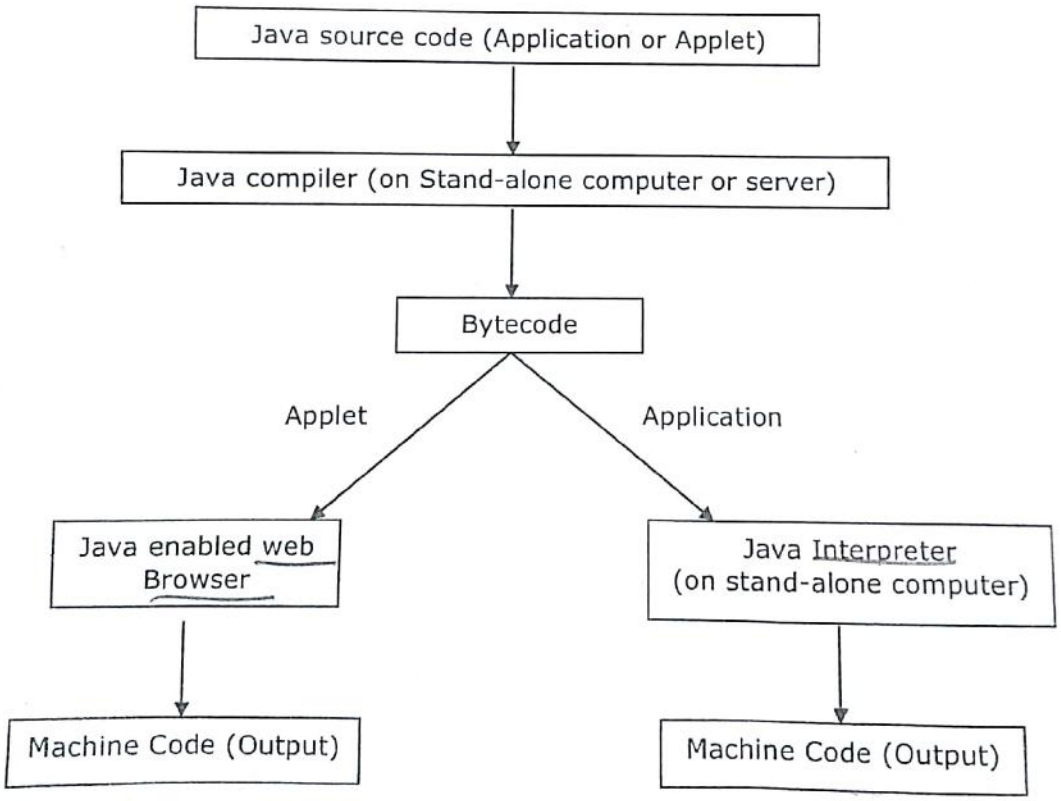
Applets

1 Java is not much different from any other programming language when used to create applications. It is because of Java's ability to create applets that makes it so very important.

2 An applet is a small Java program which is designed to be transmitted over the Internet. An applet is located on a remote computer(server) and is downloaded on a local computer(client) through the Internet. On the client's computer it is executed using a Java enabled web browser.

3 An applet is an ***intelligent program*** which can interact with the user and dynamically change to carry out tasks required by the user.

The diagram given below depicts how applications and applets are executed.



A) Fundamentals of Java

Q.1 Describe the Java character set.

1. Java uses the ***Unicode*** character set. The Unicode is an international character set with 16-bit character coding system.
2. It consists of characters of almost all languages all around the world. These characters could be alphabets, digits or any other special character.
3. Java uses Unicode instead of the normal ASCII character set since Java is used for Internet programming. An applet written using Java may be transmitted anywhere in the world where it may become necessary to use the native language of that place.
4. The ASCII character set which supports English letters, digits and some other symbols is a subset of the Unicode character set. ASCII character set numerically ranges from **0-127 (128 characters)** whereas the Unicode character set contains more than **65000** characters.

Q.2 What are tokens in Java ?

1 The smallest individual unit in a Java program is known as a token.

2 A Java program is basically a collection of classes. Classes contain declaration statements and methods. Methods contain executable statements. These declaration and executable statements are made up of the following tokens:

- Keywords
- Identifiers
- Literals (constants)
- Operators
- Separators

3 In other words, a Java program is a collection of tokens, comments and white spaces.

Q.3 What is an identifier? What rules should be observed while framing an identifier?

1. Identifiers are names given to various program elements, such as classes, methods, variables, objects, labels, packages, interfaces etc in Java programs.
2. Rules of formation of an identifier are as follows:
   a) An identifier can consist of letters, digits, dollar sign and an underscore.
   b) An identifier cannot start with a digit.
   c) An identifier name cannot be a keyword.
   d) Uppercase and lowercase letters are distinct. Hence "area" is different from an identifier called "Area".
   e) An identifier can be of any length.
3. Examples :

Valid identifiers: ($35) (_result) (add2) (ende) (end_result) (float) (Int)

Invalid identifiers: (end result) (while) (sum's) (float) (int)

Q.4 What are keywords in Java ? What restrictions apply to their use?

1. Keywords are reserved words that have standard, predefined meanings in Java. The keywords should always be written in lower-case.
2. The restriction which applies to their use is that they can be used only for their intended purposes; they cannot be used as programmer defined identifiers.
3. Keywords allowed in Java are as follows:

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

assert added in 1.4 and **enum** added in 1.5.

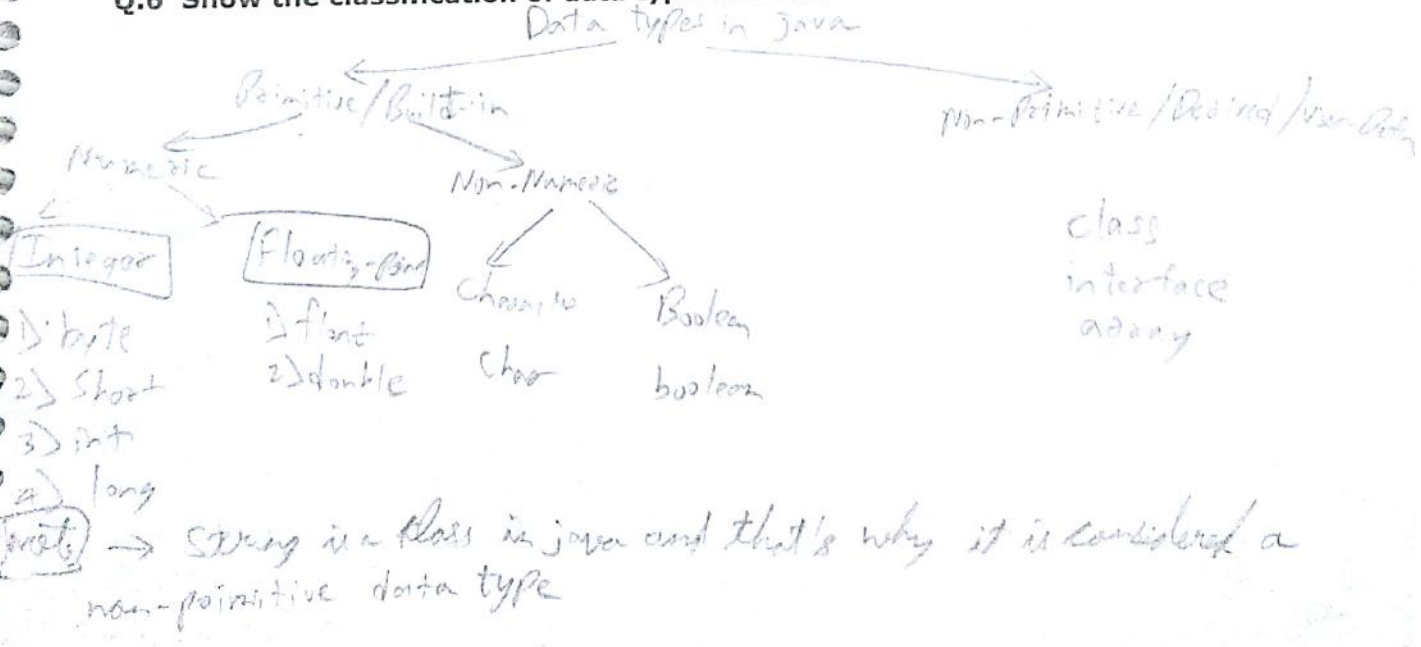
4. Certain words like **true, false** and **null** are also reserved by Java and hence cannot be used by programmer.

Q.5 Give a list of separators used in Java.

Given below is a list of separators along with their usage:

Parantheses ()	Normally used with functions and expressions
Braces {}	Normally used to define a block, sometimes used for array initialization
Brackets []	Normally used with arrays
Semicolon ;	Used to separate statements
Comma ,	Normally used to separate identifiers in variable declaration, sometimes used within *for* statement
Period .	A key feature of object oriented programming.

Q.6 Show the classification of data types in Java.



→ String is a class in java and that's why it is considered a non-primitive data type

Describe all the primitive/built-in data types in Java.

Description of built-in data types in Java is as follows:

Category	Data type	Memory Require -ment	Range	Default
Integer Type	byte	1 byte	-128 to 127	0
Integer Type	short	2 bytes	-32768 to 32767	0
Integer Type	int	4 bytes	-2147483648 to 2147483647	0
Integer Type	long	8 bytes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	0L
Floating-point Type	float (IEEE 754 floating point)	4 bytes	+/- 1.4e-45 to +/- 3.4e+38	0.0F
Floating-point Type	double (IEEE 754 floating point)	8 bytes	+/- 4.9e-324 to +/- 1.79e+308	0.0d
Character Type	char	2 bytes	0 to 65535	\u0000
Boolean Type	boolean	1 bit	true / false	false

Q.7 What are variables? What are constants? What is final variable?

1. A variable is a program element whose value can change during program execution.
2. Although the value of the variable can change during execution of the program, its data type cannot change.
3. Constants (also called **literals**) in Java refer to fixed values that do not change during the execution of a program.
4. A final variable (also called symbolic constant) is that variable which is assigned a value and that value is its final value.
5. A final variable is declared as follows in general:

   final data-type variable-name = value;

   Some valid examples are as shown:

   ```
   final float PI=3.141593;
   final int D=5;
   ```
6. Final variables are normally written in **capitals** to distinguish them from normal variables. However, this is not a rule.
7. After declaring a final variable, it should not be assigned any other value within the program. This is the biggest difference between a variable and a final variable.

Q.8 Explain Type Conversion and Type Casting in Java?

1. In programming, it is fairly common to assign a value of one type to a variable of another type. If the two types are compatible, then before assignment, Java will perform the data type conversion itself. This is known as ***automatic type conversion or simply Type Conversion.***
2. Automatic type conversion will take place if the following two conditions are met:
   a) **The two types are compatible**
   b) **The destination type is larger than the source type.**
3. When these two conditions are met, a ***widening*** *conversion* takes place. For example, data type int is large enough to hold a byte value. For widening conversions, the numeric types i.e. integer and Real are compatible with each other. However these numeric types are not compatible with char or boolean. Also char and boolean are not compatible with each other.
4. If we however want to give an int value to a byte variable automatic type conversion will not take place since a byte is smaller than an int. This type of conversion is

called ***narrowing*** conversion, since the source is explicitly made narrower to fit in destination type.

5. For a narrowing conversion or for type conversion between two incompatible types we must explicitly use a procedure called ***Type Casting*** which involves cast operator. It has the following general form: ***(target-type) value*** where target-type is the desired type to which the specific value should be converted.
6. The following examples show casting of int to byte.

Eg 1 :-

```
int x = 5;
byte a;
a = (byte)x;
a = 5
```

Eg 2 :-

```
int x = 260;
byte a;
a = (byte)x;
a = 4 ∴ (byte 9 = 260 % 256 = 4)
```

[no of elements in range of byte]

[127 + 128 = 255]

7. The following examples show casting of floating-point value to integer types.

Eg 1 :-

```
double x = 3.45;
int a;
a = (int)x;
a = 3
```

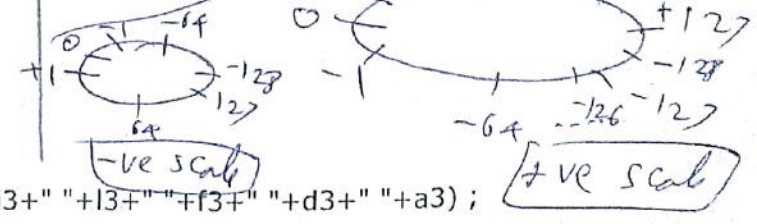
o/p :- 3 3 3 3 3.0 2.0 131

Rule :- If we do any operation on 2 variables of data type int / long / float / double then the result can be stored in a variable of same data type. However if we perform any operation on 2 variables of data type byte / short / char, then the result is of data type 'int'. That's why, statements 1, 2, 3 are invalid

8. Consider the following program:

```
class DTConvert
{
public static void main(String[ ] args)
{
 byte b1=1, b2=2, b3 ;
 short s1=1, s2=2, s3 ;
 int i1=1, i2=2, i3 ;

 long l1=1, l2=2, l3 ;
 float f1=1.0f, f2=2.0f, f3 ;
 double d1=1.0, d2=2.0, d3 ;
 char c1='A', c2='B', c3 ;

1) // b3 = b1 + b2 ;→Error
2) // s3 = s1 + s2 ;→Error
   i3 = i1 + i2 ;
   l3 = l1 + l2 ;
   f3 = f1 + f2 ;
   d3 = d1 + d2 ;
3) // c3 = c1 + c2 →Error

   int a1, a2, a3 ;
   a1 = b1 + b2 ;
   a2 = s1 + s2 ;
   a3 = c1 + c2 ;

   System.out.println(a1+" "+a2+" "+i3+" "+l3+" "+f3+" "+d3+" "+a3) ;
}}
```

```
byte a;
int x = some value;
a = (byte)x;
a = some value % 256 = y;
   is 0 - y - 127
Yes                    No
a = y                  Refer diagram
```



B) Operators

Q.1 Describe the five arithmetic operators in Java ?

There are five arithmetic operators in Java . They are as follows:

Operator	Purpose
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Remainder after division(modulus)

Arithmetic Operators
⇒ + - * / %
→ 17 % 5 = 2
→ 20 % 8 = 4
→ 15 % 5 = 0
→ 15 / 5 = 3
→ 5 % 17 = 5
→ Note :- If first operand is lesser than 2nd operand, Ans is the 1st operand.

→ 17 % 5 = 2
→ 17 % -5 = 2
→ -17 % 5 = -2
→ Note :- The sign of the result is same as the sign of 1st operand irrespective of the sign of the 2nd operand

→ 13.8 % 2.9 = 2.2
→ Note :-
"The Quotient must be an integer". Here the quotient is 4.
Hence
2.9 * 4 = 11.6
and
13.8 - 11.6 = 2.2
which is the Ans

13.871 % 2.356 = 2.355

int a = 10, b = 4
a/b = 2

Q.2 Describe the increment and decrement operator.

1. Increment operator (++) causes its operand to be increased by one.
2. Decrement operator (--) causes its operand to be decreased by one.
3. The operator can be placed either before (prefix) or after (postfix) a variable to change its value.

Rule 1 :- The result of prefix notation and Postfix notation will be same if the expression has only 1 operator and only 1 operand

Rule 2 :- ++ and -- can be used only with single variable. Hence i+2 and --(x+2) are invalid

Rule 3 :- i++ → i = i+1 and not only i+1

Exercise:

```
x=4 ;
z = ++x ;
System.out.println(x+" "+z) ;
```

O/P :- 5 5

Note :- First x is incremented, then that value is given to z.

```
x=4 ;
z = x++ ;
System.out.println(x+" "+z) ;
```

O/P :- 5 4

```
x=4 ;  y=6 ;
z = ++x + y++ ;
System.out.println(x+" "+y+" "+z) ;
```

O/P :- 5 7 11

Sol :- x = 5, y = 7, z = 5 + 6 = 11

```
x=4 ;
z = x++ + x++ ;
System.out.println(x+" "+z) ;
```

O/P :- 6 9

```
x=4 ;
z = x++ + x++ + ++x + ++x + x++ ;
System.out.println(x+" "+z) ;
```

O/P :- 9 32

Sol :- z = 4 + 5 + 7 + 8 + 8, x = 9

```
x=4 ;  y=6 ;
z = y++ - ++x + ++y + x++ - ++x ;
System.out.println(x+" "+y+" "+z) ;
```

O/P :- 7 8 7

```
x=4 ;
SopIn(x+" "+x--+" "+--x+" "+x--) ;
System.out.println(x) ;
```

O/P :- 4 4 2 2

1

Q.3 Describe the six relational operators included in Java. With what type of operands can they be used? What type of expression is obtained?

1. The six relational operators in Java are as follows:

Operator	Name
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
==	equal to
!=	not equal to

2. They can be used with integers, floating-point numbers or individual characters.
3. The resulting expression will have a boolean value of true or false.
4. The last two operators are sometimes called equality operators.

Q.4 Describe the logical operators included in Java ?

1. The logical operators in Java are as follows:

Operator	Name
&&	Short Circuit AND
||	Short Circuit OR
&	AND
|	OR
^	Exclusive-OR (XOR)
!	NOT (Boolean Inversion)

2. The && and & operators return true only if both operands are true.
3. The || and | operators return true if either one operand or both operands are true.
4. The && and || operators are called short-circuit operators. This is because the && operator does not evaluate the right operand if left operand is false. Similarly the || operator does not evaluate the right operand if left operand is true.
5. The ^ operator returns true if exactly one operand is true.
6. The first five logical operators operate on two boolean expressions while ! works on a single boolean expression and returns the opposite of the boolean operand it works on.
7. &, | and ^ can also be called Non Short Circuit operators.

No.	Values of x, y, z	Expression	true/ false	x	y	z
1	x=4 ; y=2 ; z=6 ;	x < y && x == ++z	False	4	2	6
2	x=4 ; y=2 ; z=6 ;	x < y & x == ++z	False	4	2	7
3	x=4 ; y=2 ; z=3 ;	x >= y && x == ++z	True	4	2	4
4	x=4 ; y=2 ; z=6 ;	x > y || x != ++z	True	4	2	6
5	x=2 ; y=2 ; z=1 ;	x <= y | x != ++z	True	2	2	2
6	x=4 ; y=2 ; z=3 ;	x >= y ^ x == z++	True	4	2	4
7	x=4 ; y=2 ; z=3 ;	(x == y) ^ !(y != z)	False	4	2	3

Q.5 Describe the assignment operators?

1. Assignment operators are used to assign value of an expression to an identifier.
2. The most commonly used assignment operator is '=' which is used as
   identifier = expression
3. Java contains the following five additional assignment operators:
   `+=` `-=` `*=` `/=` `%=`
   Collectively they are known as ***compound assignment operators***.
4. The assignment expression
   identifier += expression
   is equivalent to
   identifier = identifier + expression
5. Hence writing **x+=2** is same as writing x = x + 2.
6. Assignment operators have the lowest precedence among all arithmetic operators. Moreover, assignment operations have right-to-left associativity.

Q.6 Describe the conditional operator (ternary operator).

1. Instead of an *if-else* statement a conditional operator (?:) can be used to form a conditional expression.
2. A conditional expression is written in the form
   expression 1 ? expression 2 : expression 3
3. When evaluating a conditional expression, *expression 1* is evaluated first. If *expression 1* is true, then *expression 2* is evaluated and this becomes the value of the conditional expression. However, if *expression 1* is false, then *expression 3* is evaluated and this becomes the value of the conditional expression.

Exercise:

```
class conditional
{
public static void main(String[ ] args)
{
 int a, b, c, max ;
 String s ;

 a=2 ; b=4 ;
 s = (a > b) ? "a greater than b" : "b greater than a" ;
 System.out.println(s) ;

 a=2 ; b=4 ; c=1 ;
 max = a > b ? ( a > c ? a : c ) : ( b > c ? b : c ) ;
 System.out.println(max) ;
}
}
```

o/p :- b greater than a
4

Q.7 Describe the *instanceof* operator of Java.

1. The instanceof operator allows us to determine whether an object belongs to a particular class or not. The instanceof operator in general is used as
   Object instanceof Class
   It returns true if the object on LHS is an instance of the class on RHS.
2. For example, *c1 instanceof Complex* is true if the object c1 belongs to class student. Otherwise it is false.

Q.8 Describe the string concatenation operator.

```
class stringcat
{
public static void main(String[ ] args)
{
 String a , b , c ;
 int x , y ;

 a = "Lal" ; b = "Baadshah" ;
 c = a + b ;
 System.out.println(c) ;
 c = a + " "+ b ;
 System.out.println(c) ;

 a = "area" ;
 x = 5 ; y = 1 ;
 b = a + x + y ;
 System.out.println(b) ;
 System.out.println(a + x + y) ;
 System.out.println(x + y) ;
 System.out.println(x + y + a) ;

 System.out.println(x + " " + y) ;
 System.out.println(" " + x + y) ;
 System.out.println(x + y + " ") ;
}
}
```

Q.2 Describe the precedence and associativity of all the operators.

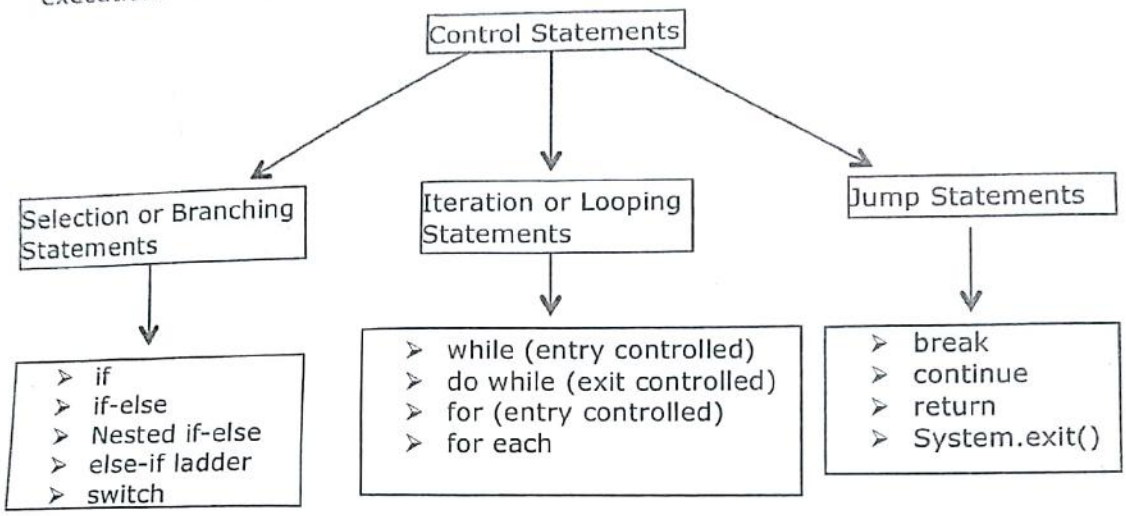
7. Precedence and associativity of all operators is as shown:

Operator category	Operators	Ass.
Dot, Function call, Array element	. () []	L→R
Unary operators	++ --	**L→R**
	- ! ~ (type)	R→L
Arithmetic multiply, divide and remainder	* / %	L→R
Arithmetic add and subtract	+ -	L→R
Left shift, Right Shift, Right shift with zero fill	<< >> >>>	L→R
Relational operators	< <= > >= instanceof	L→R
Equal to, Not Equal to	= = !=	L→R
Bitwise AND or AND	&	L→R
Bitwise XOR or XOR	^	L→R
Bitwise OR or OR	|	L→R
short-circuit AND	&&	L→R
short-circuit OR	||	L→R
Conditional operator	? :	R→L
Assignment operator	= + = - = * = / = % =	R→L

8. The table mentions all the operators in the decreasing order of precedence.

Q.1 What is a control statement ?

1. Statements in a program are normally executed one after another till the last statement.
2. A control statement is a statement which is used to alter this normal flow of execution. Control statements in Java are of following types:



Selection or Branching statements

Q.1 What is a selection statement?

1. A selection statement is a control statement which allows choosing between two or more execution paths in a program.
2. The selection statements in Java are the if statement, if-else statement, nested if-else statement, else-if ladder and the switch statement.
3. These statements allow us to ***select*** a particular execution path based on a boolean expression (also called condition) which may be true or false.
4. A selection statement is sometimes also called conditional statement.

Q.2 Describe the if and if – else statement.

1. The *if* statement can be written as

   if (test expression) statement

2. The *statement* will be executed only if the *test expression* is true. If the *test expression* is false then the *statement* will be ignored.
3. The *statement* can be either simple or compound. In practice, it is often a compound statement.
4. The general form of an *if - else* statement is

   if (test expression) statement 1 else statement 2

If the test expression is true statement1 will execute. If the *test expression* is false *then statement2* will execute.

Q.3 Describe the nested if-else statement.

1. Nesting means "One Within Another". It is possible to nest if-else statements with another. There are several different forms that nested *if – else* statements can take. The most general form of two-layer nesting is

```
if(e1)
        if(e2)
          s1
        else
          s2
else
        if(e3)
          s3
        else
          s4
```

where e1, e2 and e3 represent test expressions, and s1, s2, s3 and s4 represent statements which can be simple or compound.

2. In this situation, one complete if – else statement (if e2 s1 else s2) will be executed if e1 is nonzero (true), and another complete if – else statement (if e3 s3 else s4) will be executed if e1 is zero (false).
3. It is, of course, possible that s1, s2, s3 and s4 are themselves if – else statements. We would then have multilayer nesting.

Q.4 How is the following nested if-else statement interpreted?
if e1 if e2 s1 else s2
which logical expression is associated with the else clause?

1. In this situation, one complete if – else statement (if e2 s1 else s2) will be executed if e1 is nonzero (true).
2. This is because the rule is that the else clause is always associated with the ***closest preceding unmatched*** if.
3. Thus, the else is associated with second if. Hence it is the expression e2 which is associated with the else.

Q.5 Describe the else-if ladder?

1. In else-if ladder there is a chain of *if's* and the statement associated with each *if* is an *else*. This construct is mainly used when multiple decisions are involved.
2. It takes the following general form:

```
if(expression 1)
        statement-1
else  if(expression 2)
            statement-2
                .
                .
                .
      else  if(expression n)
                statement-n
            else
              default-statement
statement-x
```

3. The conditions are evaluated from the top of the ladder downwards. As soon as a true condition is found, the statement associated with it is executed and the control is transferred to *statement-x* (which is not a part of ladder) thereby skipping rest of the ladder.
4. When all the expressions are false, the *default-statement* is executed and the control is transferred to *statement-x*. However, this default statement is optional.

Read two integers from command line and display their maximum using if and if-else.

```
class TwoIf
{
public static void main(String[ ] args)
{
 int a , b ;

 a = Integer.parseInt(args[0]) ;
 b = Integer.parseInt(args[1]) ;
 if(a>b)
      S.O.P(a);
 if(b>a)
      S.O.P(b);

}
}
```

```
class IfElse
{
public static void main(String[ ] args)
{
 int a , b ;

 a = Integer.parseInt(args[0]) ;
 b = Integer.parseInt(args[1]) ;
 if(a>b)
      S.O.P(a);
 else
   S.O.P(b);

}
}
```

Display maximum of three integers a, b,c

java NestedIfelse 2 6 4

```
class NestedIfelse
{ Public static void main (String args[])
  { int a,b,c;
    a=Integer.ParseInt(args[0]);
    b=Integer.ParseInt(args[1]);
    c=Integer.ParseInt(args[2]);
    if(a>b)
        if(a>c)
            S.O.P(a);
        else
            S.O.P(c);
    else
        if(b>c)
            S.O.P(b);
        else
            S.O.P(c);
  }
}
```

Q.6 What is the purpose of the switch statement ? Explain its working in detail?

1. The general form of the switch statement is

```
switch (expression)
{
        case constant1:   block-1
                          break;
        case constant2:   block-2
                          break;
        .....
        .....
        default:   default-block
                   break;
}
statement-x
```

2. The *expression (also called argument of switch)* should be of data type **byte, short, int, char, String (as of Java 7) and enum (as of Java 5)**
3. **constant1, constant 2..... must be a constant (literal) of the same data type as expression. It may also be a final variable provided it is immediately initialized with a literal (constant).**
4. The case labels (constant1, constant 2...) must be **unique** within a given switch statement.
5. The *break* statement after each block signals the end of a particular case and causes an exit from the switch statement, thereby transferring the control to *statement-x* following the *switch.*
6. When the *switch* is executed, the value of the *expression* is successively compared with values constant1, constant 2... . If a case is found whose value matches the value of the expression, then the block of statements in that case is executed. After this, break terminates the switch and control is transferred to *statement-x.*
7. One of the labeled groups of statements within the switch statement may be labeled *default*. This group will be selected if none of the case labels matches the value of the expression. This way we can generate error messages or even do some rectification.
8. The *default* group may appear **anywhere** within the switch statement. It need not necessarily be placed at the end. The default block is optional.
9. Moreover, it is syntactically correct if break is not written after each block. In that case, the compiler will also execute that block which comes next. This mechanism is called **fall-through**. In fact, the default block too works just like any other case for fall-through.

Write a program to read an integer from command line.
If the user enters 1 display "One".
If the user enters 2 display "Two".
If the user enters any other integer display "Try Again".
Do the program using else-if ladder and switch.

```
class OneTwo
{
public static void main(String[ ] args)
{
 int n ;

 n = Integer.parseInt(args[0]) ;
 if(n==1)
      S.O.P("One");
 else if(n==2)
      S.O.P("Two");
 else
    S.O.P("Try Again");
}
}
```

```
class OneTwo
{
public static void main(String[ ] args)
{
 int n ;

 n = Integer.parseInt(args[0]) ;
 switch(n)
{ case 1 : S.O.Pln("One");
          break;
  case 2 : S.O.P("Two");
          break;
  default : S.O.P("Try Again");
          break;
}
}
```

Example 1

O/P: - Yes

```
char ch = 'y';
switch(ch)
{
      case 'y': System.out.println("Yes"); break ;
      case 'n': System.out.println("No"); break ;
}
```

Example 2

O/P: Colour of love

```
String s = "red" ;
switch(s)
{
      case "red": System.out.println("Color of love");        break ;
      case "white": System.out.println("Color of peace");    break ;
      default: System.out.println("Neither red nor white");  break ;
}
```

Iteration or looping statements

Q.1 Explain while loop.

1. The general form of the statement is
   while (expression) statement
2. The statement will be executed repeatedly, as long as the expression is true.
3. This statement can be simple or compound, though it is typically a compound statement.

Q.2 Explain do while loop.

1. When a loop is constructed using the while statement, the test for continuation of the loop is carried out at the beginning of each pass. Sometimes, however, it is desirable to have a loop with the test for continuation at the end of each pass. This can be accomplished by means of the do - while statement.
2. The general form of the do - while statement is
   do statement while (expression)
3. The statement will always be executed at least once, since the test for repetition does not occur until the end of the first pass through the loop.

Q.3 Explain the for statement ?

1. The for statement is the most commonly used looping statement in Java.
2. The general format of the for statement is

   for (expression 1; expression 2; expression 3) statement

   where *expression 1* is used to initialize some variable (called index or loop control variable) that controls the looping action, *expression 2* represents a condition that must be true for the loop to continue execution and *expression 3* is used to alter the value of the index.
3. Normally, *expression 1* is an assignment expression, *expression 2* is a logical expression and *expression 3* is a unary expression or an assignment expression.
4. From a syntactic standpoint all three expressions need not be present in the for statement, though the semicolons must be shown.

Using while, do while and for loops, display digits from 0 to 9.

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

```
i=0
while(i<=9)
{ S.o.P(i+" ");
  i++;
}
```

```
i=0
do
{S.o.P(i+" ");
  i++;
} while(i<=9);
```

```
for(i=0; i<=9; i++)
   S.o.P(i+" ");
```

Note:

```
i=0;
for( ; ;)
   S.o.P(i+" ");
```

Here loop executes infinitely

```
}
}
```

→ break statement terminates "loop" and "switch"

Valid for loop:

```
for( i=1 , j=1  ; i<=n ; i++ , j++ )
{
      .....
}
```

Invalid for loop:

```
for(i=1 , j=1 ; i<=n , j<=n ; i++ , j++)
{
      .....
}
```

Q.4 Explain the Enhanced for loop.

1. The Enhanced for loop (as of Java 5) is also called for each loop.
2. It is used to loop through an array or collection.
3. Its format is

   for(declaration : expression)
4. The declaration consists of a newly declared block variable having same data type as the array.
5. The expression represents an array or a collection we want to loop through. It could even be a method call that returns an array.

The following segment shows how elements of an array can be displayed:

```
int x[ ]={2,4,6,8};
for (int i=0;i<x.length;i++)
    S.o.p(x[i]+" ");
```

The above for loop can also be written as:-

```
for(int a:x) → (for each loop)
    S.o.p(a+" ");
```

Note:- 1) In "for each" loop, the data type of a and x must be same.

2) In first execution, a = x[0]
   In second execution, a = x[1]
   and so on

The following program shows how command line arguments can be displayed:

Java CommandForEach Asya Ned Robb

```
class CommandForEach
{
public static void main (String[ ] args)
{
  for(int j=0 ; j<args.length ;j++)
      S.o.p(args[j]+" ");
```

OR

```
{ for (String a:args)
      S.o.p(a+" ");
```

```
}}
```

Jump statements

Q.1 Explain the break and continue statement.

1. The break statement is used to terminate loops or to exit from a switch. It can be used within a while, a do - while, a for or a switch statement.
2. The break statement is written simply as
   break;
   without any embedded expressions or statements.
3. If a break statement is included in a while, do-while or for loop, then control will immediately be transferred out of the loop when the break statement is encountered.
4. In the event of several nested while, do - while, for or switch statements, a break statement will cause a transfer of control out of the immediate enclosing statement, but not out of the outer surrounding statements.
5. The continue statement is used to bypass the remainder of the current pass through a loop.
6. The continue statement can be included within a while, a do - while or a for statement. It is written simply as
   continue;
   without any embedded statements or expressions.
7. The loop does not terminate when a continue statement is encountered. Rather, the remaining loop statements are skipped and the computation proceeds directly to the next pass through the loop. This is an important distinction between *continue* and *break*.

Q.2 Explain Labeled break and Labeled continue statements.

1. Normally a break statement is used with switch or within a loop to cause their termination. However, a break statement can also be used independently with the help of a label.
2. When used independently, it is called labeled break and it works somewhat similar to goto statement of C.
3. By using a labeled break we can break out of one or more blocks of code. Moreover, these blocks may not be associated with a switch or a loop. Moreover, the associated label makes it possible to specify where the execution should precisely resume.
4. The general form of labeled break is
   break label;
   Here, the *label* is simply an identifier which identifies a specific block of code. Now when the statement executes control is transferred out of this block of code.
5. The statement ***break label;*** must be necessarily enclosed in the block with the same label. However, the enclosing block need not be the immediately enclosing block. This means that a labeled break can be used to break out of even a nested set of blocks which is not possible with a normal break.
6. A block can be labeled by simply putting an identifier followed by a colon before the block starts.
7. Just like labeled break we can have labeled continue. There are however two differences between labeled break and labeled continue.
8. Firstly a labeled break will terminate the labeled block while a labeled continue simply continues with the next execution of the labeled loop.
9. Moreover, a labeled break can be used independently without a switch or loop. However a labeled continue has to be used with a loop.

Q.3 Explain return and System.exit().

1. The return statement transfers control out of the method and takes the execution back to the calling portion.
2. System.exit() statement stops the program execution after which the the VM shuts down.
3. The most important distinction between them is that
   return terminates a **method** while
   System.exit() terminates the **program**.

Examples:

```
class break1
{
public static void main(String
args[ ])
{
 int i ;
 for(i=1 ; i<=10 ; i++)
 {
  if(i%3==0)
       break ;
  System.out.print(i) ;
 }
}
}
```

O/P :- 12

```
class break2
{
public static void main(String args[ ])
{
 int i , j ;
 for(i=1 ; i<=3 ; i++)
 {
  for(j=1 ; j<=389 ; j++)
  {
   if(j==3)
        break ;
    System.out.print(j) ;
  }
 System.out.print("#") ;
 }}
}
```

O/P :- 12 # # # 12 # 12 # 12 #

```
class break3
{
public static void main(String
args[ ])
{
 int i , j ;

 outer:
 for(i=1 ; i<=3 ; i++)
 {
  for(j=1 ; j<=389 ; j++)
  {
    if(j==3)
         break outer ;
    System.out.print(j) ;
  }
  System.out.print("#") ;
 }
}
}
```

O/P : 12

Note :- The label given for a "for loop" is only for that for loop and not for the next statements

```
class break4
{
public static void main(String args[ ])
{
 int i , j ;
 outer:
 for(i=1 ; i<=3 ; i++)
 {
  System.out.print("#") ;
 }
 for(j=1 ; j<=389 ; j++)
 {
  if(j==3)
    break outer ;
  System.out.print(j) ;
 }
}
}
```

O/P :- Error

Note :- "break outer" is not inside the lable outer.

Indentation :- Pushing to right

* If break is not present in switch case statements, fallthrough.


```
class breakblocks {
public static void main(String args[ ]) {
  boolean x=true;
  first:
  {
   second:
   {
    third:
    {
     System.out.println("Block 3, Statement 1");
     if(x)
       break third;
     System.out.println("Block 3, Statement 2");
    }
    System.out.println("Block 2, Statement 1");
    if(x)
      break second;
    System.out.println("Block 2, Statement 2");
  }
  System.out.println("Block 1, Statement 1");
  if(x)
    break first;
  System.out.println("Block 1, Statement 2");
 }
}}
```

Restriction - A labelled break must be written inside a block of the same label, otherwise the compiler reports an error. Hence, if we write "break second" instead of "break third"; it is valid. But long vice versa is invalid.

O/P :- Block 3, Statement 1
Block 2, Statement 1
Block 1, Statement 1

Def^n of continue :-
Continue skips the remaining statements of loop

```
class continue1
{
public static void main(String
args[ ])
{
 int i ;
 for(i=1 ; i<=10 ; i++)
 {
     if(i%3==0)
          continue ;
     System.out.print(i) ;
 }
}}
```

O/P :- 1 2 4 5 7 8 10

```
class continue2 {
public static void main(String args[ ])
{ int i , j ;
 for(i=1 ; i<=2 ; i++)
 {
  for(j=1 ; j<=3 ; j++)
  {
     if(j==2)
          continue;
     System.out.print(j);
  }
  System.out.print("#");
 }
}}
```

Note :- similar to break 2

O/P :- 1 3 # 1 3 #

```
class continue3
{
public static void main(String
args[ ])
{
 int i , j ;
```

```
 outer:
 for(i=1;i<=2;i++)
 {
  for(j=1;j<=3;j++)
  {
     if(j==2)
          continue outer;
     System.out.print(j);
  }
  System.out.print("#");
 }
}}
```

Note :- similar to break 3

O/P :- 1 1

Naming Conventions

Q.1 Describe the naming conventions followed in java.

1. **Classes and Interfaces**
   The first letter should be in uppercase and the remaining in lowercase. If several words are linked together to form the name, the first letter of the inner words should be in uppercase. This format is sometimes called CamelCase. For classes, the names are normally nouns.
   Example: Dog, Account, EmployeeData
   For interfaces, the names are normally adjectives.
   Example: Runnable, Serializeable

2. **Methods & Instance Variables**
   The first letter should be in lowercase and then normal CamelCase should be used. Moreover, the names are normally verb-noun pairs.
   Example: calculate, findAverage, getBalance

3. **Local Variables**
   Local variables are given in lowercase separated by underscores if required.
   Example: length, day_of_week

4. **Constants or Final Variables**
   Final variables are given in uppercase separated by underscores if required.
   Example: PI, INTEREST_RATE

Data Output

1) `S.O.Pln("Welcome to java");`

O/P:- Welcome to java

O/P: Welcome to java

2)
```
int n=8;
```
`S.O.Pln("The value of n is "+n);`

O/P: The value of n is 8

3)
```
double salary=500.5;
int code=123;
```
`S.O.Pln("Salary of employee "+code+" is "+salary);`

O/P: Salary of employee 123 is 500.5

4)
```
int n=3 , f =6 ;
```
`S.O.Pln(f+" is the factorial of "+n);`

O/P: 6 is the factorial of 3

Data Input

WAJP to read and then display employee name, code, salary and employee grade (a single character like A, B, C or D)

```
import java.io.*;
class ReadData
{ public static void main (String args[]) throws IOException
  { InputStreamReader isr = new InputStreamReader (System.in);
    BufferedReader stdin = new BufferedReader (isr);
    System.out.Print("Enter name : ");
    String s1;
    s1 = stdin.readLine();
    S.O.Pln("Name : "+s1);
    S.O.P ("Enter code : ");
int n;
    n = Integer.ParseInt (stdin.readLine());
    S.O.Pln("Code = "+n);
    S.O.P("Enter Salary : ");
    double Sal;
    Sal = Double.parse Double(stdin.readLine());
    S.O.Pln("Salary : "+sal);
    S.O.P("Enter grade : ");
     char grade;
      grade = (char)stdin.read();
       S.O.Pln("Grade:"+grade);
}}
```

123 (User types)
↓ readLine() (does reading)
"123"
↓ ParseInt() (Parse means Convert)
123

I/O :-

```
Enter name : Jack
Name : Jack
Enter Code : 123
Code : 123
Enter Salary : 500.5
Salary : 500.5
Enter Grade : A
Grade : A
```

Above program using Scanner class

(Note) :- The method nextLine() reads string with blank spaces while method next() terminates at occurance of first blank space.

Q.1 Describe the general form of class in java.

1. A class is a collection of some variables and some methods (functions). A class (or class definition or class specification) starts with the keyword class followed by the class name. This is followed by the body of the class. The general form of class definition is as shown:

```
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
        }
} // end of class
```

2. The data items or variables defined inside the class are known as ***instance variables (state)*** and the methods of the class are called ***member methods (behaviour)***. Together they are called ***members*** of the class. The code is contained in these member methods and these methods work on the instance variables.
3. Each object of the class has its own copy of instance variables.
4. The members of the class may optionally be preceded by keywords ***private***, ***public*** ***or protected***.
5. The wrapping (packaging) of instance variables & methods into a single unit called class is known as ***encapsulation.***

Q.2 Give different types of Access Specifiers or Access Modifiers.

1. Java provides us with three Access Specifiers - *private, public* and *protected* – which are sometimes also called modifiers.
2. The class members that have been declared as ***private*** can be accessed only by those member methods which are defined for that particular class.
3. On the other hand, ***public*** members can be accessed from outside the class also. That is, they can be accessed by any method in the program.
4. *protected* access specifier is used only in inheritance.
5. When ***no access specifier is used, then by default*** the member of the class is public within its own package, but cannot be accessed outside of its package. (A package is a collection of classes.). Default access mode is sometimes also called ***friendly access.***

Q.3 Give different types of Non-Access Specifiers or Non-Access Modifiers.
Non-Access Modifiers in Java are: final static abstract transient synchronized native

Demonstration Program on Classes and Objects

```
class Box
{ private double width;
  private double height;
  private double depth;

  public void SetDimensions(double w,
                            double h, double d)
  { width = w;
    height = h;
    depth = d;
  }

  public double Volume()
  { double V;
    V = width * height * depth;
    return V;
  }
}

class BoxMain
{ public static void main(String args[])
  { Box b1 = new Box();
    // Box b1;
    // b1 = new Box();
    Box b2 = new Box();
    b1.SetDimensions(2, 3, 4);
    b2.SetDimensions(5, 6, 2);
    double V1, V2;
    V1 = b1.Volume();
    V2 = b2.Volume();
    S.O.println("Volume of Box1 is" + V1);
    S.O.println("Volume of Box2 is" + V2);
  }
}
```

Note :- Let's say instead of 'width' we use the name 'V', then in Statement (1), we will have to write: "V = this.V * height * depth;", Here 'this.V' corresponds to instance variable 'V'

b1: width = 2, height = 3, depth = 4

b2: width = 5, height = 6, depth = 2

O/P :- Volume of Box1 is 24.0
Volume of Box2 is 60.0

"new" → allocates memory at run time (Dynamic memory allocation)

Demonstrate concept of default access specifier.

```
class Box1
{
   double width, height, depth;

   public double volume()
   {
     double v = width * height  * depth ;
     return v;
   }
}

class Box1Main
{
public static void main(String args[])
{
 Box1 b1 = new Box1();
 Box1 b2 = new Box1();

 b1.width=2;  b1.height=3; b1.depth=4;   (1)

 b2.width=5;  b2.height=6; b2.depth=2;   (2)

 double v1, v2;
 v1 = b1.volume();
 v2 = b2.volume();

 System.out.println("Volume of Box1 is " +v1);
 System.out.println("Volume of Box2 is " +v2);
}
}
```

Note about default Access Specifier: 1) In this program, width, height, depth are default (~~same as public, exception will learn in packages~~). The default access specifier will work the same way as public access specifier till we don't do packages.

2) Statements numbered as (1) and (2) cannot be written in the earlier program since they were 'private' in nature.

Q.4 What is a constructor ?

1. A constructor is a ***special method*** whose main operation is to initialize the objects of its class. A constructor is distinct from other member methods of the class, and it has the ***same name*** as its class.
2. It is invoked(or executed or called) ***automatically*** whenever an object of its class is created(i.e when the class is instantiated). It is called constructor because it constructs the values of instance variables of its class, although it can be used for other purposes also. Hence its main purpose is ***initialization***.
3. The constructor has ***no return type specification (not even void)***. But it can have as many arguments as required.
4. It is also possible to define a class which has no constructor at all. In such a case, the Java compiler adds its own default constructor to the class. This default constructor automatically initializes all instance variables to zero.
5. If a normal member method is written for the task of initialization, then it would be mandatory to invoke (call) this method for each of these objects separately. This would be very inconvenient if there are large number of objects. Thus, constructors reduce the burden on the programmer to specifically initialize data within each object that is created.
6. A constructor can have **any access specifier** (private, public, protected, default) but it **cannot be declared final, static, abstract.**
7. It is valid for a class to have a method with same name as class name and having some return type, but it will not be regarded as a constructor.

```
class BoxCon
{
private double width,height,depth;
Public BoxCon(double w, double h,double d)
{ width=w;
  height=h;
  depth=d;
}
Public double rdume()
{ ... }
}
class BoxConMain
{
Public static void main(string args[])
{
BoxCon b1=new BoxCon(2,3,4);
BoxCon b2=new BoxCon(5,6,2);
...
}
}
```

Imp pts of Constructor :- given above

Note :- ① Advantage of Constructor :-
A constructor is automatically invoked when an object is created. So it saves no of statements. If there are 1000 objects created then we need to write 2000 statements if constructor is not used whereas when constructor is used we need to write only 1000 statements.

Instance variables are automatically initialized to 0

Arguments are also called "signatures"

Q.5 What is method Overloading?

1. Method overloading is a mechanism in which we have more than one methods with same name and different parameter list. The correct method is invoked depending on the ***number & type*** of parameters.
2. On seeing several methods with same name but different parameter list, the C compiler reports an error thinking that the programmer has made a mistake.
3. On the other hand, Java compiler decides which one of these methods will be called depending on the number and data type of parameters mentioned in method call.
4. Overloaded methods may have ***different return types*** and the return type of the method never plays any role in method overloading. Hence, the Java compiler will report an error if two methods have same name, exactly same parameter list but different return types.
5. Overloaded methods may have ***different access specifiers***.
6. Method Overloading is one of the many ways in which Java achieves ***polymorphism***.
7. Given below is an example to demonstrate the concept of Method Overloading:

```
class BoxMO
{
 private  double  width, height, depth;

 public void setDimensions(double w, double h, double d)
 {    width = w;    height = h;    depth  = d;  }

 public void setDimensions(double s)
 {    width = height = depth  = s;  }

 public void setDimensions()
 {    width = height = depth  = 0;  }

 public double volume()
 {
  double v = width * height * depth ;
  return v;
 }
}

class BoxMOMain
{
public static void main(String args[])
{
 BoxMO  b1 = new  BoxMO();
 BoxMO  b2 = new  BoxMO();
 BoxMO  b3 = new  BoxMO();

 b1.setDimensions(2,3,4);
 b2.setDimensions(5);
 b3.setDimensions();

 double v1,v2,v3;
 v1 = b1.volume();       v2 = b2.volume();       v3 = b3.volume();

 System.out.println("Volume of Box1 is " +v1);
 System.out.println("Volume of Box2 is " +v2);
 System.out.println("Volume of Box3 is " +v3);
}}
```

Note:- The compiler reports an error if 2 methods have same name and also the same argument list.

default constructor or no args constructor

Q.6 What are different types of constructors? What is constructor overloading?

1. In practice it may be necessary to initialize various instance variables of different objects with different values when they are created. Java permits us to achieve this objective by passing arguments to the constructor when the objects are created. The constructors that can take arguments / parameters are called ***parameterized constructors***.
2. The parameter list can be specified within parentheses similar to the parameter list in the method.
3. Since Java allows method overloading and since a constructor is basically a method, a constructor with parameters can co-exist with another constructor without parameters.
4. When a constructor does not accept any arguments, it is called ***default constructor***. It is invoked when the object is created with no parameters. If no constructors are defined, then the compiler supplies a default constructor on its own which performs zero initialization.
5. Java also allows us to initialize one object with another object using a constructor known as ***copy constructor***. A copy constructor takes an object reference as parameter and copies one object into another.
6. Constructor overloading is a technique in which a class has more than one constructors with same name but different argument lists.
7. Given below is an example to demonstrate constructor overloading:

```
class BoxConO
{
  private double width,height,depth;

  public BoxConO(double w,double h,double d)
  {   width = w;    height = h;    depth  = d; }

  public BoxConO(double s)
  {    width = height = depth = s; }

  public BoxConO()
  {   width = height = depth = 0; }

  public BoxConO(BoxConO b)
  {
    width = b.width;    height = b.height;
    depth  = b.depth;
  }
  public double volume()
  {
    double v = width * height * depth ;
    return v;
  }
}
class BoxConOMain
{
public static void main(String args[])
{
  BoxConO b1 = new BoxConO(2,3,4);
  BoxConO b2 = new BoxConO(5);
  BoxConO b3 = new BoxConO();
  BoxConO b4 = new BoxConO(b1);
   ...              ...
}}
```

Also called no args constructor

Parameterized constructor

No args constructor (default constructor)

Copy constructor

default constructor or no args constructor

Note: 1) Even if statement (1) is removed, width, height, depth of b3 are still 0. This is because the instance variables are automatically initialized to 0 (not local variables)

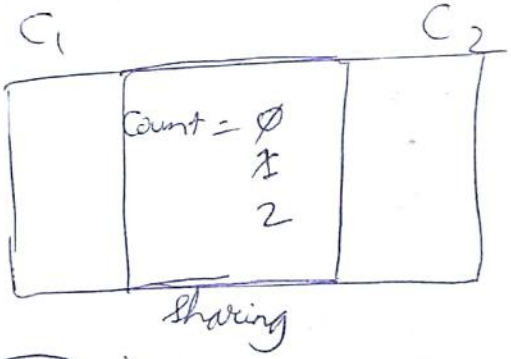
2) The compiler reports an error if the 3rd constructor (no args constructor) is entirely removed.

Q.7 Describe static members.

1. A class in Java can have static instance variables and static methods which are together called static members of class. To create such a member we need to precede its declaration with the word keyword ***static***.
2. Static instance variables are ***not allocated memory separately for each object***. All objects of the same class ***share*** the same copy of the static instance variable.
3. Static Methods have the following properties or restrictions:
   - They can call only other static methods of the class.
   - They can access only static instance variables
   - They cannot use the keywords *this* and *super*.
4. When a member is declared static, it can be accessed even before any object of that class is created and without reference to any object.
5. The most common example of static method is main(). main() is declared static because it is called before any object of the class is created.
6. Static methods and variables can be accessed outside of their class independent of any object. To do so, we just need to specify the name of their class followed by the dot operator. For example, if we wish to call a static method outside its class, we can do so using the following general form: ***classname.method()***
   where *classname* is the name of the class in which the static method is defined.
7. A static member can also be accessed using an object as we access other non-static members.

```
class Counter
{
 private static int count;

 public Counter()
 {    count++ ;  }

 public static void show()
 {
    System.out.print(count);
 }
}

class CounterMain
{
public static void main(String args[])
{
 Counter c1 = new Counter();
 Counter c2 = new Counter();

 c1.show();
 c2.show();
(1) Counter.show();
 }
}
```

O/P :- 2 2 2



Note : 1) If statement (1) was in class Counter, then we can write only show() i.e Classname Counter.show() is not required.

2) Why main is static?

Ans :- If we want to call this program we will have to give the command "java CounterMain" which is actually a method call to method main() of class CounterMain i.e. CounterMain.main(). It should be noted that the method main() is accessed using it's class name because of which it must be kept static.

Q.8 Explain how an object is created in Java. Also explain the concept of assigning object reference variables.

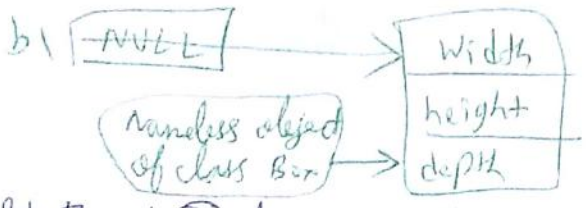
1. Creation of an object in Java is a 2 step procedure. In step 1 we first create a variable of class type. This variable is not in reality an object of the class. In fact, it is simply a variable that can *refer* to an object i.e. it is a reference variable. The value of this reference variable is right now equal to null. i.e. it has nothing.
2. In step 2, we use the new operator to dynamically allocate memory for the object. This newly created object is actually nameless. So what Java does is after creating the object, it stores the address of that object in the reference variable which was created earlier.
3. Now the newly created object can be accessed using the reference variable as if the name of this reference variable is the name of the object.
4. Given below is an example:

Box b1 = new Box();

The above statement can be splited as :-

(1) Box b1; b1 [NULL]

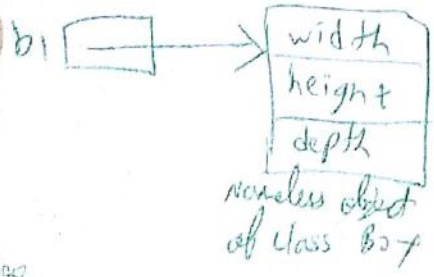
(2) b1 = new Box();



Statement (1) does not create an object called b1, it simply creates a reference variable b1 initialized to NULL. A reference variable is a variable which can store some address. Statement (2) creates a nameless object of the class Box and stores it's address in b1 reference variable b1. It should be noted that b1 is not the name of the object but name of a reference variable.

5. Assignment of object reference variables can be understood with the help of following example.

Box b1 = new Box();



Now

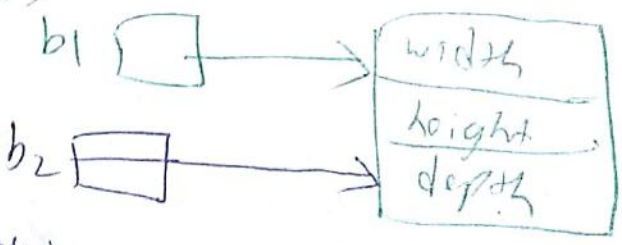
Box b2 = b1;

⇓

Box b2;

b2 = b1;

Now,



Now, if b1.width = 9

then, b2.width also equal to 9

Q.9 What default value is assigned to instance variable ?

The instance variables are automatically initialized to following values:

byte = 0 short = 0 int = 0 long = 0L char = '\u0000'

float = 0.0f double = 0.0d boolean = false String = null

Q.10 Explain the concept of Methods with Variable Argument Lists (var-args)

1. As of Java 5, Java allows us to create methods which can accept variable number of arguments. This feature is sometimes also called "variable-length argument lists" or "variable arguments" or simply "var-args".
2. To declare a method using a var-arg parameter, we must write the data type with an ellipsis, a space and then the name of the array which will hold the parameters received.
3. Its valid to have other parameters in a method that uses a var-arg.
4. The var-arg must be the last parameter in the method's signature.
5. We can have only one var-arg in a method.

```
class VarArg
{
public static void main(String[ ] args)
{
 show('#',true,2,4,6);
 show('%',false,2,4);
}
public static void show(char c , boolean b , int... x)
{
        System.out.print(c) ;
        System.out.print(b) ;
        for(int i=0 ; i<x.length ; i++)
                System.out.print(x[i]) ;
        System.out.println() ;
}
}
```

Output:

```
#true246
%false24
```

State whether valid or invalid:

1. **void dostuff (int x...) { }** – Invalid – Refer Pt 2
2. **void dostuff (int... x , char y) { }** – Invalid – Refer Pt 4
3. **void dostuff (int... x) { }** – Valid
4. **void dostuff (char y , int... x) { }** – Valid
5. **void dostuff (char... y , int... x) { }** – Invalid – Refer Pt 5

Q.11 Explain the finalize() method in Java. Also explain the concept of Garbage collection in Java.

1. In java, objects are allocated memory dynamically at run-time using new operator. In some languages like C++, memory allocated dynamically should be de-allocated manually using the delete operator.
2. However, Java takes a different approach. It handles memory de-allocation completely automatically for the programmer. The mechanism that accomplishes this automatic memory de-allocation is called ***Garbage Collection***.
3. Garbage Collection works like this. When there is no reference variable referring to a particular object it is assumed that this object is no longer needed. Hence, the memory occupied by that object can be taken back.
4. Garbage collection system of java occurs at irregular intervals during the execution of the program. It cannot be specifically told when the garbage collection will take place. **Moreover, it is also possible that Garbage Collection does not take place at all and memory allocated to the object is never de-allocated.**
5. In some programs it is required to perform certain activities just before an object is destroyed by garbage collector. This activity could be something like releasing a resource which was held by that object.
6. These activities can be performed in a method called ***finalize()***. The java run time system calls this method automatically every time it is about to destroy an object.
7. The finalize() method has the following general format:

```
protected void finalize()
{
  // Any last minute activity
}
```

   Here, the keyword protected is an access specifier that prevents access to finalize() code outside its own class.
8. It should be noted that we can never say for sure ***when and whether*** the finalize() method will execute. It may also happen that the finalize() method ***never executes***. This is because the finalize() method executes just before the garbage collector and it is possible that Garbage Collection does not take place at all and memory allocated to the object is never de-allocated.
9. Hence, the program should not rely on finalize() method for any activity. It should have other means which will do the same work.

 instance of operator

Eg → Complex C3 = new Complex C();
C3 instance of Complex ⇒ true
C3 instance of Box ⇒ false

A) Arrays

Q.1 Explain how an array is declared and created in Java.

1. Unlike C, an array in Java is created in two steps. The first step involves creation of an array variable as shown: `type arrayname[];`

   Here ***type*** is the data type of the values which the array can store and ***arrayname*** is the name of the array.
2. Consider the following example where we create an array variable called x.
   `int x[];`

   Although we have created an array, no memory in reality is allocated till now i.e. the variable x is set to null which means it is an array with no value.
3. The second step now involves allocating memory using new operator as shown:
   `arrayname = new type[size];`

   new operator allocates memory equivalent to *size* number of elements.
   For example, we may write `x = new int[20];`

   This statement will now allocate memory for an array x which can store 20 elements of data type int.
4. The two steps described above can be combined as shown:

```
int x[];
x=new int[20];
      ↓
int x[] = new int[20];
```

5. Moreover, instead of writing the statement `int x[];` the same statement can be written as `int [] x`

Q.2 One Dimensional Array initialization.

```
int x[] ={11, 22, 33};
x[0] = 11
x[1] = 22
x[2] = 33
```

We cannot write :- new, array dimensions in array initialization.

Q.3 Creating Multi - Dimensional Array .

(1)
```
int m[][];
m = new int[3][2];
      ↓
int m[][] = new int[3][2];
```

m[0][0]	m[0][1]
m[1][0]	m[1][1]
m[2][0]	m[2][1]

Note :- In Above Statement (1) can also be written as :-
1) `int[][] m`
2) `int [] m[]`

Q.4 Initializing Multi - Dimensional Array .

int m[][] = {{11, 22}, {33, 44}, {55, 66}}; ⇒ This can also be written as

int m[3][2] = {11, 22, 33, 44, 55, 66}
[Syntax error]

11	22
33	44
55	66

Q.5 Jagged Arrays or Variable sized Arrays or Irregular Dimensional Arrays

int m[][] = new int[3][];
m[0] = new int[1];
m[1] = new int[2];
m[2] = new int[1];

m[0][0]	
m[1][0]	m[1][1]
m[2][0]	

Eg :- int d[][] = {{10, 20}, {40, 50, 60, 70}, {80, 90, 100}};

10	20		
40	50	60	70
80	90	100	

Q.6 Explain the arraycopy method.

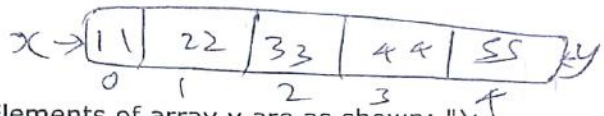
1. In Java class *System* contains a method called *arraycopy* which is used to copy one array into another. The general format of this method is as shown:

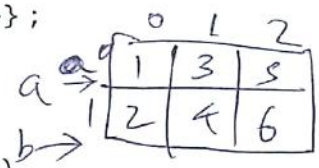
   System.arraycopy(source_array, source_start, target_array, target_start, size)

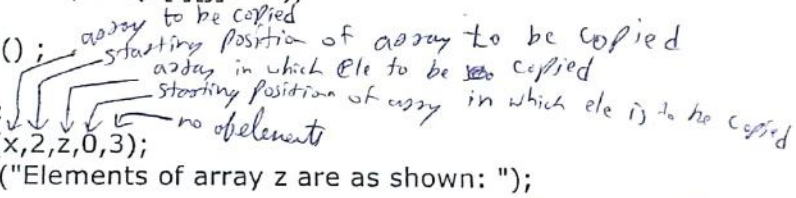
2. This statement will copy *size* no of elements from *source_array* starting from position *source_start* into the *target_array* from position *target_start* onwards.
3. Using this method we can avoid a for loop to copy one array into another.

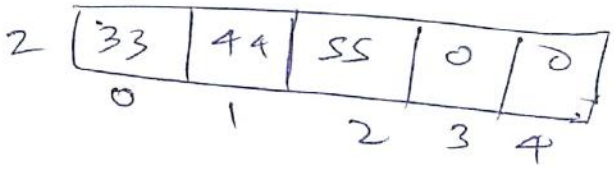
Q.7) WAJP to demonstrate the concept of
1) Array length 3) Array Assignment 2) arraycopy

```
class DemoArray
{
public static void main(String args[])
{
 int x[] = {11,22,33,44,55};

 int  n;
 n = x.length;
 System.out.println("Length of the array x is: "+n);

 int y[ ] ;
 y = x ;
 //  int y[ ] = x ;
 System.out.println("Elements of array y are as shown: ");
 for(int i : y)
       System.out.print(i+" ");

 System.out.println() ;

 int a[ ][ ] = {{1,3,5},{2,4,6}} ;
 int b[ ][ ] ;
 b = a ;
// int b[ ][ ] = a ;
 for(int i = 0 ; i<2 ; i++)
       for(int j = 0 ; j<3 ; j++)
             System.out.print(b[i][j]+" ");

 System.out.println() ;

 int z[]=new int[5];
 System.arraycopy(x,2,z,0,3);
 System.out.println("Elements of array z are as shown: ");
 for(int i : z)
       System.out.print(i+" ");
}
}
```









Output:

Length of the array x is: 5

Elements of array y are as shown: 11 22 33 44 55

1 3 5 2 4 6

Elements of array z are as shown: 33 44 55 0 0

NOTE: Elements of an array are automatically initialized to 0

Q.8) WAJP to demonstrate the concept of Passing an Array to Function.

```
class PassArray
{
public static void main(String args[])
{
 int x[ ] = {7,8,9};
 show1(x) ;
 show2(x) ;

 int y[ ][ ] = {{1,2,3},{4,5,6}};
 show3(y);
 show4(y);
}
 public static void show1(int[ ] x)
 {
  for(int i : x)
        System.out.print(i+" ");
 }
 public static void show2(int... x)
 {
  for(int i : x)
        System.out.print(i+" ");
 }
 public static void show3(int[ ][ ] y)
 {
  for(int i = 0 ; i<y.length ; i++)
        for(int j = 0 ; j<y[i].length ; j++)
              System.out.print(y[i][j]+" ");
 }
 public static void show4(int[ ]... y)
 {
  for(int i = 0 ; i<y.length ; i++)
        for(int j = 0 ; j<y[i].length ; j++)
              System.out.print(y[i][j]+" ");
 }
 }

// public static void show4(int...[ ] y)  ... Wrong Syntax
```

O/P :- 7 8 9
7 8 9
1 2 3 4 5 6
1 2 3 4 5 6

y →	0	1	2
0	1	2	3
1	4	5	6

[This will give no of rows]

[it will give no of columns in row no i]

[y is 2-D array of integers]

Q.9 Explain the Arrays class.

1. In Java, there is a class Arrays which has methods to perform operations on a one dimensional array.
2. All the methods of class Arrays are static.
3. Class Arrays is predefined in package java.util.
4. Of all the methods in class Arrays, the following two are important:

   a. **static void sort (array)**
   Using Quick Sort, this method sorts all array elements in ascending order.

   b. **static int binarySearch(array , element)**
   This method searches the 'element' in the 'array' and returns its position. If the element is not present, it simply returns some negative value.

Demonstration Program:

A class having multiple methods

```
import java.util.* ;
class SortSearchArray
{
public static void main(String args[])
{
 int x[ ] = {7,2,8,4,6};
 Arrays.sort(x);
 for(int i : x)
       System.out.print(i+" ");

 System.out.println() ;

 int j ;
 j = Arrays.binarySearch(x,6) ;
 System.out.print(j+" ");

 System.out.println() ;

 j = Arrays.binarySearch(x,5) ;
 System.out.print(j+" ");
}
}
```

Arrays is in package java.util

Note:- 1) -ve value is returned since no array can have a -ve position

2) If Statement (1) is not written, then Statement (2) will assign 'j' any random -ve value

Output:
2 4 6 7 8
2
-8 (Any random -ve value)

V.V Diff topic

Note:

In Java every array is an object, here when we create a 1-D array say 'x' or a 2-D array say 'a', These x and a are actually addresses of those objects.

Eg:-
```
int x[ ] = {2,4,6};
```

x → | 2 | 4 | 6 |
0 1 2

S.O.P(x); ⇒ This will display starting address of array 'x'

- For 2-D array :-

Eg:
```
int a[2][3] = {1,2,3,4,5,6};
```

a →

	0	1	2
0	1	2	3
1	4	5	6

SOP(a[0]); → This will display starting add of row 0

SOP(a[1]); → " " " row 1

B) Strings

Q.1 What are strings in Java ?

2. A string is simply a sequence of characters in Java. But there is a difference in the way a string is implemented in Java and other programming languages.
3. In other programming languages like C and C++ a string is implemented as an array of characters. However, in Java a string is implemented as objects of a predefined class ***String***.
4. Implementing a string as a built-in object allows Java to provide many features which otherwise would have become a little inconvenient. For example, class String has methods to compare two strings, search for a substring, concatenate two strings and many more. Once we create a string(that is an object of class String) then we can use any of these methods on this object.
5. Similarly, there are many constructors in class String which allow us to create a string in different ways as per our requirement.

Q.2 Explain how a string can be declared and initialized.

Q.3 Examples on how the + and += operator can be used to concatenate strings.

```
String   firstname="Kamaal";
String   lastname="Khan";
String   fullname=firstname+" "+lastname;
System.out.println(fullname);


int  salary = 2;
String    s=fullname +" is a "+salary+" rupees  people ";
System.out.println(s);


String  s1 = "new" ;
String  s2 = "york" ;
s1 += s2 ;
System.out.println(s1) ;
```

Q.4 Explain the most commonly used String methods.

Methods of Class String are as follows:

String length method

```
1)
String s1="Jack";
int n;
n=s1.length();
System.out.println(n);
```

o/p = 4

```
2)
n="Jack".length();
System.out.println(n);
```

o/p = 4

Character Extraction Method

```
char ch;
String s1="Jack";
ch=s1.charAt(3);
System.out.println(ch);
```

o/p : k

String Comparison methods

```
String s1="HODOR";
String s2="hodor";

1)
boolean  x;
x=s1.equals(s2);
System.out.println(x);
```

o/p : false

```
2)
x=s1.equalsIgnoreCase(s2);
System.out.println(x);
```

o/p : true

```
3)
int n;

n="back".compareTo("bank");
System.out.println(n);

n="bank".compareTo("back");
System.out.println(n);

n="bank".compareTo("bank");
System.out.println(n);
```

o/p : -11
11
0

Concept of String Constant Pool

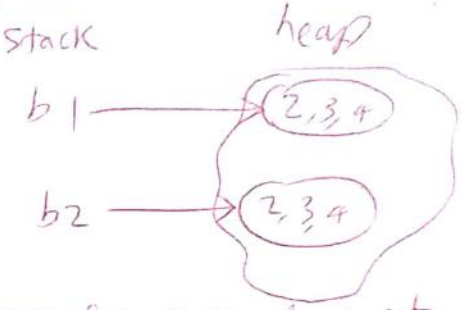
Box b1 = new Box(2,3,4)

Here, b1 is reference variable and is stored in stack
"new Box(2,3,4)" is an object and is stored in heap

Now, Box b2 = new Box(2,3,4);

Here b2 is in stack
"new Box(2,3,4)" is again created in heap

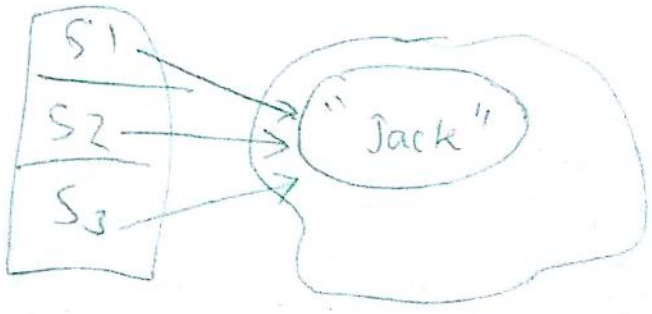
~~Now, let's talk~~



Now, Let's talk about String object :-

String s1 = "Jack";
Now, s1 is stored in stack
"Jack" is stored in String Constant Pool

String s2 = "Jack";
Now, s2 is stored in stack
"Jack" is not created again, instead :-



Searching Strings

```
int p;

1)
p="London".indexOf();
System.out.println(p)
```

O/P = 2

Method returns -1 when search is not successful.

```
2)
p="London".lastIndexOf('n');
System.out.println(p);
```

O/P = 5

```
3)
p="London".indexOf("on");
System.out.println(p);
```

O/P = 1

```
4)
p="London".lastIndexOf("on");
System.out.println(p);
```

O/P 4

Other Methods

```
1)
String s1="Japan";
String s2=new String();
s2=s1.substring(1);
System.out.println(s2);
```

O/P: apan [Till 3 i.e till n-1]

```
2)
s2=s1.substring(1,4);
System.out.println(s2);
```

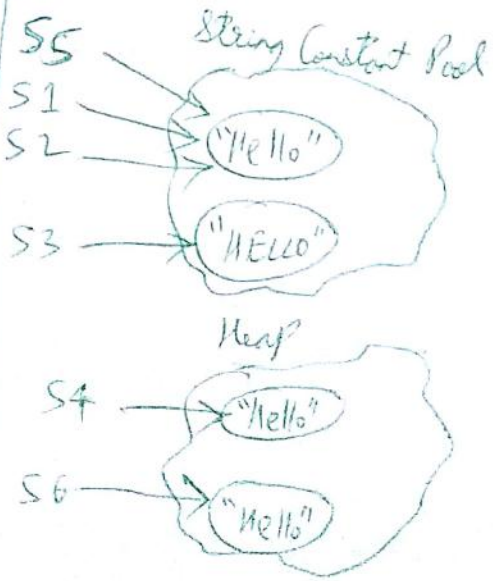
O/P: apa

```
3)
String s1="new";
String s2=new String();
s2=s1.concat("york");
System.out.println(s2);
```

O/P: newyork

```
4)
String s1="circle";
String s2=new String();
s2=s1.replace('c','t');
System.out.println(s2);
```

O/P: tirtle



5)
```
String s1 = new String("  new  york  ");
String s2=new String();
s2=s1.trim();
System.out.println(s2);
```
O/P: ~~Nep~~ new york

6)
```
String  lower="new";
String  upper=new String();
upper = lower.toUpperCase();
System.out.println(upper);
```
O/P: NEW

7)
```
String upper="NEW";
String lower=new String();
lower=upper.toLowerCase();
System.out.println(lower);
```
O/P: new

8)
```
String s1="OCJP";
String s2=new String();
s2=s1.toString();
System.out.println(s2);
```
[toString() will return value of invoking string, and nothing else]

O/P: OCJP

Q.5 How to create string using StringBuffer & StringBuilder class ?

Examples

1) StringBuffer S1 = new StringBuffer("OCAJP");
S.O.Pln(S1);

```
System.out.println(s1);
```
Output: OCAJP

2) StringBuilder S2 = new StringBuilder("OCPJP");
S.O.Pln(S2);

```
System.out.println(s2);
```
Output: OCPJP

Wrong Syntax:
StringBuffer S3 = "OCJP";
StringBuilder S4 = "OCJP";

Both these classes have Same API i.e. same methods.

Q.6 Explain the most commonly used StringBuilder (or StringBuffer) methods.

Methods of Class StringBuilder (or StringBuffer) are as follows:

String length and capacity methods

```
StringBuilder s1=new StringBuilder("Sansa");
int n,m;

1)
n=s1.length();
System.out.println(n);
```

O/P : 5

```
2)
m=s1.capacity();
System.out.println(m);
```

O/P : 21

Character Extraction methods

```
1)
char ch;
StringBuilder s1=new StringBuilder("jack");
ch=s1.charAt(3);
System.out.println(ch);
```

O/P : K

```
2)
StringBuilder s2=new StringBuilder("Banana");
s2.setCharAt(4,'c');
System.out.println(s2);
```

O/P : Banaca

String Modification methods

```
1)
StringBuilder s1=new StringBuilder("Study");
StringBuilder s2=new StringBuilder("Circle");
s1.append(s2);
System.out.println(s1);
```

O/P : StudyCircle

```
2)
StringBuilder msg=new StringBuilder("0123456");
msg.insert(2," followed by ");
System.out.println(msg);
```

O/P : 01 followed by 23456

```
3)
StringBuilder s1=new StringBuilder("rama");
s1.reverse();
System.out.println(s1);
```

O/P : amar

```
4)
StringBuilder s1=new StringBuilder("Mumbai");
s1.delete(0,3);
System.out.println(s1);
```

o/p: bai

```
5)
StringBuilder s1=new StringBuilder("frog");
s1.deleteCharAt(1);
System.out.println(s1);
```

o/p: fog.

```
6)
StringBuilder s1=new StringBuilder("Are you listening ?");
s1.replace(8,14,"watch");
System.out.println(s1);
Are you watching ?
```

o/p: Are you watching?

Q.7 Compare the String, StringBuffer and StringBuilder classes. (Asked in Viva in Placements)

Sr. No.	String	StringBuilder	StringBuffer
1	String objects are **immutable**.	StringBuilder objects are **nutable**.	StringBuffer objects are **mutable**.
2	String objects are allocated memory in **String Constant Pool**.	StringBuilder objects are allocated memory in **Heap**.	StringBuffer objects are allocated memory in **Heap**.
3	String methods are **thread safe**.	ingBuilder methods re **not thread safe**.	StringBuffer methods are **thread safe**.
4	String methods are **synchronized** and hence **slow**.	StringBuilder methods are **not synchronized** and hence **fast**.	StringBuffer methods are **synchronized** and hence **slow**.
5	We can use operators **+, =, +=, == and !=**	We can use operators **== and !=**	We can use operators **== and !=**

Q.8 Explain the heap and stack memory.

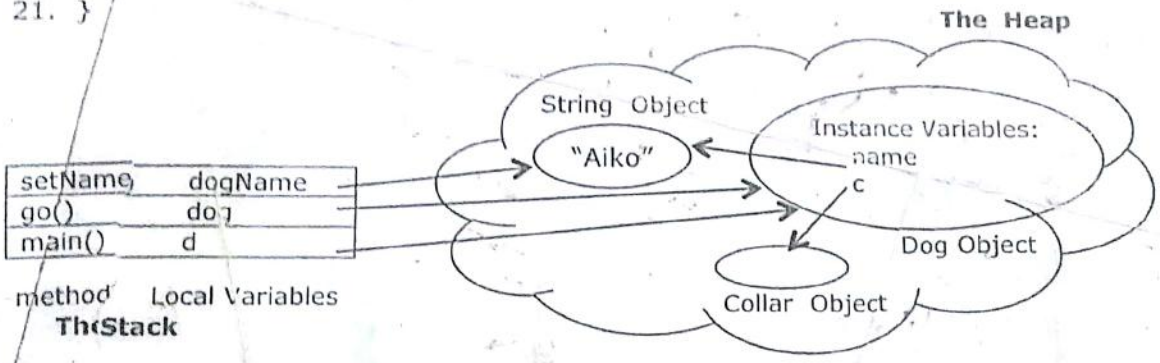
The various pieces (methods, variables a o objects) of a java program live in one of two places in memory: the stack or the heap.

* Instance variables and objects live on the heap:
* Local variables

and method calls live on the stack

Let's take a look at a java program and how its various pieces are created and stored into the stack and the heap:

```
1.  class collar { }
2.
3.  class Dog {
4.  Collar c;                  // instance variable
5.  String name;               // instance variable
6.
7.  public static void main (String [] args)   {
8.
9.      Dog d;                 // local variable: d
10.     d = new Dog ();
11.     d.go (d) ;
12.     }
13.  void go(Dog dog) {    // local variable: dog
14.     c = new Collar ();
15.    dog.setName ("Aiko");
16.  }
17.  void setName ( String dogName) {    // local var:  dogName
18.    name = dogName;
19.    // do more stuff
20.     }
21.  }
```



The above figure shows the state of the stack and the heap once the program reaches line 9. Following are some key points:

- Line 7-main () is placed on the stack.
- Line 9-Reference variable d is created on the stack, but there's no Dog object yet.
- Line 10-A new Dog object is created and is assigned to the d reference variable.
- Line 11-A copy of the reference variable d is passed to the go () method.
- Line 13-The go () method is placed on the stack , with the dog parameter as a local variable.
- Line 14-A new Collar object is created on the heap and is assigned to Dog's instance variable. • Line 15 creates "Aiko" on heap
- Line 17-setName () is added to the stack with the dogName parameter as its local variable.
- Line 18-The name instance variable now also refers to the String object.
- Notice that two different local variables refer to the same Dog object.
- Notice that one local variable and one instance variable both refer to the same string Aiko.
- After Line 19 completes, setName () completes and is removed from the stack. At this point the local variable dogName disappears, too, although the String object it referred to is still on the heap.

Q.7 How String literals (constants) are stored in memory.

1. To make java more memory efficient, the JVM sets aside a special area of memory called the "*String constant pool*".
2. When the complier encounters a String literal, it checks the pool to see if an identical String literal already exists.
3. If a match is found, the reference to the new literal is directed to the existing String literal, and no new String literal object is created. The existing string simply has an additional reference.
4. That's why making string objects immutable is a good idea. If several reference variables refer to the same string without even knowing it, it would be very bad if any of them could change the string's value.

Q.8 When should we use class StringBuilder instead of class String? (For Placements)

1. The java.lang.StringBuilder class should be used when you have to make a lot of modifications to strings of characters.
2. String objects are immutable, so if you choose to do a lot of manipulations with string objects, you will end up with a lot of abandoned String objects in the String pool. Even in these days of gigabytes of RAM, it's not a good idea to waste precious memory on discarded String pool objects.
3. On the other hand, objects of type StringBuilder can be modified over and over again without leaving behind a large number of discarded string objects.
4. A common use for stringBuilders is file I/O when large, ever-changing streams of input are being handled by the program. In these cases, large blocks of characters are handled as units, and stringBuilder objects are the ideal way to handle a block of data, pass it on, and then reuse the same memory to handle the next block of data.

Important Note:

Many of the exam questions covering this chapter's topics use a tricky bit of java syntax known as "chained methods." A statement with chained methods has this general form:

result = method1 (). Method2 (). Method3 ();

In theory, any number of methods can be chained in this fashion, although typically you won't see more than three. Here's how to decipher these "handy java shortcuts" when you encounter them:

1. Determine what the leftmost method call will return (let's call it x).
2. Use x as the object invoking the second (from left) method. If there are only two chained methods, the result of the second method call is the expression's result.
3. If there is a third method, the result of the second method call is used to invoke the third method, whose result is the expression's result. Consider the example,

```
String x = "abc";
String y = x.concat ("def").toUpperCase().replace ('C', 'x' ); // chained methods
System.out.println ("y = " + y);  // result is "y = ABxDEF"
```

Let's look at what happened. The literal *def* was concatenated to *abc*, creating a temporary, intermediate string (soon to be lost), with the value *abcdef*. The toUpperCase() method was called on this string, which created a new (soon to be lost) temporary string with the value *ABCDEF*. The replace() method was then called on this second string object,which created a final string with the value *ABxDEF* and referred *y* to it.

Q.1 What does inheritance mean in Java?

1. The mechanism of deriving (creating) a new class from an old one is called *inheritance* (or *derivation*). The old class is referred to as the *SuperClass* and the new one is called the *Subclass*.
2. Therefore, a subclass is a specialized version of the superclass. It inherits all the instance variables and methods defined by the superclass and adds its own unique elements.
3. Inheritance is mainly used to achieve 1) Code Reusability and 2) Polymorphism
4. Java uses the keyword ***extends*** to inherit one class from another. Given below is an example where class B is a subclass of class A.

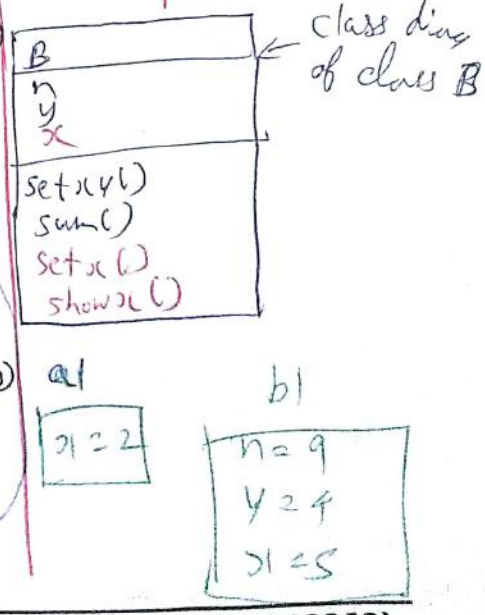
```
class A
{
        ...
        ...
}

class B extends A
{
        ...
        ...
}
```

Q.2 What are the different ways / forms / types of inheritance?

1. The type of inheritance where there is *only* one subclass and *only* one superclass is called *single inheritance.*
2. One superclass may be inherited by more than one subclass. This process is known as *hierarchical inheritance.*
3. The mechanism of deriving a class from another 'subclass' is known as *multilevel inheritance.*
4. In some situations it may be necessary to apply two or more types of inheritance(for example, hierarchical and multilevel) to implement a particular program. This is known as hybrid inheritance.

Single Inheritance:

```
class A
{ public int x;
  public void setx(int a)
  {x=a;}
  public void showx()
  {S.o.Pln(x);}
}
class B extends A
{ private int n,y;
  public void setxy(int a, int b)
  { setx(a); //x=a;
    y=b;}
  public void sum()
  {n=x+y; S.o.Pln(n);}}
class SI
{ public static void main(String args[])
  { A a1=new A();
    a1.setx(2); a1.showx();
    B b1=new B();
    b1.setxy(5,4);
    b1.showx();
    b1.sum();
  }
}
```



Hierarchical Inheritance

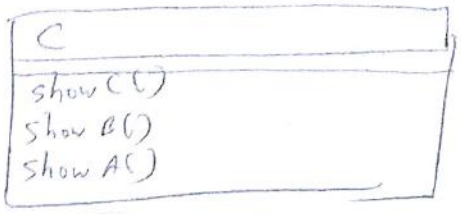
o/p:-
class A
class A
class B
class A
class C

```
class A
{
  public void showA()
  {System.out.println("Class A");}
}

class B extends A
{
  public void showB()
  {System.out.println("Class B");}
}

class C extends A
{
  public void showC()
  {System.out.println("Class C");}
}

class  HI
{
public static void main(String args[])
{
 A  a1 = new  A() ;
 B  b1 = new  B() ;
 C  c1 = new  C() ;

 a1.showA() ;

 b1.showA() ;
 b1.showB() ;

 c1.showA() ;
 c1.showC() ;
}
}
```

Multi-level Inheritance

```
class A
{
  public void showA()
  {System.out.println("Class A");}
}

class B extends A
{
  public void showB()
  {System.out.println("Class B");}
}

class C extends B
{
  public void showC()
  {System.out.println("Class C");}
}

class  MI
{
public static void main(String args[])
{
 A  a1 = new  A() ;
 B  b1 = new  B() ;
 C  c1 = new  C() ;

 a1.showA() ;

 b1.showA() ;
 b1.showB() ;

 c1.showA() ;
 c1.showB() ;
 c1.showC() ;
}
}
```

Class diag of class C



o/p :-

Class A
Class A
Class B
Class A
Class B
Class C

Q.3 Describe behavior of constructors in inheritance.

In inheritance every constructor invokes the default constructor of its superclass using (explicit or implicit) method call ***super()***, unless the constructor invokes an overloaded constructor of the same class using ***this()***.

Rules for Constructors:

1 Constructors can use any access modifier, including private.

2 The constructor name must match the name of the class.

3 Constructors must not have a return type.

4 **If you don't type *any* constructor into your class code, a default constructor will be automatically generated by the complier. This default constructor is ALWAYS a no –arg constructor.**

5 **If you want a no-arg constructor and you've typed any other constructor(s) into your class code, the compiler won't provide the no-arg constructor (or any other constructor) for you. In other words, if you've typed in a constructor with arguments, you won't have a no-arg constructor unless you type it in yourself!**

6 **The first line in a constructor must be a call to super() or a call to this(). If you have neither of those calls in your constructor, the compiler will insert a no-arg call to super() as the first line of constructor.**

7 **One constructor cannot have both the statements – this() and super(). Only one of them is allowed as the first statement.**

8 Abstract classes have constructors , and those constructors are always called when a concrete subclass is instantiated.

9 Inerfaces do not have constructors.

10 The only way a constructor can be invoked is from within another constructor. In other words, you can't write a code that actually calls a constructor as follows:

```
Class Horse
{
   Horse()
   { }
   Void doStuff ()
   {
        Horse ();   // calling the constructor - illegal !
   }
}
```

1)

```
class A
{
 public A()
 { System.out.println("SuperClass"); }
}

class B extends A
{
 public B()
 {
   super() ; // Output same even if this statement is removed. Cant write A();
   System.out.println("SubClass");
   }
}

class SuperCon1
{
public static void main(String args[])
{
 A a = new A() ;
 B b = new B() ;
}
}
```

Here also there is a call "super()" to class object

O/P: SuperClass
SuperClass
SubClass

2)

```
class A
{
 public A()
 {  System.out.println("SuperClass"); }
}

class B extends A
{
      public B()
      {
        super() ;
        System.out.println("SubClass");
      }
      public B(int b)
      {
        System.out.println(b) ;}
}

class SuperCon2
{
public static void main(String args[])
{
 A a = new A() ;
 B b1 = new B() ;
 B b2 = new B(5) ;
}
}
```

How a default super() is present

There is a default call to super() here

O/P: - SuperClass
SuperClass
Subclass
SuperClass
5

O/P: SuperClass
SuperClass
subclass
SuperClass
5

3)
Points 4 and 6

```
class A
{
 public A()
 {  System.out.println("No-Arg Constructor"); }

 public A(int a)
 {  System.out.println(a); }

}

class B extends A
{}

class SuperCon3
{
public static void main(String args[])
{
 B b = new B() ;
}
}
```

B() { super(); } — Here this is optional

O/P :- No-Arg Constructor

4)
Point 7

```
class A
{
public A()
 {  System.out.println("SuperClass"); }
}

class B extends A
{
 public B()
 {
  this(6); // Cant write B(6);
  System.out.println("SubClass");
 }
public B(int b)
 {
  System.out.println(b) ;
 }
}

class SuperCon4
{
public static void main(String args[])
{
 B b1 = new B() ;
}
}
```

Note :- 1) this() is a call, call is a no-arg constructor will call another parameterized constructor of same class no-arg
2) super() will call only default constructor of superclass
3) super() and this() cannot come at same place

Here super() is hidden

O/P :- Superclass
Subclass

O/P :- Superclass
6
Subclass

Another use of this -

```
class A
{ int x;
 public void setx(int x)
 { this.x = x;
 }
}
```

this.x = instance variable, x = local variable

Note :- this. can be used to represent instance variable

5)

Point 5

```
class A
{
  public A(int a)
  {  System.out.println(a); }
}

class B extends A
{
public B()
{  System.out.println("SubClass"); }
}
class SuperCon5
{
public static void main(String args[])
{
  B  b = new B() ;
}
}
```

O/P :- error

Note :- 1) The default super() will try to execute no-arg constructor of class A, which is absent

2) What is the O/P if statements (1) and (2) are removed.

O/P :- Still error

3) If we want to overcome to error, we should do one of these :-

i) remove parameter constructor from class A

or

ii) add a no-args constructor in class A

The following table Shows what the compiler will (or won't) generate for your class.

Class Code (What You Type)	Compiler-Generated Constructor Code (in Bold)

class Foo { }	Class Foo {
	Foo ( ) {
	super ( ) ;
	}
	}

class Foo {
Foo ( ) { }
}
	class Foo {
	Foo () {
	super () ;
	}
	}

public class Foo { }	public class Foo {
	public Foo () {
	super () ;
	}
	}

class Foo {
Foo (string s) { }
}
	Class Foo {
	Foo (string s) {
	super () ;
	}
	}

Class Foo {
Foo (string s)
{
Super () ;
}
}
	Nothing.
	Compiler doesn't need to insert anything.

Class Foo {
void Foo () { }
}
	Class Foo {
	Void Foo () { }
	Foo () {
	Super () ;
	}
	}
	
	Note:
	void Foo() is a method, not a constructor.

Note :- access specifier of class and auto generated constructor will be same.

this is a method

Note :- > Method name and class name can be same even if it is not constructor.

Q.4 Explain what is method Overriding?

1. In Inheritance, when a method in a subclass has same name, same argument-list (signature) and same return type as the method in the superclass then the subclass method is said to override the superclass method and the technique is known as method overriding.
2. Hence, if an overridden method (superclass method) is called from within the subclass, the call will always refer to subclass method thereby hiding the method of the superclass.
3. If the method call is outside the superclass as well as subclass then the ***correct method to be invoked is determined as per the invoking object***. If the object belongs to the superclass then superclass method is invoked. If the object belongs to the subclass then subclass method is invoked.
4. Method Overriding occurs only when method name and argument lists are same. If the argument lists are different, then it would become method overloading. Moreover, of the two methods involved in overriding one should be in the superclass and one should be in the subclass.

```
class A
{
 public void show()
 {System.out.println("Class A");}
}

class B extends A
{
 public void show()
 {
      System.out.println("Class B");
      super.show();
 }
}

class  MOR1
{
public static void main(String args[])
{
 A  a1 = new  A() ;
 B  b1 = new  B() ;

 a1.show() ;
 b1.show() ;
}
}
```

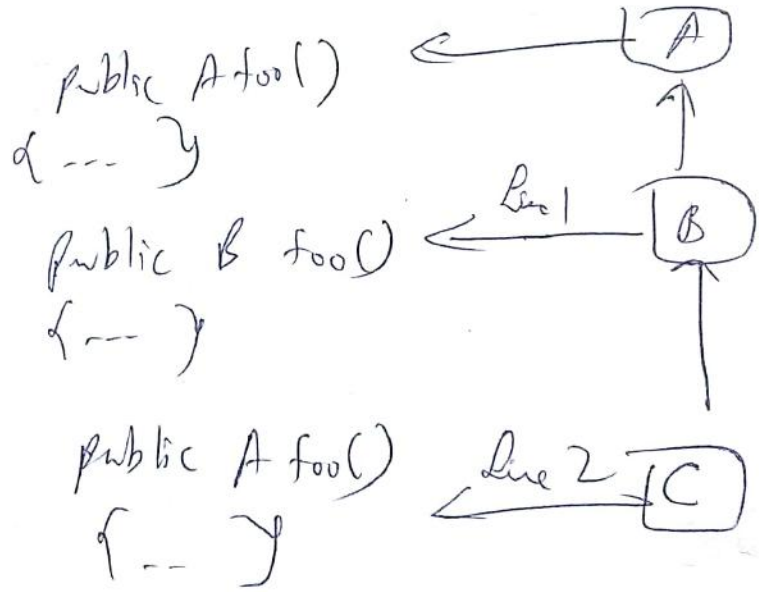
Rules of method overriding [Heavily Imp for exam]

1. The argument list must exactly match that of the overriding method. If they don't match, you can end up with an overloaded method you didn't intend.
2. The return type must be the same as, or a subtype of, the return type declared in the original overridden method in the superclass. If name and argument list is same, but return type condition is violated, then Compiler reports error.
3. The access level can't be more restrictive than that of the overridden method. The access level CAN be less restrictive than that of the overridden method.

   private → default → protected → public
4. If a method is private in superclass, we may have a method in subclass with same name, argument list and return type. However, this will not be considered method overriding.
5. You cannot override a method marked final. Compiler reports error.
6. If you have two methods with same name, argument list and return type in superclass and subclass, and if one of them is static then the other must be static too. Otherwise, compilation fails. What is important is that even if both are static, it is not considered method overriding.

Consider the following:

```
class A
{
  public void doo()
  { }

  public A foo()
  { return new A() ; }

  protected void moo()
  { }

  private void zoo()
  { }

  public static void loo()
  { }
}

class B extends A
{
  // Line 1
}

class C extends B
{
  // Line 2
}
class MOR3 { public static void main(String args[]) { } }
```



Example of Legal and Illegal overrides at Line 1	
public int doo() { }	Invalid. Because return type differ - Rule 2 is violated
public B foo() { return new B() ; }	Valid. Because return type B is subclass of A
public A foo() { return new A() ; }	Valid
public void moo() { }	Valid. Because access specifier can be less restrictive
void moo() { }	Invalid. Because attempting to assign weaker access privileges — Rule 3
private void zoo() { }	Valid. But this is not method overriding
public static void loo() { }	Valid. But this is not method overriding
public void loo() { }	Invalid. Method in superclass is static — Rule 6
Example of Legal and Illegal overrides at Line 2	
public A foo() { return new A() ; }	Note: 1) Invalid if public B foo() { return new B(); } is present in class B - Otherwise Valid 2) Public whether C foo() { return new C(); } is valid public B foo() { return new B(); } is present or absent

Q.5 State rules of method overloading.

1	Overloaded methods MUST change the argument list.
2	Overloaded methods CAN change the return type.
3	Overloaded methods CAN change the access modifier.
4	A method can be overloaded in the same class or in a subclass. In other words, if class A defines a ***doStuff(int i)*** method, the subclass B could define a ***doStuff(String s)*** method without overriding the superclass version that takes an int. So two methods with the same name but in different classes can still be considered overloaded if the subclass inherits one version of the method and then declares another overloaded version in its class definition.

Q.6 State differences between method overloading and method overriding.

	Overloaded method	Overridden method
Argument list	Must change	Must not change
Return type	Can change	Can't change except for covariant returns.
Access Modifier	Can change	Must not make more restrictive (can be less restrictive).
Place	A method can be overloaded in the same class or in the subclass.	A method can be overridden in the subclass.
Invocation	Reference type determines which overloaded version (based on declared argument types) is selected. Happens at compile time.	Object type determines which method is selected. Happens at runtime.

```
class A
{
  public void show()
  { System.out.println("Class A"); }
}

class B extends A
{
  public void show()
  {    System.out.println("Class B"); }

  public void show(int x)
  {          System.out.println(x);    }
}

class  MORMOL
{
public static void main(String args[])
{
 A a1 = new A() ;
 B b1 = new B() ;

 a1.show() ;
 b1.show() ;

 //a1.show(5) ; ERROR
 b1.show(6) ;
}
}
```

O/P :- class A
class B
6

This is error bcos object of class A cannot use method of class B but vice versa is possible
i.e obj of subclass can use super class method but vice versa is not possible.

Q.7 Explain IS-A and HAS-A relationship.

1. In java, a program may have multiple classes and any two classes may be related to each other in several ways. One such relationship between two classes is that of inheritance. Inheritance is also called **"is-a"** relationship since a subclass ***is a*** specialized form of its superclass.
2. Another way through which two classes can be related to each other is ***Association*** which is also called **"has-a"** relationship. In java, Association occurs when one class (or object) ***has an*** instance variable which is of another class. Association is sometimes also called "class containership".
3. Consider the following example

```
class Date
{
  int  d , m , y ;
}
class Account {...}
class Current extends Account
{
   ...
   ...
   Date   account_opening_date ;
}
```

Account and Current – IS-A relationship.

Date and Current – HAS-A relationship.

Q.8 Explain upcasting and downcasting.

1. Sometimes it may be required to convert a variable's type from one class to another class. This is however possible only if the classes are related to each other by way of inheritance.
2. Converting a superclass variable to subclass type is called downcasting and converting subclass variable to superclass type is called upcasting.

```
class Animal { }

class Dog extends Animal { }

class Cat extends Animal { }

class Casting
{
public static void main(String args[])
{
      // Normal Reference
      Animal a = new Animal();
      Dog d = new Dog();
      Cat c = new Cat();

      // Polymorphic reference
      Animal a1 = new Dog();
      Animal a2 = new Cat();

      Line 1
}}
```

rule(1) → In casting, there should be relation of Inheritance

rule(2) → Converting superclass type to subclass is not possible except (eg 14) or next page

rule(3) → cast operator should be used while casting

C → Compilation ; S → Successfull
R → Run ; F → Fail

No.	Line 1	Result
1	a=d;	CSRS. Upcasting
2	a=(Animal)d;	CSRS. Upcasting
3	a=c;	CSRS. Upcasting
4	a=(Animal)c;	CSRS. Upcasting
5	d=a; c=a;	CF. Incompatible types: Animal cannot be converted to Dog
6	d=(Dog)a; c=(Cat)a;	CSRF. Class Cast Exception - Animal cannot be cast to Dog/Cat
7	d=c;	CF. Incompatible types: Cat cannot be converted to Dog
8	d=(Dog)c;	CF. Incompatible types: Cat cannot be converted to Dog.
9	a=a1; a=a2;	CSRS
10	a1=a; a2=a;	CSRS
11	a1 = d;	CSRS. Upcasting
12	a2 = c;	CSRS. Upcasting
13	d=a1;	CF. Incompatible types: Animal cannot be converted to Dog
14	d=(Dog)a1;	CSRS. Valid Downcasting, Copy of No (6)
15	a1 = (Animal) d;	CSRS. Upcasting
16	a2 = (Animal) c;	CSRS. Upcasting.

a → Animal object
d → Dog object
c → Cat object
a1 → Dog object
a2 → Cat object

Consider eg, d = a; → Here, Cast operator is not used, also a cannot be casted to d as a contain animal obj
d = a1; → Here, Cast operator is not used.
d = (Dog) a; → Here, a cannot be casted to d as a contain animal obj
d = (Dog) a1; → Here, everything is valid

Q.9 Explain what is polymorphism.

Polymorphism is one of the important characteristics of object-oriented programming. It means '***one name many tasks***' or '***same name different tasks***'. Which task is to be performed is decided by the compiler from the context. Polymorphism can be achieved in Java by many ways.

Compile-time polymorphism:

1. Method overloading is an example of compile-time polymorphism. It can be considered as polymorphism since we have methods with ***same name performing different tasks***.
2. In method overloading, the correct method to be executed is selected at compile-time by looking at the arguments in the method call and matching the same with the arguments / parameters of the appropriate method.
3. The compiler takes the ***decision*** of which method to execute at ***compile time***. Therefore such polymorphism is known as *compile time polymorphism*. It is also called as *early binding or static binding*.

Method overriding must happen for polymorphic ref

Run-time polymorphism:

1. Method Overriding along with polymorphic reference (Dynamic Method Dispatch) is an example of run-time polymorphism. In polymorphic reference, the correct method to be executed is selected at run-time, rather than compile-time.
2. Method overriding is the basis of polymorphic reference. In polymorphic reference, a superclass reference variable stores address of objects of subclass. When an Overridden method is called through a superclass reference variable, Java selects the method to be executed at run-time depending on the object whose address is stored in superclass reference variable.
3. When different types of objects are referred to, different versions of an overridden method will be called. In other words, ***it is the content of reference variable and not data type of the reference variable that determines which version of an overridden method to execute.***
4. Hence, by making the same reference variable refer to different subclass objects we can use the same method call to execute different methods. This is java's way of implementing run-time polymorphism. Here, the word polymorphism refers to the fact that the same method call is capable of executing different methods.

1)

```
class A
{
 public void show()
 {   System.out.println("Class A"); }
}

class B extends A
{
 public void show()
 {   System.out.println("Class B"); }
}

class C extends A
{
 public void show()
 {   System.out.println("Class C"); }
}

class Ref1
{
public static void main(String args[])
{
 B  b=new B();
 C  c=new C();

 A  a;

 a=b;
 a.show();

 a=c;
 a.show();
 }
 }
```

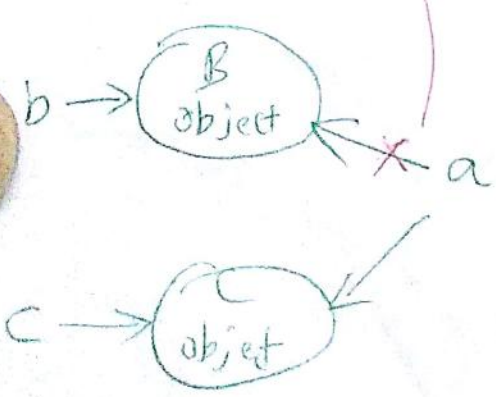
Note :- Statements ① and ② look exactly same but they produce diff outputs. This is Polymorphism. Precisely speaking this is Run time Polymorphism since selection of method show() is done at run time.

this is called Polymorphic reference.

[Applicat^n :- Client Server
Here, a is server and b, c are clients]

O/P :- ClassB
ClassC

due to



2)

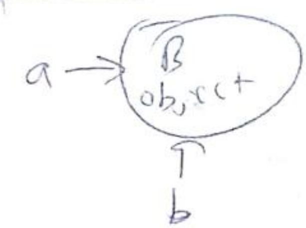
```
class A
{
  public void show()
  {    System.out.println("Class A"); }
}

class B extends A
{
  public void show()
  {    System.out.println("Class B"); }

  public void showB()
  {    System.out.println("showB "); }

}

class Ref2
{
public static void main(String args[])
{
 A  a = new  B() ;
 a.show(); 
 // a.showB(); ERROR

  B  b ;
  b = (B) a ;
  b.showB() ;
 }
}
```

a.show(); → o/p:- class B

// a.showB(); ERROR → [Variable of superclass cannot invoke method of subclass]

Valid downcasting Refer → b = (B) a ;

o/p :- class B
showB



3)

```
class A
{
  public static void show()
  {    System.out.println("Class A") ; }
}

class B extends A
{
 public static void show()
 {    System.out.println("Class B") ; }

}

class RefStatic
{
public static void main(String args[])
{
 A  a = new  B() ;
 B  b = new  B() ;
 a.show();
 b.show();
}
}
```

A a = new B() ; → o/p :- class A

B b = new B() ; → o/p :- class B

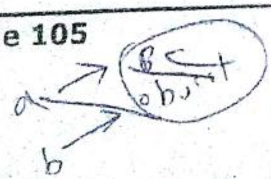
[Bcoz of static, Method overriding doesn't take place]

o/p :- Class A
Class B

V. Imp

[Becoz of static, there is no method overriding so the concept of polymorphic reference is not applicable & now (datatype of a and b) is checked not the contents of a and b]

Sandeep J. Gupta (9821882868)



Q.10 Explain what are abstract classes and abstract methods?

1. Sometimes it is required that the subclass should ***compulsorily*** override a method given in the superclass i.e. the subclass cannot use the version of the method defined in the superclass. In such cases the method of superclass is declared to be abstract using the following general form:

   abstract return-type function-name(parameter-list);

   It should be noticed that no function body is present since an abstract method is never used by an object.

2. A class which contains one or more abstract methods must also be declared abstract. This is done using the keyword abstract in front of the keyword class at the beginning of class declaration. For example,

```
abstract class shape
{
        ...
        abstract void area();
        ...
}
```

is an abstract class having abstract method area().

Rules of abstract class:

1. A class which contains one or more abstract methods must also be declared abstract.
2. An abstract class cannot be instantiated i.e. we cannot create its objects using new operator, although we can create its reference variable.
3. A constructor or a static method cannot be declared abstract.

```
abstract class A
{
  abstract public static void display();  // INVALID

  abstract A () { } // INVALID

  A() { } // VALID
}
```

4. Any subclass of an abstract class must either implement all of the abstract method of the superclass or be itself declared abstract.
5. We may have an abstract class which does not have any abstract method.

```
// VALID CLASS
abstract class A
{
  public void show()
  { System.out.println("SuperClass"); }
}
```

6. A method cannot be declared both final and abstract.

```
abstract class A {
  abstract private void display();  // INVALID }
```

7. A method cannot be declared both private and abstract.

```
abstract class A {
  abstract final void display();  // INVALID }
```

8) Abstract methods do not have body.

1)

```
abstract class A
{
  abstract public void display();     // public and abstract can be written in any order

  public void show()
  { System.out.println("SuperClass"); }
}

class B extends A
{
 public void display() // ERROR IF ITS NAME IS show()
 {
  System.out.println("SubClass");
 }
}

class  Abstract1
{  public static void main(String args[]) {
  // A  a=new A();  Error
   B  b=new B();
   b.display();
   b.show();
} }
```

2) Points 2 and 4

```
abstract class A
{
  abstract public void showA() ;
}

abstract class B extends A
{
  public abstract void showB() ;
}

class C extends B
{
  public void showA()
  { System.out.println("A"); }

  public void showB()
  { System.out.println("B"); }
}

class  Abstract3
{
public static void main(String args[])
{
  B  b = new C();
  b.showA();
  b.showB();
}
}
```

Q.11 Describe the usage of the word final in inheritance.

The keyword final is called modifier in general. In inheritance, it is used to create final method and final class.

a) final method:

1. Although method overriding is a very useful feature of java, there will be certain situations when we would want to prevent it from happening. To disallow a method from being overridden we should specify the word final as a modifier at the beginning of the method.
2. Hence, a final method is that method which cannot be overridden.
3. For example, the following program segment will report a compile-time error since method show() of class A cannot be overridden:

```
class A
{
    final void show()
    {
        System.out.show("This is final method");
    }
}
class B extends A
{
    void show()
    {
        System.out.show("This will report an error");
    }
}
```

b) final class:

1. A final class is that class which cannot be inherited. To prevent a class from being inherited, we just need to precede the class declaration with the word final.
2. For example, the following program segment will report a compile-time error since class B tries to inherit class A which is declared to be final:

```
final class A
{
    ...
}
class B extends A
{
    ...
}
```

3. Declaring a class as final also automatically makes all its methods final.
4. Moreover, it is obvious that a class cannot be declared both abstract and final since an abstract class is incomplete by itself and relies upon its subclasses for complete implementation.

Q.12 What is an interface? How do we define an interface?

1. An interface is similar to a class. Like a class it also has two types of members, that is, instance variables and methods. However, there are two differences between a class and an interface.
2. The first difference is that the methods of the interface are ***implicitly abstract*** i.e. they have no bodies. Since the method is abstract any class implementing the interface should compulsorily override the method.
3. The second difference is that the variables declared inside the interface are ***implicitly final and static.*** This means that these variables cannot be changed by the implementing class and that's why it is compulsory to initialize all of them to some constant value.
4. The general form of an interface is as shown:

```
interface InterfaceName
{
   data-type  final-variable1-name=value;
   data-type  final-variable2-name=value;
   ...
   ...
   data-type  final-variablen-name=value;

   return-type  method1-name(parameter list);
   return-type  method2-name(parameter list);
   ...
   ...
   return-type  methodn-name(parameter list);
}
```

5. Here, ***interface*** is the keyword which should be followed by the name of the interface. All the variables and methods of the interface are implicitly public.
6. Moreover, it should be noted that methods end with semicolon with no body since they are abstract.
7. Consider the following example:

```
interface A
{
 int x = 5 ;
 void show() ;
}

class B implements A
{
 public void show()
 { System.out.println(x); }
}

class Interface1
{
public static void main(String args[])
{
 B b = new B();
 b.show();
}
}
```

o/p :- 5

8. Since Java does not allow multiple inheritance, we can use interfaces to achieve many of the goals of multiple inheritance.

```
interface A
{
 int x = 5 ;
 void show() ;
}
interface B
{
 int x = 6 ;
 void show() ;
}
abstract class C
{
 final static int x = 7 ;
 abstract public void show() ;
}

class Interface2 extends C implements A , B
{
 public void show()
 { System.out.println(A.x + " " + B.x + " " + C.x); }

 public static void main(String args[])
 {
  Interface2 i = new Interface2();
  i.show();
 }
}
```

o/p = 5 6 7

Rules:

1. Interface members (methods and variables) cannot be declared private or protected. They are implicitly public.
2. Interface methods cannot be declared final or static since they are implicitly abstract.
3. The variables of an interface cannot be left uninitialized.

```
interface A
{
 int x ;  // INVALID
 void show() ;
}
```

4. That subclass method which overrides the abstract method of interface must be kept public.

```
interface A
{ void show() ; }
class B implements A {
 void show() // INVALID
 { ... }
}
```

5. An interface cannot be instantiated i.e. we cannot create its objects using new operator, although we can create its reference variable.
6. An interface can be inherited from another interface. The new sub-interface will inherit all the members of the super-interface.

```
interface A
{ ... }
interface B extends A
{ .. }
```

7. The following are legal & illegal examples of class and interface declarations:

```
class Foo { }                                   // OK
Class Bar implements Foo { }                    // No! can't implement a class
interface Baz { }                               // OK
interface Fi { }                                // OK
interface Fee implements Baz { }                // No! an interface can't implement an interface
interface Zee implements Foo { }                // No! an interface can't implement a class
interface zoo extends Foo { }                   // No! an interface can't extend a class
interface Boo extends Fi { }                    // OK. An interface can extend an interface
class Toon extends Foo, Button { }              // No! a class can't extend multiple classes
class Zoom implements Fi, Baz { }               // OK. A class can implement multiple interfaces
interface Vroom extends Fi, Baz { }             // OK. An interface can extend multiple interfaces
class Yow extends Foo implements Fi { }         // OK. A class can do both (extends must be 1st)
class Yow extends Foo implements Fi, Baz { }    // OK. A class can do all 3 (extends must be 1st)
```

Difference between abstract class and interface.

No.	Interface	Abstract Class
1	All methods in an interface are abstract by default.	In an abstract class, all methods need not be abstract.
2	All instance variables in an interface are final and static by default.	In an abstract class, instance variables need not be final and static compulsorily.
3	A subclass can be implemented from more than one interfaces.	A subclass cannot be inherited from more than one abstract classes.
4	No member of an interface can be declared private.	Non-abstract members of an abstract class can be declared private.
5	Since instance variables are final, they must be compulsorily initialized.	Instance variables need not be initialized.
6	Keyword "implements" is required to derive a class from an interface.	Keyword "extends" is required to derive a class from an abstract class.
7	Interface cannot have constructors.	Abstract class can have constructors.
8	Give one example program for each of them	

Q.13 What are initialization blocks?

1. In a class, the code that performs operations is normally put in two places: methods and constructors. Initialization blocks are the third place in a java program where operations can be performed.
2. Initialization blocks run when the class is first loaded (a static initialization block) or when an instance is created (an instance initialization block).
3. Let's look at an example:

```
class SmallInit
{
  static int x ;

  int y ;

  static { x = 7 ; }        // static init block

  { y = 8; }                // instance init block
}
```

4. As you can see, the syntax for initialization blocks is pretty concise. They don't have names , they can't take arguments, and they don't return anything.
5. A static initialization block runs once, when the class is first loaded . An instance initialization block runs once every time a new instance(object) is created.
6. Remember when we talked about the order in which constructor code executed? Instance init block code runs right after the call to super() in a constructor – in other words, after all super constructors have run.
7. You can have many initialization blocks in a class. It is important to note that unlike methods or constructors, the order in which initialization blocks appear in a class matters.
8. When it's time for initialization blocks to run, if a class has more than one, they will run in the order in which they appear in the class file - in other words, from top to bottom.
9. Based on the rules we just discussed, lets determine the output of the following programs:

```
class InitBlock3
{
  private static int a , b , c ;

  { a = 2 ; b = 4 ; }

  static { c = a + b ; }

  public void show()
  { System.out.println (c) ; }

  public static void main (String [] args )
  {
    InitBlock3 i = new InitBlock3() ;
    i.show() ;
  }
}
```

O/P :- 0

Page 112

```
class Init
{
Init(int x)
{ System.out.println ("1-arg const") ; }

Init()
{ System.out.println ("no-arg const") ; }

static { System.out.println ("1st static init") ; }

{ System.out.println ("1st instance init") ; }

{ System.out.println ("2nd instance init") ; }

static { System.out.println ("2nd static init") ; }

public static void main (String [] args )
{
  new Init () ;
  new Init (7) ;
}
}
```

To figure this out, remember these rules:

1. init blocks execute in the order in which they appear.
2. Static init blocks run once, when the class is first loaded.
3. Instance init blocks run every time a class instance is created.
4. Instance init blocks run after the constructor's call to super()