1. **Few Facts of Java-**

* Java is Object Oriented. However, it is not considered as pure object oriented as it provides support for primitive data types (like int, char, etc)
* The Java codes are first compiled into byte code (machine independent code). Then the byte code is run on Java Virtual Machine (JVM) regardless of the underlying architecture.
* Java syntax is similar to C/C++. But Java does not provide low level programming functionalities like pointers. Also, Java codes are always written in the form of classes and objects.

1. **JDK, JRE and JVM?**

Ans- JVM, JRE and JDK all three are platform dependent because configuration of each Operating System is different. But Java is platform independent.

* **JDK (Java Development Kit)** : The Java Development Kit (JDK) is a software development environment used for developing Java applications and applets.

It includes the Java Runtime Environment (JRE), an interpreter/loader (java), a compiler (javac), an archiver (jar), a documentation generator (javadoc) and other tools needed in Java development.

* **JRE (Java Runtime Environment) :** JRE contains the parts of the Java libraries required to run Java programs and is intended for end users. JRE can be view as a subset of JDK.

It combines the Java Virtual Machine (JVM), platform core classes and supporting libraries.

JRE is part of the Java Development Kit (JDK) but can be downloaded separately. JRE was originally developed by Sun Microsystems Inc., a wholly-owned subsidiary of Oracle Corporation

* **JVM**: JVM (Java Virtual Machine) is an abstract machine. It is a specification that provides runtime environment in which java bytecode can be executed line by line. JVMs are available for many hardware and software platforms.

1. **Important Features of Java**

Ans-

* **Simple**
* **Platform Independent**
* **Architectural Neutral:** A Language or Technology is said to be Architectural neutral which can run on any available processors in the real world without considering their development and compilation.
* **Portable:** If any language supports platform independent and architectural neutral feature known as portable.
* **Multi-Threading:** A flow of control is known as a thread. When any Language executes multiple thread at a time that language is known as multithreaded e. It is multithreaded.
* **Distributed:** Using this language we can create distributed applications. In distributed application multiple client system depends on multiple server systems so that even problem occurred in one server will never be reflected on any client system.
* **Networked:** It is mainly designed for web-based applications, J2EE is used for developing network-based applications.
* **Robust:** Simply means of Robust are strong. It is robust or strong Programming Language because of its capability to handle Run-time Error, automatic garbage collection
* **Dynamic:** It supports Dynamic memory allocation due to this memory wastage is reduce and improve performance of the application.
* **Secured:** It is a more secure language compared to other language; In this language, all code is covered in byte code after compilation which is not readable by human.
* **Object Oriented:** It supports OOP's concepts because of this it is most secure language

1. **Naming conventions in Java?**

Ans- They must be followed while developing software in java for good maintenance and readability of code. Java uses CamelCase as a practice for writing names of methods, variables, classes, packages and constants.

Classes and Interfaces- First letter of each word should be capitalized.

Methods- first letter should be in lowercase and first letter of each word capitalized.

Variables- Should not start with “$” and “\_”. One-character variable names should be avoided for temporary variables.

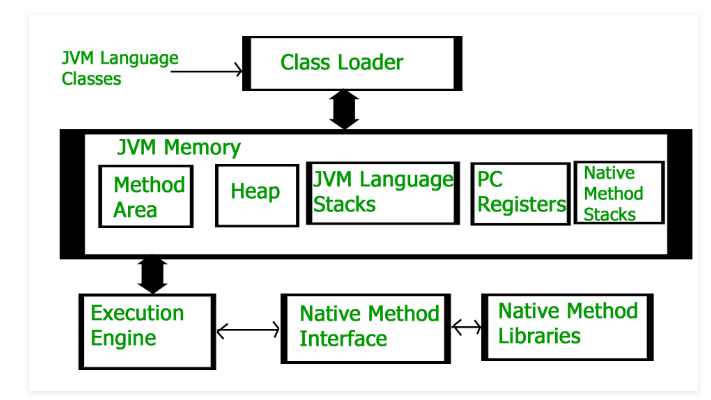
Constant- Should be all uppercase and separated by “\_”

Package- should be in lowercase.

1. **How JVM works?**

Ans- JVM (Java Virtual Machine) acts as a run-time engine to run Java applications. JVM is the one that actually calls the **main** method present in a java code. JVM is a part of JRE(Java Runtime Environment).

Java applications are called WORA (Write Once Run Anywhere). This means a programmer can develop Java code on one system and can expect it to run on any other Java enabled system without any adjustment. This is all possible because of JVM.

When we compile a *.java* file, *.class* files(contains byte-code) with the same class names present in *.java* file are generated by the Java compiler. This *.class* file goes into various steps when we run it. These steps together describe the whole JVM. 

**Class Loader Subsystem**  
It is mainly responsible for three activities.

* Loading
* Linking
* Initialization

**Loading:** The Class loader reads the *.class* file, generate the corresponding binary data and save it in method area. For each *.class* file, JVM stores following information in method area.

* Fully qualified name of the loaded class and its immediate parent class.
* Whether *.class* file is related to Class or Interface or Enum
* Modifier, Variables and Method information etc.

After loading *.class* file, JVM creates an object of type Class to represent this file in the heap memory.

This Class object can be used by the programmer for getting class level information like name of class, parent name, methods and variable information etc

***Student s1 = new Student();***

***// Getting hold of Class object created by JVM.***

***Class c1 = s1.getClass();***

***// Printing type of object using c1.***

***System.out.println(c1.getName());***

***// getting all methods in an array***

***Method m[] = c1.getDeclaredMethods();***

***for (Method method : m)***

***System.out.println(method.getName());***

***// getting all fields in an array***

***Field f[] = c1.getDeclaredFields();***

***for (Field field : f)***

***System.out.println(field.getName());***

**Note:** For every loaded *.class* file, only **one** object of Class is created.

Student s2 = new Student();

// c2 will point to same object where

// c1 is pointing

Class c2 = s2.getClass();

System.out.println(c1==c2); // true

**Linking :** Performs verification, preparation, and (optionally) resolution.

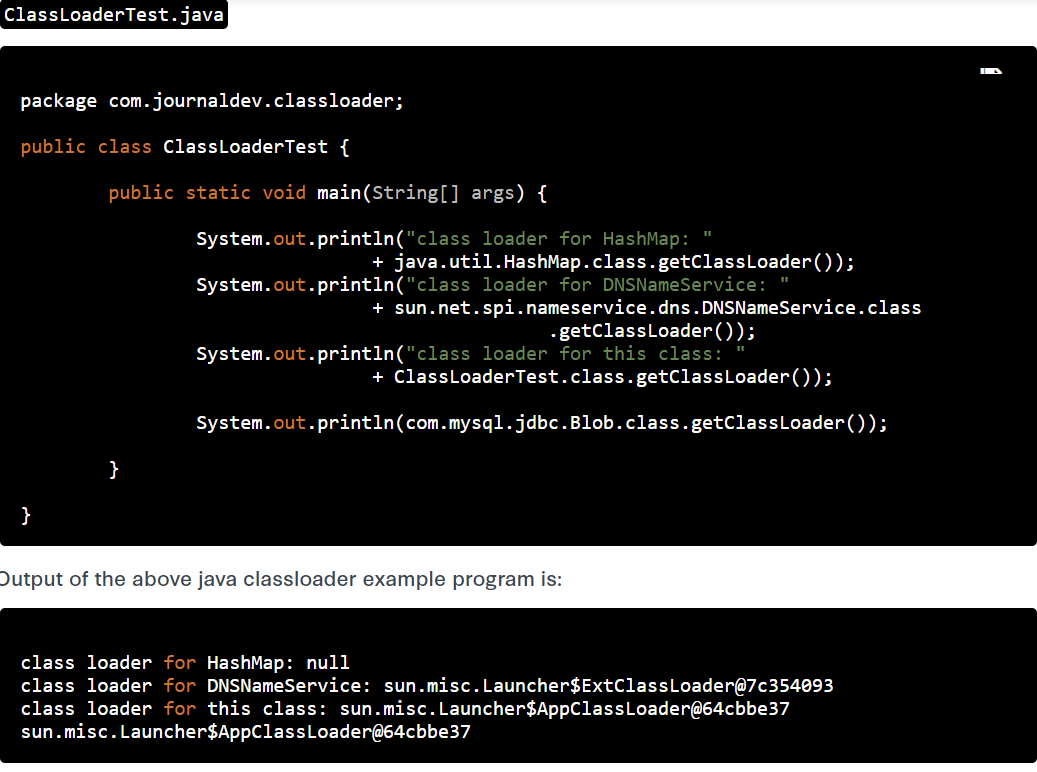
* *Verification* : It ensures the correctness of *.class* file i.e. it check whether this file is properly formatted and generated by valid compiler or not. If verification fails, we get run-time exception *java.lang.VerifyError*.
* *Preparation* : JVM allocates memory for class variables and initializing the memory to default values.
* *Resolution* : It is the process of replacing symbolic references from the type with direct references. It is done by searching into method area to locate the referenced entity.

**Initialization :** In this phase, all static variables are assigned with their values defined in the code and static block(if any). This is executed from top to bottom in a class and from parent to child in class hierarchy.  
In general, there are three class loaders :

* ***Bootstrap class loader*** : Every JVM implementation must have a bootstrap class loader, capable of loading trusted classes. It loads core java API classes present in *JAVA\_HOME/jre/lib* directory. This path is popularly known as bootstrap path. It is implemented in native languages like C, C++.
* ***Extension class loader*** : It is child of bootstrap class loader. It loads the classes present in the extensions directories *JAVA\_HOME/jre/lib/ext*(Extension path) or any other directory specified by the java.ext.dirs system property. It is implemented in java by the *sun.misc.Launcher$ExtClassLoader* class.
* ***System/Application class loader***: It is child of extension class loader. It is responsible to load classes from application class path. It internally uses Environment Variable which mapped to java.class.path. It is also implemented in Java by the *sun.misc.Launcher$AppClassLoader* class.

**Note :**JVM follow Delegation-Hierarchy principle to load classes. System class loader delegate load request to extension class loader and extension class loader delegate request to boot-strap class loader. If class found in boot-strap path, class is loaded otherwise request again transfers to extension class loader and then to system class loader. At last if system class loader fails to load class, then we get run-time exception *java.lang.ClassNotFoundException*.

Java ClassLoader is hierarchical and whenever a request is raised to load a class, it delegates it to its parent and in this way uniqueness is maintained in the runtime environment. If the parent class loader doesn’t find the class then the class loader itself tries to load the class.



As you can see that [java.util.HashMap](https://www.journaldev.com/11560/java-hashmap" \t "_blank) ClassLoader is coming as null that reflects Bootstrap ClassLoader whereas DNSNameService ClassLoader is ExtClassLoader. Since the class itself is in CLASSPATH, System ClassLoader loads it.

When we are trying to load [HashMap](https://www.journaldev.com/11560/java-hashmap), our System ClassLoader delegates it to the Extension ClassLoader, which in turns delegates it to Bootstrap ClassLoader that found the class and load it in JVM.

The same process is followed for DNSNameService class but Bootstrap ClassLoader is not able to locate it since it’s in $JAVA\_HOME/lib/ext/dnsns.jar and hence gets loaded by Extensions Classloader

**JVM Memory**  
**Method area: In** method area, all class level information like class name, immediate parent class name, methods and variables information etc. are stored, including static variables. There is only one method area per JVM, and it is a shared resource.

**Heap area: Information** of all objects is stored in heap area. There is also one Heap Area per JVM. It is also a shared resource.

**Stack area: For** every thread, JVM create one run-time stack which is stored here. Every block of this stack is called activation record/stack frame which store methods calls. All local variables of that method are stored in their corresponding frame. After a thread terminate, it’s run-time stack will be destroyed by JVM. It is not a shared resource.

**PC Registers: Store** address of current execution instruction of a thread. Obviously each thread has separate PC Registers.

**Native method stacks :**For every thread, separate native stack is created. It stores native method information.

**Execution Engine**  
Execution engine execute the *.class* (bytecode). It reads the byte-code line by line, use data and information present in various memory area and execute instructions. It can be classified in three parts:-

* ***Interpreter*:** It interprets the bytecode line by line and then executes. The disadvantage here is that when one method is called multiple times, every time interpretation is required.
* ***Just-In-Time Compiler (JIT****)* : It is used to increase efficiency of interpreter. It compiles the entire bytecode and changes it to native code so whenever interpreter see repeated method calls, JIT provide direct native code for that part so re-interpretation is not required, thus efficiency is improved.
* ***Garbage Collector***: It destroy un-referenced objects.

**Java Native Interface (JNI):**  
It is an interface which interacts with the Native Method Libraries and provides the native libraries(C, C++) required for the execution. It enables JVM to call C/C++ libraries and to be called by C/C++ libraries which may be specific to hardware.

**Native Method Libraries:**  
It is a collection of the Native Libraries(C, C++) which are required by the Execution Engine.

1. **JVM Shutdown Hook in Java?**

Ans- Shutdown Hooks are a special construct that allows developers to plug in a piece of code to be executed when the JVM is shutting down. This comes in handy in cases where we need to do special clean up operations in case the VM is shutting down.

***public class ShutDownHook***

***{***

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

***{***

***Runtime.getRuntime().addShutdownHook(new Thread()  {***

***public void run()       {***

***System.out.println("Shutdown Hook is running !");***

***}***

***});***

***System.out.println("Application Terminating ...");***

***}***

***}***

Output:

Application Terminating ...

Shutdown Hook is running !

**Note: *Shutdown hooks are called when the application terminates normally*** *(when all threads finish, or when System.exit(0) is called). Also, when the JVM is shutting down due to external causes such as a user requesting a termination (Ctrl+C), a SIGTERM being issued by O/S (normal kill command, without -9), or when the operating system is shutting down.*

***It is not guaranteed that shutdown hooks will always run.*** *If the JVM crashes due to some internal error, then it might crash down without having a chance to execute a single instruction. Also, if the O/S gives a SIGKILL (http://en.wikipedia.org/wiki/SIGKILL) signal (kill -9 in Unix/Linux) or TerminateProcess (Windows), then the application is required to terminate immediately without doing even waiting for any cleanup activities.*

***We can have more than one Shutdown Hooks,*** *The JVM can execute shutdown hooks in any arbitrary order. Moreover, the JVM might execute all these hooks concurrently.*

**Once shutdown sequence starts, it can be stopped by Runtime.halt() only.**

1. **If a .java file has more than one class then each class will compile into a separate class files.**
2. **Does JVM create an object of class Main?**

Ans- The answer is “No”. We have studied that the reason for main() static in Java is to make sure that the main() can be called without any instance. To justify the same, we can see that the following program compiles and runs fine.

|  |
| --- |
| Not Main is abstract  **abstract class Main {**  **public static void main(String args[])  {**  **System.out.println("Hello");**  **} }** |

Output:

Hello

Since we can’t create object of [abstract classes in Java](https://www.geeksforgeeks.org/abstract-classes-in-java/), it is guaranteed that object of class with main() is not created by JVM.

1. **What makes it JAVA as platform independent language?**

Ans-  A compiler is a program that translates the source code for another program from a programming language into executable code.  
This executable code may be a sequence of machine instructions that can be executed by the CPU directly, or it may be an intermediate representation that is interpreted by a virtual machine. This intermediate representation in Java is the **Java Byte Code.**

**Step by step Execution of Java Program:**

* Whenever, a program is written in JAVA, the javac compiles it.
* The result of the JAVA compiler is the **.class file or the bytecode** and not the machine native code (unlike C compiler).
* The bytecode generated is a non-executable code and needs an interpreter to execute on a machine. This interpreter is the JVM and thus the Bytecode is executed by the JVM.
* And finally, program runs to give the desired output.
* So here bytecode make JAVA language as platform independent and also portable.

In case of C or C++ (language that are not platform independent), the compiler generates an .exe file which is OS dependent. When we try to run this .exe file on another OS it does not run, since it is OS dependent and hence is not compatible with the other OS.

**Java is platform independent but JVM is platform dependent.**

1. **JDBC?**

Ans-It is an application programming interface (API) for the programming language Java, which defines how a client may access any kind of tabular data, especially relational database.

t acts as a middle layer interface between java applications and database.

The JDBC classes are contained in the Java Package **java.sql** and **javax.sql**.  
JDBC helps you to write Java applications that manage these three programming activities:

1. Connect to a data source, like a database.
2. Send queries and update statements to the database
3. Retrieve and process the results received from the database in answer to your query

JDBC drivers are client-side adapters (installed on the client machine, not on the server) that convert requests from Java programs to a protocol that the DBMS can understand. There are 4 types of JDBC drivers:

1. Type-1 driver or JDBC-ODBC bridge driver
2. Type-2 driver or Native-API driver
3. Type-3 driver or Network Protocol driver
4. Type-4 driver or Thin driver: This driver interacts directly with database. It does not require any native database library, that is why it is also known as Thin Driver.

* Does not require any native library and Middleware server, so no client-side or server-side installation.
* It is fully written in Java language, hence they are portable drivers.

**Which Driver to use When?**

* If you are accessing one type of database, such as Oracle, Sybase, or IBM, the preferred driver type is type-4.
* If your Java application is accessing multiple types of databases at the same time, type 3 is the preferred driver.
* Type 2 drivers are useful in situations, where a type 3 or type 4 driver is not available yet for your database.
* The type 1 driver is not considered a deployment-level driver and is typically used for development and testing purposes only.

1. **Is main method compulsory in Java?**

Ans- Prior to JDK 7, the main method was not mandatory in a java program. You could write your full code under [static block](https://www.geeksforgeeks.org/g-fact-79/) and it ran normally.

* The static block is first executed as soon as the class is loaded before the main();
* It will run static block first and then it will see no main() is there. Therefore, it will give **“exception”**, as exception comes while execution. However, if we don’t want an exception, we can terminate the program by System.exit(0);
* From JDK7 main method is mandatory. The compiler will verify first, whether main() is present or not. If your program doesn’t contain the main method, then you will get an **error** “main method not found in the class”. It will give an error (byte code verification error because in it’s byte code, main is not there) not an exception because the program has not run yet.

1. **In java file name and class name should be the same if the class is declared as public.**
2. **In Java, Using predefined class name as Class or Variable name is allowed but you cannot use a keyword as name of a class, name of a variable nor the name of a folder used for package**.

**// Number is predefined class name in java.lang package**

// Note : java.lang package is included in every java program by default

public class Number

**{**

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

**System.out.println("It works"); }}**

***Output: It works***

**Using String as User Defined Class:**

|  |
| --- |
| **// String is predefined class name in java.lang package**  **// Note : java.lang package is included in every java program by default**  **public class String**  **{**  **public static void main (String[] args)   {**  **System.out.println("I got confused");  } }** |

However, in this case you will get run-time error like this:  [Main thread](https://www.geeksforgeeks.org/main-thread-java/)is looking for main method() with predefined **String class** array argument. But here, it got main method() with user defined String class. Whenever Main thread will see a class name, it tries to search that class scope by scope.

First it will see in your program, then in your package.If not found, then [JVM](https://www.geeksforgeeks.org/jvm-works-jvm-architecture/) follows delegation hierarchy principle to load that class.Hence you will get run-time error. To fix, write this -**public static void main (java.lang.String[] args)**

1. Identifiers are used for identification purpose. In Java, an identifier can be a class name, method name, variable name or a label. It should be alphanumeric character or “$” or “\_” and should not start with the digit, are case-sensitive and reserved words can’t be used as an identifier.
2. values of type boolean are not converted implicitly or explicitly (with casts) to any other type



**By default, fraction value is double in java and use byte and short if memory is a constraint.**

1. **Enum in java?**

Ans-Enumerations serve the purpose of representing a group of named constants like the planets, days of the week, colors, directions, etc.

* Enums are used when we know all possible values at **compile time**.
* It is not necessary that the set of constants in an enum type stay **fixed** for all time.
* In Java, we can also add variables, methods and constructors to it. The main objective of enum is to define our own data types(Enumerated Data Types).
* Enum declaration can be done outside a Class or inside a Class but not inside a Method.

**enum Color**

**{**

**RED, GREEN, BLUE;**

**}**

**public class Test {**

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

**Color c1 = Color.RED;**

**System.out.println(c1);**

**} }**

**Output:RED**

* First line inside enum should be list of constants and then other things like methods, variables and constructor.
* Every enum internally implemented by using Class.

/\* internally above enum Color is converted to

class Color

{

public static final Color RED = new Color();

public static final Color BLUE = new Color();

public static final Color GREEN = new Color();

}\*/

* Every enum constant represents an **object** of type enum.
* Enum type can be passed as an argument to **switch** statement.
* Every enum constant is always implicitly **public static final**. Since it is **static**, we can access it by using enum Name. Since it is **final**, we can’t create child enums.
* We can declare **main() method** inside enum. Hence we can invoke enum directly from the Command Prompt.
* All enums implicitly extend **java.lang.Enum class**. As a class can only extend **one** parent in Java, so an enum cannot extend anything else.
* **toString() method** is overridden in **java.lang.Enum class**,which returns enum constant name but enum can implement many interfaces.
* **Enum methods**- These methods are present inside **java.lang.Enum**.
* **values() method** can be used to return all values present inside enum.
* Order is important in enums.By using **ordinal() method**, each enum constant index can be found, just like array index.
* **valueOf() method** returns the enum constant of the specified string value, if exists

**enum Color**

**{**

**RED, GREEN, BLUE;**

**}**

**public class Test**

**{**

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

**{**

**// Calling values()**

**Color arr[] = Color.values();**

**// enum with loop**

**for (Color col : arr)**

**{**

**// Calling ordinal() to find index**

**// of color.**

**System.out.println(col + " at index "**

**+ col.ordinal());**

**}**

**// Using valueOf(). Returns an object of Color with given constant.**

**// Uncommenting second line causes exception IllegalArgumentException**

**System.out.println(Color.valueOf("RED"));**

**// System.out.println(Color.valueOf("WHITE"));**

**}**

**}**

Output: RED at index 0

GREEN at index 1

BLUE at index 2

RED

**enum and constructor :**

* enum can contain constructor and it is executed separately for each enum constant at the time of enum class loading.
* We can’t create enum objects explicitly and hence we can’t invoke enum constructor directly.

**enum and methods :**

* enum can contain **concrete** methods only i.e. no any **abstract** method.

|  |
| --- |
| **// Java program to demonstrate that enums can have constructor**  **// and concrete methods.**    **// An enum (Note enum keyword inplace of class keyword)**  **enum Color**  **{**  **RED, GREEN, BLUE;**    **// enum constructor called separately for each constant**  **private Color()**  **{**  **System.out.println("Constructor called for : " + this.toString());**  **}**    **// Only concrete (not abstract) methods allowed**  **public void colorInfo()**  **{**  **System.out.println("Universal Color");**  **}**  **}**    **public class Test**  **{**  **public static void main(String[] args)**  **{**  **Color c1 = Color.RED;**  **System.out.println(c1);**  **c1.colorInfo();**  **}**  **}** |

Output:

Constructor called for : RED

Constructor called for : GREEN

Constructor called for : BLUE

RED

Universal Color

1. **Enum with customized value in Java?**

Ans-By default enums have their own string values, we can also assign some custom values to enums.

|  |
| --- |
| **enum TrafficSignal**  **{**  **// This will call enum constructor with one**  **// String argument**  **RED("STOP"), GREEN("GO"), ORANGE("SLOW DOWN");**    **// declaring private variable for getting values**  **private String action;**    **// getter method**  **public String getAction()**  **{**  **return this.action;**  **}**    **// enum constructor - cannot be public or protected**  **private TrafficSignal(String action)**  **{**  **this.action = action;**  **}**  **}**  **public class EnumConstructorExample**  **{**  **public static void main(String args[])**  **{**  **// let's print name of each enum and there action**  **TrafficSignal[] signals = TrafficSignal.values();**    **for (TrafficSignal signal : signals)**  **{**  **// use getter method to get the value**  **System.out.println("name : " + signal.name() +  " action: " + signal.getAction() );**  **}**  **}**  **}** |

Output:

name : RED action: STOP

name : GREEN action: GO

name : ORANGE action: SLOW DOWN

1. We have to create parameterized constructor for this enum class. Why? Because as we know that enum class’s object can’t be create explicitly so for initializing we use parameterized constructor. And the constructor cannot be the public or protected it must have private or default modifiers. Why? if we create public or protected, it will allow initializing more than one objects. This is totally against enum concept.
2. We have to create one getter method to get the value of enums.

Here enum have three members- RED, GREEN and YELLOW which have their own different custom values- STOP,GO and SLOW DOWN.

1. **Scope of a variable is the part of the program where the variable is accessible.**

Member variable has class level scope, local variables has method level scope and loop variables has block scope means outside the loop they are not accessible and will get error at compile time

When a method has the same local variable as a member, this keyword can be used to reference the current class variable.

1. **Final variables?**

Ans- A final variable in Java can be assigned a value only once, we can assign a value either in declaration or later. They are used to create immutable objects (objects whose members can’t be changed once initialized).

A **blank final** variable in Java is a [final](https://www.geeksforgeeks.org/g-fact-48/) variable that is not initialized during declaration. Values must be assigned in constructor, we do this because but if we initialize here, then all objects get the same value. So, we use blank final.

**class Test**

**{**

**final public int i;**

**Test(int val)**

**{  this.i = val;  }**

**Test()**

**{        // Calling Test(int val)**

**this(10);   }**

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

**{**

**Test t1 = new Test();**

**System.out.println(t1.i);**

**Test t2 = new Test(20);**

**System.out.println(t2.i);**

**}**

**}**

**Output:**

**10**

**20**

**If we have more than one constructors or overloaded constructor in class, then blank final variable must be initialized in all of them. However constructor chaining can be used to initialize the blank final variable.**

1. **For-each loop?**

**Ans-** For-each is another array traversing technique like other loops. Instead of declaring and initializing a loop counter variable, you declare a variable that is the same type as the base type of the array, followed by a colon, which is then followed by the array name.

for (type var : array) {

statements using var;

}

**Limitations of for-each loop**

1. For-each loops are**not appropriate when you want to modify the array**

for (int num : marks)

{

// only changes num, not the array element

num = num\*2;

}

1. For-each loops **do not keep track of index**. So we can not obtain array index using For-Each loop

for (int num : numbers)

{

if (num == target)

{

return ???; // do not know the index of num

}

}

1. For-each **only iterates forward over the array in single steps**

// cannot be converted to a for-each loop

for (int i=numbers.length-1; i>0; i--)

{

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

}

1. For-each **cannot process two decision making statements** at once

// cannot be easily converted to a for-each loop

for (int i=0; i<numbers.length; i++)

{

if (numbers[i] == arr[i])

{ ...

}

}

1. **Switch case?**

**Ans-** Expression can be of type byte, short, int, char or an enumeration. Beginning with JDK7, *expression* can also be of type String.

* Duplicate case values are not allowed.
* The value for a case must be a constant or a literal. Variables are not allowed.
* The default statement is optional. Can be placed anywhere inside the switch block.
* The break statement is used inside the switch to terminate a statement sequence.
* The break statement is optional. If omitted, execution will continue into the next case.
* We can use a switch as part of the statement sequence of an outer switch. This is called a nested switch

**Break:** In Java, break is majorly used for:

* Terminate a sequence in a switch statement (discussed above).
* To exit a loop.

**Continue:**  That is, you might want to continue running the loop but stop processing the remainder of the code in its body for this particular iteration.

1. **String in switch statements?**

Ans- **Important Points:**

* **Expensive operation:**Switching on strings can be more expensive in term of execution than switching on primitive data types. Therefore, it is best to switch on strings only in cases in which the controlling data is already in string form.
* **String should not be NULL:** Ensure that the expression in any switch statement is not null while working with strings to prevent a NullPointerException from being thrown at run-time.
* **Case Sensitive Comparison:**The switch statement compares the String object in its expression with the expressions associated with each case label as if it were using the String.equals method; consequently, the comparison of String objects in switch statements is case sensitive.
* **Better than if-else:** The Java compiler generates generally more efficient bytecode from switch statements that use String objects than from chained if-then-else statements.

1. Unlike C++, we don’t need [forward declarations](http://en.wikipedia.org/wiki/Forward_declaration) in Java. Identifiers (class and method names) are recognized automatically from source files.The Java program compiles and runs fine. Note that Test1 and fun() are not declared before their use.
2. **The widening primitive conversion happens only when a operator like ‘+’ is present between the operands**.

* The result of adding Java chars, shorts or bytes is an int:
* If either operand is of type double, the other is converted to double.
* Otherwise, if either operand is of type float, the other is converted to float.
* Otherwise, if either operand is of type long, the other is converted to long.
* Otherwise, **both operands are converted to type int**

System.out.print("Y" + "O");

      System.out.print('L' + 'O');

*When we use double quotes, the text is treated as a string and “YO” is printed, but when we use single quotes, the characters ‘L’ and ‘O’ are converted to int. This is called widening primitive conversion. After conversion to integer, the numbers are added ( ‘L’ is 76 and ‘O’ is 79) and 155 is printed.*

1. **Type conversion in Java?**

Ans- Widening conversion takes place when two data types are automatically converted else they need to be casted or converted explicitly. F. This happens when:

* The two data types are compatible.
* When we assign value of a smaller data type to a bigger data type

**Widening or Automatic Type Conversion:** *Byte -> Short -> Int -> Long -> Float ->Double*

|  |
| --- |
| **int i = 100;**    **//automatic type conversion**  **long l = i;**    **//automatic type conversion**  **float f = l;**  **System.out.println("Int value "+i);**  **System.out.println("Long value "+l);**  **System.out.println("Float value "+f);** |

**Output:**

Int value 100

Long value 100

Float value 100.0

**Narrowing or Explicit Conversion**

If we want to assign a value of larger data type to a smaller data type we perform explicit type casting or narrowing.

*Double -> Float -> Long ->Int ->Short ->byte*

**Ex-1 char and number are not compatible with each other**

**char ch = 'c';**

**int num = 88;**

**ch = num;**

**Output:** incompatible types: possible lossy conversion from int to char

ch = num;

**Ex-2 double d = 100.04;**

**//explicit type casting**

**long l = (long)d;**

**//explicit type casting**

**int i = (int)l;**

**System.out.println("Double value "+d);**

**//fractional part lost**

**System.out.println("Long value "+l);**

**//fractional part lost**

**System.out.println("Int value "+i);**

Output:

Double value 100.04

Long value 100

Int value 100

Ex-3 While assigning value to byte type the fractional part is lost and is reduced to modulo 256(range of byte).

**byte b;**

**int i = 257;**

**double d = 323.142;**

**System.out.println("Conversion of int to byte.");**

**//i%256**

**b = (byte) i;**

**System.out.println("i = " + i + " b = " + b);**

**System.out.println("\nConversion of double to byte.");**

**//d%256**

**b = (byte) d;**

**System.out.println("d = " + d + " b= " + b);**

Output:

Conversion of int to byte.

i = 257 b = 1

Conversion of double to byte.

d = 323.142 b = 67

1. **Explicit type casting in Expressions?**

Ans- While evaluating expressions, the result is automatically updated to larger data type of the operand. But if we store that result in any smaller data type it generates compile time error, due to which we need to type cast the result.

***byte b = 50;***

***//type casting int to byte***

***b = (byte)(b \* 2);***

1. **Type promotion in Expressions**

Ans- While evaluating expressions, the intermediate value may exceed the range of operands and hence the expression value will be promoted. Some conditions for type promotion

* Java automatically promotes each byte, short, or char operand to int when evaluating an expression.
* if one operand is a long, float or double the whole expression is promoted to long, float or double respectively.

1. **Null in Java?**

Ans- a) **== and !=**The comparison and not equal to operators are allowed with null in Java.

b) We cannot call a non-static method on a reference variable with null value, it will throw NullPointerException, but we can call static method with reference variables with null values. Since static methods are bonded using static binding, they won’t throw Null pointer Exception.

c) the java instanceof operator is used to test whether the object is an instance of the specified type (class or subclass or interface). At run time, the result of the instanceof operator is true if the value of the Expression is not null.

d) null is not Object or neither a type. It’s just a special value, which can be assigned to any reference type and you can type cast null to any type

e) **null is Case sensitive:** null is literal in Java and because keywords are **case-sensitive** in java, we can’t write NULL or 0 as in C language. It will throw compile-time error : can't find symbol 'NULL' *Object obj = NULL;*

1. A new feature was introduced by JDK 7 which allows to **write numeric literals using the underscore character**. Numeric literals are broken to enhance the readability.

***int inum = 1\_00\_00\_000;***

***System.out.println("inum:" + inum);***

***long lnum = 1\_00\_00\_000;***

***System.out.println("lnum:" + lnum);***

***float fnum = 2.10\_001F;***

***System.out.println("fnum:" + fnum);***

***double dnum = 2.10\_12\_001;***

***System.out.println("dnum:" + dnum);***

**Output:**

inum: 10000000

lnum: 10000000

fnum: 2.10001

dnum: 2.1012001

1. **Binary search in JAVA?**

Ans- It can be done in 2 ways:

1. [Arrays.binarysearch()](https://www.geeksforgeeks.org/arrays-binarysearch-java-examples-set-1/)works for arrays which can be of primitive data type also

*int arr[] = { 10, 20, 15, 22, 35 };*

***Arrays.sort(arr);***

*int key = 22;*

***int res = Arrays.binarySearch(arr, key);***

*if (res >= 0)*

*System.out.println(key + " found at index = "*

*+ res);*

*else*

*System.out.println(key + " Not found");*

*key = 40;*

*res = Arrays.binarySearch(arr, key);*

*if (res >= 0)*

*System.out.println(key + " found at index = "*

*+ res);*

*else*

*System.out.println(key + " Not found")*

**Output:**

22 found at index = 3

* 1. ot found

1. [Collections.binarysearch()](https://www.geeksforgeeks.org/collections-binarysearch-java-examples/) works for objects Collections like [ArrayList](https://www.geeksforgeeks.org/arraylist-in-java/) and [LinkedList](http://geeksquiz.com/linked-list-in-java/).

*List<Integer> al = new ArrayList<Integer>();*

*al.add(1);*

*al.add(2);*

*al.add(3);*

*al.add(10);*

*al.add(20);*

*// 10 is present at index 3.*

*int key = 10;*

***int res = Collections.binarySearch(al, key);***

*if (res >= 0)*

*System.out.println(key + " found at index = "*

*+ res);*

*else*

*System.out.println(key + " Not found");*

*key = 15;*

*res = Collections.binarySearch(al, key);*

*if (res >= 0)*

*System.out.println(key + " found at index = "*

*+ res);*

*else*

*System.out.println(key + " Not found");*

*Note:*

*Q-* **What if input is not sorted?**  
A- If input list is not sorted, the results are undefined.

Q-**What if there are duplicates?**  
A- If there are duplicates, there is no guarantee which one will be found.

Q- **What is significant value of negative value returned by both functions?**  
A- The function returns an index of the search key, if it is contained in the array; otherwise, (-(insertion point) – 1).

1. Sorting in JAVA?

Ans- There are two in-built methods to sort in Java.

1. [Arrays.Sort()](https://www.geeksforgeeks.org/arrays-sort-in-java-with-examples/) works for arrays which can be of primitive data type also
2. [Collections.sort()](https://www.geeksforgeeks.org/collections-sort-java-examples/) works for objects Collections like ArrayList and LinkedList.

*Note:*

*Q-* **Which sorting algorithm does Java use in sort()?**  
A- Previously, Java’s Arrays.sort method used Quicksort for arrays of primitives and Merge sort for arrays of objects. In the latest versions of Java, Arrays.sort method and Collection.sort() uses Timsort.

**Q-Which order of sorting is done by default?**  
A- It by default sorts in ascending order.

**Q-How to sort array or list in descending order?**  
A-It can be done with the help of Collections.reverseOrder().

**Arrays.sort(arr, Collections.reverseOrder());**

**Collections.sort(al, Collections.reverseOrder());**

Q- **How to sort only a subarray**

1. Sort subarray from index 1 to 4, i.e.,

**// only sort subarray {7, 6, 45, 21} and keep other elements as it is.**

**Arrays.sort(arr, 1, 5);**

1. **Class and objects?**

**Ans-** A class is a user defined blueprint or prototype from which objects are created.  It represents the set of properties or methods that are common to all objects of one type.

An object represents the real-life entities which consists of –

1. **State**: It is represented by attributes of an object. It also reflects the properties of an object.
2. **Behavior**: It is represented by methods of an object. It also reflects the response of an object with other objects.
3. **Identity**: It gives a unique name to an object and enables one object to interact with other objects.

Example of an object: dog

*Identity: Name of the dog*

*State: Breed, age, color*

*Behaviour: Bark, Sleep and eat*

In Java, when we only declare a variable of a class type, only a reference is created (memory is not allocated for the object). To allocate memory to an object, we must use new(). So the object is always allocated memory on heap.

1. **Ways to create object in Java?**

**Ans-**

1. **Using new keyword**

// creating object of class Test

Test t = new Test();

1. **Using Class.forName(String className) method:** If we know the name of the class & if it has a public default constructor we can create an object. Class.forName actually loads the Class in Java but doesn’t create any Object. To Create an Object of the Class you have to use the new Instance Method of the Class.

// creating object of public class Test

// consider class Test present in com.p1 package

Test obj = (Test)Class.forName("com.p1.Test").newInstance();

1. **Using clone() method**: clone() method is present in Object class. Whenever clone() is called on any object, the JVM actually creates a new object and copies all content of the previous object into it. Creating an object using the clone method does not invoke any constructor.  
   To use clone() method on an object we need to implement **Cloneable** and define the clone() method in it.

public class Test implements Cloneable

{

@Override

protected Object clone() throws CloneNotSupportedException

{

return super.clone();

}

}

// creating object of class Test

Test t1 = new Test();

// creating clone of above object

Test t2 = (Test)t1.clone();

1. **Deserialization**: De-serialization is technique of reading an object from the saved state in a file.

Whenever we serialize and then deserialize an object, JVM creates a separate object. In **deserialization**, JVM doesn’t use any constructor to create the object.  
To deserialize an object we need to implement the Serializable interface in the class.

FileInputStream file = new FileInputStream(filename);

ObjectInputStream in = new ObjectInputStream(file);

Object obj = in.readObject();

1. **Swapping objects in JAVA?**

Ans- What if we don’t know members of Car or the member list is too big. This is a very common situation as a class that uses some other class may not access members of other class.

Using wrapper class it is possible to do that even if the user class doesn’t have access to members of the class whose objects are to be swapped.

If we create a wrapper class that contains references of Car, we can swap cars by swapping references of wrapper class.

*// A Wrapper over class that is used for swapping*

*class CarWrapper*

*{*

*Car c;*

*// Constructor*

*CarWrapper(Car c)   {this.c = c;}*

*}*

*// A Class that use Car and swaps objects of Car*

*// using CarWrapper*

*class Main*

*{*

*// This method swaps car objects in wrappers*

*// cw1 and cw2*

*public static void swap(CarWrapper cw1,*

*CarWrapper cw2)*

*{*

*Car temp = cw1.c;*

*cw1.c = cw2.c;*

*cw2.c = temp;*

*}*

*}*

public static void main(String[] args)

     {

        Car c1 = new Car(101, 1);

        Car c2 = new Car(202, 2);

        CarWrapper cw1 = new CarWrapper(c1);

        CarWrapper cw2 = new CarWrapper(c2);

        swap(cw1, cw2);

        cw1.c.print();

        cw2.c.print();

     }

1. **Inheritance in Java?**

Ans- It is the mechanism in java by which one class is allow to inherit the features (fields and methods) of another class. Inheritance supports the concept of “reusability”, i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

**Important facts about inheritance in Java**

* **Default superclass**: Except [Object](https://www.geeksforgeeks.org/object-class-in-java/) class, which has no superclass, every class has one and only one direct superclass (single inheritance). In the absence of any other explicit superclass, every class is implicitly a subclass of [Object](https://www.geeksforgeeks.org/object-class-in-java/) class.
* **Superclass can only be one:** A superclass can have any number of subclasses. But a subclass can have only **one** superclass. This is because Java does not support [multiple inheritance](https://www.geeksforgeeks.org/java-and-multiple-inheritance/) with classes. Although with interfaces, multiple inheritance is supported by java.
* **Inheriting Constructors:**A subclass inherits all the members (fields, methods, and nested classes) from its superclass. Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.
* **Private member inheritance:** A subclass does not inherit the private members of its parent class. However, if the superclass has public or protected methods (like getters and setters) for accessing its private fields, these can also be used by the subclass.

1. **Encapsulation in Java?**

Ans- Encapsulation is defined as the wrapping up of data under a single unit. It is the mechanism that binds together code and the data it manipulates.

* Technically in encapsulation, the variables or data of a class is hidden from any other class and can be accessed only through any member function of own class in which they are declared. As in encapsulation, the data in a class is hidden from other classes, so it is also known as **data-hiding**.
* Encapsulation can be achieved by: Declaring all the variables in the class as private and writing public methods in the class to set and get the values of variables.

**Advantages of Encapsulation**:

* **Data Hiding:** The user will have no idea about the inner implementation of the class. It will not be visible to the user that how the class is storing values in the variables. He only knows that we are passing the values to a setter method and variables are getting initialized with that value.
* **Increased Flexibility:** We can make the variables of the class as read-only or write-only depending on our requirement. If we wish to make the variables as read-only then we have to omit the setter methods like setName(), setAge() etc. from the above program or if we wish to make the variables as write-only then we have to omit the get methods like getName(), getAge() etc. from the above program

1. **Abstraction in Java?**

Ans- Data Abstraction is the property by virtue of which only the essential details are displayed to the user. The trivial or the non-essentials units are not displayed to the user. Ex: A car is viewed as a car rather than its individual components.

Consider a real-life example of a man driving a car. The man only knows that pressing the accelerators will increase the speed of car or applying brakes will stop the car but he does not know about how on pressing the accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of accelerator, brakes etc in the car. This is what abstraction is.

**In java, abstraction is achieved by**[**interfaces**](https://www.geeksforgeeks.org/interfaces-in-java/)**and**[**abstract classes**](https://www.geeksforgeeks.org/abstract-classes-in-java/)**. We can achieve 100% abstraction using interfaces.**

**Abstract classes and Abstract methods :**

* An abstract class is a class that is declared with [abstract keyword.](https://www.geeksforgeeks.org/abstract-keyword-in-java/)
* An abstract method is a method that is declared without an implementation.
* An abstract class may or may not have all abstract methods. Some of them can be concrete methods
* A method defined abstract must always be redefined in the subclass,thus making [overriding](http://contribute.geeksforgeeks.org/overriding-in-java/) compulsory OR either make subclass itself abstract.
* Any class that contains one or more abstract methods must also be declared with abstract keyword.
* There can be no object of an abstract class. That is, an abstract class can not be directly instantiated with the [new operator](https://www.geeksforgeeks.org/new-operator-java/).
* An abstract class can have parametrized constructors and default constructor is always present in an abstract class.

**When to use abstract classes and abstract methods with an example**

There are situations in which we will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method. That is, sometimes we will want to create a superclass that only defines a generalization form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details.

Consider a classic “shape” example, perhaps used in a computer-aided design system or game simulation. The base type is “shape” and each shape has a color, size and so on. From this, specific types of shapes are derived(inherited)-circle, square, triangle and so on – each of which may have additional characteristics and behaviors.

|  |
| --- |
| / Java program to illustrate the concept of Abstraction  **abstract class Shape**  {      String color;        // these are abstract methods      abstract double area();      public abstract String toString();        // abstract class can have constructor      public Shape(String color) {          System.out.println("Shape constructor called");          this.color = color;      }        // this is a concrete method      public String getColor() {          return color;      }  }  **class Circle extends Shape**  {      double radius;        public Circle(String color,double radius) {            // calling Shape constructor          super(color);          System.out.println("Circle constructor called");          this.radius = radius;      }        @Override      double area() {          return Math.PI \* Math.pow(radius, 2);      }        @Override      public String toString() {          return "Circle color is " + super.color +                         "and area is : " + area();      }      }  **class Rectangle extends Shape{**        double length;      double width;        public Rectangle(String color,double length,double width) {          // calling Shape constructor          super(color);          System.out.println("Rectangle constructor called");          this.length = length;          this.width = width;      }        @Override      double area() {          return length\*width;      }        @Override      public String toString() {          return "Rectangle color is " + super.color +                             "and area is : " + area();      }    }  **public class Test**  {      public static void main(String[] args)      {          Shape s1 = new Circle("Red", 2.2);          Shape s2 = new Rectangle("Yellow", 2, 4);            System.out.println(s1.toString());          System.out.println(s2.toString());      }  } |

Output:

Shape constructor called

Circle constructor called

Shape constructor called

Rectangle constructor called

Circle color is Redand area is : 15.205308443374602

Rectangle color is Yellowand area is : 8.0

**Encapsulation vs Data Abstraction**

1. [Encapsulation](http://contribute.geeksforgeeks.org/encapsulation-in-java/) is data hiding (information hiding) while Abstraction is detail hiding (implementation hiding).
2. While encapsulation groups together data and methods that act upon the data, data abstraction deals with exposing the interface to the user and hiding the details of implementation.

**Advantages of Abstraction**

1. It reduces the complexity of viewing the things.
2. Avoids code duplication and increases reusability.
3. Helps to increase security of an application or program as only important details are provided to the user.
4. **Runtime polymorphism in Java?**

Ans- Method overriding is one of the ways in which Java supports Runtime Polymorphism. Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.

* At run-time, it depends on the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed.
* When an overridden method is called through a superclass reference, Java determines which version(superclass/subclasses) of that method is to be executed based upon the type of the object being referred to at the time the call occurs.
* In Java, we can override methods only, not the variables (data members), so **runtime polymorphism cannot be achieved by data members.**

|  |
| --- |
| // Java program to illustrate the fact that runtime polymorphism cannot be achieved by data members  class A  {  int x = 10;  void m1()      {          System.out.println("Inside A's m1 method");      }  }    // class B  class B extends A  {  int x = 20;  // overriding m1()      void m1()      {          System.out.println("Inside B's m1 method");      }  }    // Driver class  public class Test  {      public static void main(String args[])      {          A a = new B(); // object of type B            // Data member of class A will be accessed          System.out.println(a.x);  System.out.println(a.m1());      }  } |
|  |

Output:

10

Inside B's m1 method

**Explanation:** In above program, both the class A(super class) and B(sub class) have a common variable ‘x’. Now we make object of class B, referred by ‘a’ which is of type of class A. Since variables are not overridden, so the statement “a.x” will **always** refer to data member of super class.

**Advantages of Dynamic Method Dispatch**

1. Dynamic method dispatch allow Java to support [overriding of methods](https://www.geeksforgeeks.org/overriding-in-java/) which is central for run-time polymorphism.
2. It allows a class to specify methods that will be common to all of its derivatives, while allowing subclasses to define the specific implementation of some or all of those methods.
3. It also allow subclasses to add its specific methods subclasses to define the specific implementation of some.

[**Static vs Dynamic binding**](https://www.geeksforgeeks.org/static-vs-dynamic-binding-in-java/)

* Static binding is done during compile-time while dynamic binding is done during run-time.
* private, final and static methods and variables uses static binding and bonded by compiler while overridden methods are bonded during runtime based upon type of runtime object

1. Association, aggregation and composition in Java?

Ans- Association is relation between two separate classes which establishes through their Objects. Association can be one-to-one, one-to-many, many-to-one, many-to-many.

**Composition** and **Aggregation** are the two forms of association.

**Aggregation**

It is a special form of Association where:

* It represents **Has-A** relationship.
* It is a **unidirectional association** i.e. a one way relationship. For example, department can have students but vice versa is not possible and thus unidirectional in nature.
* In Aggregation,**both the entries can survive individually** which means ending one entity will not effect the other entity

*/\* Department class contains list of student Objects. It is associated with student class through its Object(s). \*/*

*class Department*

*{*

*String name;*

*private List<Student> students;*

*Department(String name, List<Student> students)*

*{*

*this.name = name;*

*this.students = students;*

*}*

*public List<Student> getStudents()*

*{*

*return students;*

*}*

*}*

**Composition**

Composition is a restricted form of Aggregation in which two entities are highly dependent on each other.

* It represents **part-of** relationship.
* In composition, both the entities are dependent on each other.
* When there is a composition between two entities, the composed object **cannot exist** without the other entity.

*// class book*

*class Book*

*{*

*public String title;*

*public String author;*

*Book(String title, String author)*

*{*

*this.title = title;*

*this.author = author;*

*}*

*}*

*// Libary class contains list of books.*

*class Library*

*{*

*// reference to refer to list of books.*

*private final List<Book> books;*

*Library (List<Book> books)*

*{*

*this.books = books;*

*}*

*public List<Book> getTotalBooksInLibrary(){*

*return books;*

*}*

*}*

A library can have no. of **books** on same or different subjects. So, If Library gets destroyed then All books within that particular library will be destroyed. i.e. book can not exist without library. That’s why it is composition.

**Aggregation vs Composition**

1. **Dependency:** Aggregation implies a relationship where the child **can exist independently** of the parent. For example, Bank and Employee, delete the Bank and the Employee still exist. whereas Composition implies a relationship where the child **cannot exist independent** of the parent. Example: Human and heart, heart don’t exist separate to a Human
2. **Type of Relationship:** Aggregation relation is **“has-a”** and composition is **“part-of”** relation.
3. **Type of association:**Composition is a **strong** Association whereas Aggregation is a **weak** Association.
4. **Access modifiers**- *private, public, protected and package-private*

**Non-Access modifiers**- *static, final, abstract, synchronized, transient, volatile and native*.

1. **‘this’ keyword in Java?**

Ans- ‘this’ is a reference variable that refers to the current object.

1. **Using ‘this’ keyword to refer current class instance variables**

// Parameterized constructor

    Test(int a, int b)

    {

        this.a = a;

        this.b = b;

    }

1. **Using this() to invoke current class constructor**

//Default constructor

    Test()

    {

        this(10, 20);

        System.out.println("Inside  default constructor \n");

    }

    //Parameterized constructor

    Test(int a, int b)

    {

        this.a = a;

        this.b = b;

        System.out.println("Inside parameterized constructor");

    }

1. **Using ‘this’ keyword to return the current class instance**

//Method that returns current class instance

    Test get()

    {

        return this;

}

1. **Using ‘this’ keyword as method parameter**

// Method that send current class instance as a parameter to another method

    void get()

    {

        display(this);

    }

1. **Using ‘this’ keyword to invoke current class method**

void display()

{

     // calling fuction show()

     this.show();

     System.out.println("Inside display function");

}

1. **Method overloading in JAVA?**

**Ans-** Overloading allows different methods to have the same name, but different signatures where the signature can differ by the number of input parameters or type of input parameters or orders of parameters or combination of them. Overloading is related to compile time (or static) polymorphism.

*// Overloaded sum(). This sum takes two int parameters*

*public int sum(int x, int y)*

*{*

*return (x + y);*

*}*

*// Overloaded sum(). This sum takes three int parameters*

*public int sum(int x, int y, int z)*

*{*

*return (x + y + z);*

*}*

*// Overloaded sum(). This sum takes two double parameters*

*public double sum(double x, double y)*

*{*

*return (x + y);*

*}*

1. **What if the exact prototype does not match with arguments.**  
   Ans. Priority wise, compiler take these steps:
2. Type Conversion but to higher type (in terms of range) in same family.
3. Type conversion to next higher family (suppose if there is no long data type available for an int data type, then it will search for float data type).

But if not found by compiler than there will be an error. Like for float datatype if double is not available then there will be an error.

1. **What is the advantage?**  
   We don’t have to create and remember different names for functions doing the same thing. For example, in our code, if overloading was not supported by Java, we would have to create method names like sum1, sum2, … or sum2Int, sum3Int, … etc.
2. We **cannot** overload by return type. It will throw a compile time error because the compiler will give error as the return value alone is not sufficient for the compiler to figure out which function it has to call. Only if both methods have different parameter types (so, they have different signature), then Method overloading is possible.
3. **Can we overload static methods?**

Ans- The answer is ‘**Yes**’. We can have two or more static methods with same name, but differences in input parameters. Also we can [overload main() in Java](https://www.geeksforgeeks.org/gfact-48-overloading-main-in-java/) like other static methods.

*// Normal main()*

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

*{*

*System.out.println("Hi Geek (from main)");*

*Test.main("Geek");*

*}*

*// Overloaded main methods*

*public static void main(String arg1)*

*{*

*System.out.println("Hi, " + arg1);*

*Test.main("Dear Geek", "My Geek");*

*}*

*public static void main(String arg1, String arg2)*

*{*

*System.out.println("Hi, " + arg1 + ", " + arg2);*

*}*

Output :

Hi Geek (from main)

Hi, Geek

Hi, Dear Geek, My Geek

1. **Can we overload methods that differ only by static keyword?**

Ans- We **cannot** overload two methods in Java if they differ only by static keyword (number of parameters and types of parameters is same).

1. **Does Java support Operator Overloading?**

Ans- Unlike C++, Java doesn’t allow user-defined overloaded operators. Internally Java overloads operators, for example, + is overloaded for concatenation.

1. **What is the difference between Overloading and**[**Overriding**](https://www.geeksforgeeks.org/overriding-in-java/)**?**

* Overloading is about same function have different signatures. Overriding is about same function, same signature but different classes connected through inheritance.
* Overloading is an example of compiler time polymorphism and overriding is an example of run time polymorphism.

1. **Method overloading with Autoboxing and unboxing.**

Ans- Conversion of primitive type to its corresponding wrapper Object is called Autoboxing and Wrapper Object to its corresponding primitive type is known as Unboxing.

*public void method(Integer i)*

*{*

*System.out.println("Reference type Integer formal argument :" + i);*

*}*

*public void method(long i)*

*{*

*System.out.println("Primitive type long formal argument :" + i);*

*}*

*Now invoking them via-*

*c.method(new Integer(15));*

*c.method(new Long(100));*

**Output:**

**Reference type Integer formal argument :15**

**Primitive type long formal argument :100**

If you are using wrapper class Object as an actual argument and compiler does not find the method with parameter(s) of the same reference type (i.e. class or interface type), then it starts searching a method with parameter(s) having the corresponding primitive data type.

**Method Overloading with Widening**

If compiler fails to find any method corresponding to autoboxing, then it starts searching a method parameter(s) of the widened primitive data type.

We are invoking another method with argument of Long wrapper Object. Compiler starts searching for the method having the same reference type (Long wrapper class). Since there is no method having with parameter of **Long wrapper class**. So, It searches for method which can accept the parameter bigger than long primitive data type as an argument

*// invoking the method with signature has widened data type*

*c.method(10);*

*c.method(new Long(100));*

*public void method(int i)*

*{*

*System.out.println("Primitive type int formal argument :" + i);*

*}*

*public void method(float i)*

*{*

*System.out.println("Primitive type float formal argument :" + i);*

*}*

Output:

Primitive type int formal argument :10

Primitive type float formal argument :100.0

1. **What happens if widening and boxing happen together? What method invocation will compiler be able to do?**

Ans- Widening of primitive type has taken priority over boxing and var-args. But widening and boxing of primitive type can not work together.

*//overloaded method primitive(byte) var-args formal argument*

*public void method(byte... a)*

*{*

*System.out.println("Primitive type byte formal argument :" + a);*

*}*

*// overloaded method primitive(int) formal arguments*

*public void method(long a, long b)*

*{*

*System.out.println("Widening type long formal argument :" + a);*

*}*

*// invokes the method having widening primitive type parameters.*

*byte val = 5;*

*c.method(val,val);*

Output:

Widening type long formal argument :5

1. **Shadowing of static members in JAVA?**

Ans- if name of a derived class static function is same as base class static function then the base class static function shadows (or conceals) the derived class static function.

*class A {*

*static void fun() {*

*System.out.println("A.fun()");*

*}*

*}*

*class B extends A {*

*static void fun() {*

*System.out.println("B.fun()");*

*}*

*}*

*public class Main {*

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

*A a = new B();*

*a.fun();  // prints A.fun()*

*}*

*}*

If we make both A.fun() and B.fun() as non-static then the above program would print “B.fun()”.

1. **Instance and static methods in JAVA?**

Ans-

**Instance method**

* They are the methods which require an object of its class to be created before it can be called. To invoke a instance method, we have to create an Object of the class in within which it defined.
* They can be overridden since they are resolved using **dynamic binding** at run time.
* **Memory allocation:**These methods themselves are stored in Permanent Generation space of heap but the parameters (arguments passed to them) and their local variables and the value to be returned are allocated in stack.

**Static Method**

* Static methods are the methods in Java that can be called without creating an object of class. They are referenced by the **class name itself** or reference to the Object of that class. They are designed with aim to be shared among all Objects created from the same class.
* Their memory allocation is same as of instance methods
* Static methods cannot be overridden. But can be overloaded since they are resolved using **static binding** by compiler at compile time.

**Instance method vs Static method**

* Instance method can access the instance methods and instance variables directly.
* Instance method can access static variables and static methods directly.
* Static methods can access the static variables and static methods directly.
* Static methods can’t access instance methods and instance variables directly. They must use reference to object. And static method can’t use [this](http://quiz.geeksforgeeks.org/this-reference-in-java/) keyword as there is no instance for ‘this’ to refer to.