**Features of Java :**

There are the following features in Java Programming Language.

* **Simple:** Java is easy to learn. The syntax of Java is based on C++ which makes easier to write the program in it.
* **Object-Oriented:** Java follows the object-oriented paradigm which allows us to maintain our code as the combination of different type of objects that incorporates both data and behavior.
* **Portable:** Java supports read-once-write-anywhere approach. We can execute the Java program on every machine. Java program (.java) is converted to bytecode (.class) which can be easily run on every machine.
* **Platform Independent:** Java is a platform independent programming language. It is different from other programming languages like C and C++ which needs a platform to be executed. Java comes with its platform on which its code is executed. Java doesn't depend upon the operating system to be executed.
* **Secured:** Java is secured because it doesn't use explicit pointers. Java also provides the concept of ByteCode and Exception handling which makes it more secured.
* **Robust:** Java is a strong programming language as it uses strong memory management. The concepts like Automatic garbage collection, Exception handling, etc. make it more robust.
* **Architecture Neutral:** Java is architectural neutral as it is not dependent on the architecture. In C, the size of data types may vary according to the architecture (32 bit or 64 bit) which doesn't exist in Java.
* **Interpreted:** Java uses the Just-in-time (JIT) interpreter along with the compiler for the program execution.
* **High Performance:** Java is faster than other traditional interpreted programming languages because Java bytecode is "close" to native code. It is still a little bit slower than a compiled language (e.g., C++).
* **Multithreaded:** We can write Java programs that deal with many tasks at once by defining multiple threads. The main advantage of multi-threading is that it doesn't occupy memory for each thread. It shares a common memory area. Threads are important for multi-media, Web applications, etc.
* **Distributed:** Java is distributed because it facilitates users to create distributed applications in Java. RMI and EJB are used for creating distributed applications. This feature of Java makes us able to access files by calling the methods from any machine on the internet.
* **Dynamic:** Java is a dynamic language. It supports dynamic loading of classes. It means classes are loaded on demand. It also supports functions from its native languages, i.e., C and C++.

**What is the difference between JDK, JRE, and JVM?**

### JVM

JVM is an acronym for Java Virtual Machine; it is an abstract machine which provides the runtime environment in which Java bytecode can be executed. It is a specification which specifies the working of Java Virtual Machine. Its implementation has been provided by Oracle and other companies. Its implementation is known as JRE.

JVMs are available for many hardware and software platforms (so JVM is platform dependent). It is a runtime instance which is created when we run the Java class. There are three notions of the JVM: specification, implementation, and instance.

### JRE

JRE stands for Java Runtime Environment. It is the implementation of JVM. The Java Runtime Environment is a set of software tools which are used for developing Java applications. It is used to provide the runtime environment. It is the implementation of JVM. It physically exists. It contains a set of libraries + other files that JVM uses at runtime.

### JDK

JDK is an acronym for Java Development Kit. It is a software development environment which is used to develop Java applications and applets. It physically exists. It contains JRE + development tools. JDK is an implementation of any one of the below given Java Platforms released by Oracle Corporation:

* Standard Edition Java Platform
* Enterprise Edition Java Platform
* Micro Edition Java Platform

### How many types of memory areas are allocated by JVM?

Many types:

1. **Class(Method) Area:** Class Area stores per-class structures such as the runtime constant pool, field, method data, and the code for methods.
2. **Heap:** It is the runtime data area in which the memory is allocated to the objects
3. **Stack:** Java Stack stores frames. It holds local variables and partial results, and plays a part in method invocation and return. Each thread has a private JVM stack, created at the same time as the thread. A new frame is created each time a method is invoked. A frame is destroyed when its method invocation completes.
4. **Program Counter Register:** PC (program counter) register contains the address of the Java virtual machine instruction currently being executed.
5. **Native Method Stack:** It contains all the native methods used in the application.

### What are the various access specifiers in Java?

In Java, access specifiers are the keywords which are used to define the access scope of the method, class, or a variable. In Java, there are four access specifiers given below.

* **Public** The classes, methods, or variables which are defined as public, can be accessed by any class or method.
* **Protected** Protected can be accessed by the class of the same package, or by the sub-class of this class, or within the same class.
* **Default** Default are accessible within the package only. By default, all the classes, methods, and variables are of default scope.
* **Private** The private class, methods, or variables defined as private can be accessed within the class only.

### What is the purpose of static methods and variables?

The methods or variables defined as static are shared among all the objects of the class. The static is the part of the class and not of the object. The static variables are stored in the class area, and we do not need to create the object to access such variables. Therefore, static is used in the case, where we need to define variables or methods which are common to all the objects of the class.

For example, In the class simulating the collection of the students in a college, the name of the college is the common attribute to all the students. Therefore, the college name will be defined as **static**.

### What is an object?

### The Object is the real-time entity having some state and behavior. In Java, Object is an instance of the class having the instance variables as the state of the object and the methods as the behavior of the object. The object of a class can be created by using the ****new**** keyword.

### What is the constructor?

### The constructor can be defined as the special type of method that is used to initialize the state of an object. It is invoked when the class is instantiated, and the memory is allocated for the object. Every time, an object is created using the **new** keyword, the default constructor of the class is called. The name of the constructor must be similar to the class name. The constructor must not have an explicit return type.

### How many types of constructors are used in Java?

Based on the parameters passed in the constructors, there are two types of constructors in Java.

* **Default Constructor:** default constructor is the one which does not accept any value. The default constructor is mainly used to initialize the instance variable with the default values. It can also be used for performing some useful task on object creation. A default constructor is invoked implicitly by the compiler if there is no constructor defined in the class.
* **Parameterized Constructor:** The parameterized constructor is the one which can initialize the instance variables with the given values. In other words, we can say that the constructors which can accept the arguments are called parameterized constructors.

### What are the differences between the constructors and methods?

### 

**Collection :**

**Differentiate between Collection and Collections.**

|  |  |
| --- | --- |
| **Collection** | **Collections** |
| java.util.Collection is an interface | java.util.Collections is a class |
| Is used to represent a group of objects as a single entity | It is used to define various utility method for collection objects |
| It is the root interface of the Collection framework | It is a utility class |
| It is used to derive the data structures of the Collection framework | It contains various static methods which help in data structure manipulation |

**. Differentiate between an Array and an ArrayList.**

|  |  |
| --- | --- |
| **Array** | **ArrayList** |
| java.util.Array is a class | java.util.ArrayList is a class |
| It is strongly typed | It is loosely types |
| Cannot be dynamically resized | Can be dynamically resized |
| No need to box and unbox the elements | Needs to box and unbox the elements |

**Differentiate between Iterable and Iterator.**

|  |  |
| --- | --- |
| **Iterable** | **Iterator** |
| Iterable is an interface | Iterator is an interface |
| Belongs to java.lang package | Belongs to java.util package |
| Provides one single abstract method called iterator() | Provides two abstract methods called hasNext() and next() |
| It is a representation of a series of elements that can be traversed | It represents the object with iteration state |

**Differentiate between ArrayList and LinkedList.**

|  |  |
| --- | --- |
| **ArrayList** | **LinkedList** |
| Implements dynamic array internally to store elements | Implements doubly linked list internally to store elements |
| Manipulation of elements is slower | Manipulation of elements is faster |
| Can act only as a List | Can act as a List and a Queue |
| Effective for data storage and access | Effective for data manipulation |

**. Differentiate between Comparable and Comparator.**

|  |  |
| --- | --- |
| **Comparable** | **Comparator** |
| Present in java.lang package | Present in java.util package |
| Elements are sorted based on natural ordering | Elements are sorted based on user-customized ordering |
| Provides a single method called compareTo() | Provides to methods equals() and compare() |
| Modifies the actual class | Doesn’t modifies the actual class |

**. Differentiate between List and Set.**

|  |  |
| --- | --- |
| **List** | **Set** |
| An ordered collection of elements | An unordered collection of elements |
| Preserves the insertion order | Doesn’t preserves the insertion order |
| Duplicate values are allowed | Duplicate values are not allowed |
| Any number of null values can be stored | Only one null values can be stored |
| ListIterator can be used to traverse the List in any direction | ListIterator cannot be used to traverse a Set |
| Contains a legacy class called vector | Doesn’t contains any legacy class |

**Differentiate between Set and Map.**

|  |  |
| --- | --- |
| **Set** | **Map** |
| Belongs to java.util package | Belongs to java.util package |
| Extends the Collection interface | Doesn’t extend the Collection interface |
| Duplicate values are not allowed | Duplicate keys are not allowed but duplicate values are |
| Only one null values can be stored | Only one null key can be stored but multiple null values are allowed |
| Doesn’t maintain any insertion order | Doesn’t maintain any insertion order |

**Differentiate between List and Map.**

|  |  |
| --- | --- |
| **List** | **Map** |
| Belongs to java.util package | Belongs to java.util package |
| Extends the Collection interface | Doesn’t extend the Collection interface |
| Duplicate elements are allowed | Duplicate keys are not allowed but duplicate values are |
| Multiple null values can be stored | Only one null key can be stored but multiple null values are allowed |
| Preserves the insertion order | Doesn’t maintain any insertion order |
| Stores elements based on Array Data Structure | Stores data in key-value pairs using various hashing techniques |

**Differentiate between Queue and Stack.**

|  |  |
| --- | --- |
| **Queue** | **Stack** |
| Based on FIFO (First-In-First-Out) principle | Based on LIFO (Last-In-First-Out) principle |
| Insertion and deletion takes place from two opposite ends | Insertion and deletion takes place the same end |
| Element insertion is called enqueue | Element insertion is called push |
| Element deletion is called dequeue | Element deletion is called pop |
| Two pointers are maintained one point to the first element and the other one points the last element on the list | Only one pointer is maintained which points to the top element on the stack |

**. Differentiate between PriorityQueue and TreeSet.**

|  |  |
| --- | --- |
| **PriorityQueue** | **TreeSet** |
| It is a type of Queue | It is based on a Set data structure |
| Allows duplicate elements | Doesn’t allows duplicate elements |
| Stores the elements based on an additional factor called priority | Stores the elements in a sorted order |

**Differentiate between the Singly Linked List and Doubly Linked List.**

|  |  |
| --- | --- |
| **Singly Linked List(SLL)** | **Doubly Linked List(DLL)** |
| Contains nodes with a data field and a next node-link field | Contains nodes with a data field, a previous link field, and a next link field |
| Can be traversed using the next node-link field only | Can be traversed using the previous node-link or the next node-link |
| Occupies less memory space | Occupies more memory space |
| Less efficient in providing access to the elements | More efficient in providing access to the elements |

**Differentiate between Iterator and Enumeration.**

|  |  |
| --- | --- |
| **Iterator** | **Enumeration** |
| Collection element can be removed while traversing it | Can only traverse through the Collection |
| Used to traverse most of the classes of the Java Collection framework | Used to traverse the legacy classes such as Vector, HashTable, etc |
| Is fail-fast in nature | Is fail-safe in nature |
| Is safe and secure | Is not safe and secure |
| Provides methods like hasNext(), next() and remove() | Provides methods like hasMoreElements() and nextElement() |

**. Differentiate between HashMap and HashTable.**

|  |  |
| --- | --- |
| **HashMap** | **HashTable** |
| It is non-synchronized in nature | It is synchronized in nature |
| Allows only one null key but multiple null values | Doesn’t allow any null key or value |
| Has faster processing | has slower processing |
| Can be traversed by Iterator | Can be traversed by Iterator and Enumeration |
| Inherits AbstractMap class | Inherits Dictionary class |

**Differentiate between HashSet and HashMap.**

|  |  |
| --- | --- |
| **HashSet** | **HasMap** |
| Based on Set implementation | Based on Map implementation |
| Doesn’t allow any duplicate elements | Doesn’t allow any duplicate keys but duplicate values are allowed |
| Allows only a single null value | Allows only one null key but any number of null values |
| Has slower processing time | Has faster processing time |
| Uses HashMap as an underlying data structure | Uses various hashing techniques for data manipulation |

**Differentiate between Iterator and ListIterator.**

|  |  |
| --- | --- |
| **Iterator** | **ListIterator** |
| Can only perform remove operations on the Collection elements | Can perform add, remove and replace operations the Collection elements |
| Can traverse List, Sets and maps | Can traverse only Lists |
| Can traverse the Collection in forward direction | Can traverse the collection in any direction |
| Provides no method to retrieve the index of the element | Provides methods to retrieve the index of the elements |
| iterator() method is available for the entire Collection Framework | listIterator() is only available for the collections implementing the List interface |

**. Differentiate between HashSet and TreeSet.**

|  |  |
| --- | --- |
| **HashSet** | **TreeSet** |
| Uses HasMap to store elements | Uses Treemap to store elements |
| It is unordered in nature | By default, it stores elements in their natural ordering |
| Has faster processing time | Has slower processing time |
| Uses hasCode() and equals() for comparing | Uses compare() and compareTo() for comparing |
| Allows only one null element | Doesn’t allow any null element |
| Takes up less memory space | Takes up more memory space |

**. Differentiate between Queue and Deque.**

|  |  |
| --- | --- |
| **Queue** | **Deque** |
| Refers to single-ended queue | Refers to double-ended queue |
| Elements can be added or removed from only one end | Elements can be added and removed from either end |
| Less versatile | More versatile |

**. Differentiate between HashMap and TreeMap.**

|  |  |
| --- | --- |
| **HashMap** | **TreeMap** |
| Doesn’t preserves any ordering | Preserves the natural ordering |
| Implicitly implements the hashing principle | Implicitly implements the Red-Black Tree Implementation |
| Can store only one null key | Cannot store any null key |
| More memory usage | Less memory usage |
| Not synchronized | Not synchronized |

**. Differentiate between ArrayList and Vector.**

|  |  |
| --- | --- |
| **ArrayList** | **Vector** |
| Non-synchronized in nature | Synchronized in nature |
| It is not a legacy class | Is a legacy class |
| Increases size by 1/2 of the ArrayList | Increases size by double of the ArrayList |
| It is not thread-safe | It is thread-safe |

**Differentiate between failfast and failsafe.**

|  |  |
| --- | --- |
| **failfast** | **failsafe** |
| Doesn’t allow modifications of a collection while iterating | Allows modifications of a collection while iterating |
| Throws ConcurrentModificationException | Don’t throw any exceptions |
| Uses the original collection to traverse over the elements | Uses a copy of the original collection to traverse over the elements |
| Don’t require extra memory | Require extra memory |
| **Spring Boot :**  **Annotations In Spring Boot :**  **@Required:** It applies to the **bean** setter method. It indicates that the annotated bean must be populated at configuration time with the required property, else it throws an exception **BeanInitilizationException**.  **@Autowired:** Spring provides annotation-based auto-wiring by providing @Autowired annotation. It is used to autowire spring bean on setter methods, instance variable, and constructor. When we use @Autowired annotation, the spring container auto-wires the bean by matching data-type.  **@Configuration:** It is a class-level annotation. The class annotated with @Configuration used by Spring Containers as a source of bean definitions.  **@ComponentScan:** It is used when we want to scan a package for beans. It is used with the annotation @Configuration. We can also specify the base packages to scan for Spring Components.  **@Bean:** It is a method-level annotation. It is an alternative of XML <bean> tag. It tells the method to produce a bean to be managed by Spring Container.  **@Component:** It is a class-level annotation. It is used to mark a Java class as a bean. A Java class annotated with **@Component** is found during the classpath. The Spring Framework pick it up and configure it in the application context as a **Spring Bean**.  **@Controller:** The @Controller is a class-level annotation. It is a specialization of **@Component**. It marks a class as a web request handler. It is often used to serve web pages. By default, it returns a string that indicates which route to redirect. It is mostly used with **@RequestMapping** annotation.  **@Service:** It is also used at class level. It tells the Spring that class contains the **business logic**.  **@Repository:** It is a class-level annotation. The repository is a **DAOs** (Data Access Object) that access the database directly. The repository does all the operations related to the database.  **@EnableAutoConfiguration:** It auto-configures the bean that is present in the classpath and configures it to run the methods. The use of this annotation is reduced in Spring Boot 1.2.0 release because developers provided an alternative of the annotation, i.e. **@SpringBootApplication**.  **@SpringBootApplication:** It is a combination of three annotations **@EnableAutoConfiguration, @ComponentScan,** and **@Configuration**.  **@RequestMapping:** It is used to map the **web requests**. It has many optional elements like **consumes, header, method, name, params, path, produces**, and **value**. We use it with the class as well as the method.  **@GetMapping:** It maps the **HTTP GET** requests on the specific handler method. It is used to create a web service endpoint that **fetches** It is used instead of using: **@RequestMapping(method = RequestMethod.GET)**  **@PostMapping:** It maps the **HTTP POST**requests on the specific handler method. It is used to create a web service endpoint that **creates** It is used instead of using: **@RequestMapping(method = RequestMethod.POST)**  **@PutMapping:** It maps the **HTTP PUT** requests on the specific handler method. It is used to create a web service endpoint that **creates** or **updates** It is used instead of using: **@RequestMapping(method = RequestMethod.PUT)**  **@DeleteMapping:** It maps the **HTTP DELETE** requests on the specific handler method. It is used to create a web service endpoint that **deletes**a resource. It is used instead of using: **@RequestMapping(method = RequestMethod.DELETE)**  **@PatchMapping:** It maps the **HTTP PATCH**requests on the specific handler method. It is used instead of using: **@RequestMapping(method = RequestMethod.PATCH)**  **@RequestBody:** It is used to **bind** HTTP request with an object in a method parameter. Internally it uses **HTTP MessageConverters** to convert the body of the request. When we annotate a method parameter with **@RequestBody,** the Spring framework binds the incoming HTTP request body to that parameter.  **@ResponseBody:** It binds the method return value to the response body. It tells the Spring Boot Framework to serialize a return an object into JSON and XML format.  **@PathVariable:** It is used to extract the values from the URI. It is most suitable for the RESTful web service, where the URL contains a path variable. We can define multiple @PathVariable in a method.  **@RequestParam:** It is used to extract the query parameters form the URL. It is also known as a **query parameter**. It is most suitable for web applications. It can specify default values if the query parameter is not present in the URL.  **@RequestHeader:** It is used to get the details about the HTTP request headers. We use this annotation as a **method parameter**. The optional elements of the annotation are **name, required, value, defaultValue.**For each detail in the header, we should specify separate annotations. We can use it multiple time in a method  **@RestController:** It can be considered as a combination of **@Controller** and **@ResponseBody**annotations**.** The @RestController annotation is itself annotated with the @ResponseBody annotation. It eliminates the need for annotating each method with @ResponseBody.  **@RequestAttribute:** It binds a method parameter to request attribute. It provides convenient access to the request attributes from a controller method. With the help of @RequestAttribute annotation, we can access objects that are populated on the server-side.  **Java 8 features :**  There is list of below features of java 1.8 as  Functional interface  Lambda Expression  Default method  Static method  For each Method  Optional Class  String Joiners  **Functional Interface**  An Interface that contains exactly one abstract method is known as functional interface.  It can have any number of default, static methods but can contain only one abstract method.  Example-  **package** com.test;  @FunctionalInterface  **public** **interface** Test {  **void** getStudentName(String name);    }  **package** com.test;  **public** **class** Main **implements** Test {  @Override  **public** **void** getStudentName(String name) {  System.***out***.println(name);  }  **public** **static** **void** main(String[] args) {  Main main = **new** Main();  main.getStudentName("ashok");  }  }  **Lambda Expression-**  Why?  Less coding  Syntax- (argument-list) -> {body}     * **Argument-list:** It can be empty or non-empty as well. * **Arrow-token:** It is used to link arguments-list and body of expression. * **Body:** It contains expressions and statements for lambda expression.   **No Parameter Syntax**  () -> {  //Body of no parameter lambda  }  **One Parameter Syntax**  (p1) -> {  //Body of single parameter lambda  }  **Two Parameter Syntax**  (p1,p2) -> {  //Body of multiple parameter lambda  }  Example-  **package** com.test;  **public** **interface** Addition {  **int** add(**int** a,**int** b);  }  **package** com.test;  **public** **class** Main {  **public** **static** **void** main(String[] args) {  // Multiple parameters in lambda expression  Addition addition = (a, b) -> (a + b);  System.***out***.println(addition.add(10, 20));  // Multiple parameters with data type in lambda expression  Addition addition2 = (**int** a, **int** b) -> (a + b);  System.***out***.println(addition2.add(100, 200));  }  }  Output  30  300  **Default method**  Java provides a facility to create default methods inside the interface. Methods which are defined inside the interface and tagged with default are known as default methods. These methods are non-abstract methods.  Example-  **package** com.test;  **public** **interface** Example {  **default** **void** m1() {  System.***out***.println("this is default m1 method");  }  }  **package** com.test;  **public** **class** TestMain **implements** Example {  **public** **static** **void** main(String[] args) {    TestMain testMain=**new** TestMain();  testMain.m1();  }  }  Output  this is default m1 method  **Static method-**  Java provides a facility to create static methods inside the interface.  **package** com.demo;  **public** **interface** Example {  **static** **void** x1() {  System.***out***.println("this is static method");  }  }  **package** com.demo;  **public** **class** MainTest **implements** Example{  **public** **static** **void** main(String[] args) {    Example.*x1*();  }  }  **forEach () method-**  The Java forEach() method is a utility function to iterate over a collection such as (list, set or map) and [stream](https://howtodoinjava.com/java8/java-streams-by-examples/). It is used to perform a given action on each the element of the collection.  **package** com.test;  **import** java.util.HashMap;  **import** java.util.Map;  **public** **class** MapDemo {  **public** **static** **void** main(String[] args) {  Map<String, String> map = **new** HashMap<String, String>();  map.put("10", "ram");  map.put("11", "shyam");  map.put("12", "ganesh");  map.forEach((k, v) -> System.***out***.println("Key = " + k + ", Value = " + v));  }  }  Output  Key = 11, Value = shyam  Key = 12, Value = ganesh  Key = 10, Value = ram  **Optional class-**  Java introduced a new class Optional in jdk8. It is a public final class and used to deal with NullPointerException in Java application.  You must import java.util package to use this class. It provides methods which are used to check the presence of value for particular variable.  Why?  **package** com.test;  **public** **class** MapDemo {  **public** **static** **void** main(String[] args) {  String[] str = **new** String[10];  String lowercaseString = str[5].toLowerCase();  System.***out***.print(lowercaseString);  }  }  Exception in thread "main" java.lang.NullPointerException  at com.test.MapDemo.main(MapDemo.java:8)  Here we are getting exception, to avoid this type of exception, we should go for optional class  **package** com.test;  **import** java.util.Optional;  **public** **class** MapDemo {  **public** **static** **void** main(String[] args) {  String[] str = **new** String[10];  Optional<String> checkNull = Optional.*ofNullable*(str[5]);  **if** (checkNull.isPresent()) { // check for value is present or not  String lowercaseString = str[5].toLowerCase();  System.***out***.print(lowercaseString);  } **else**  System.***out***.println("string value is not present");  }  }  Output  string value is not present.  **Java String Joiner-**  Java added a new final class StringJoiner in java.util package. It is used to construct a sequence of characters separated by a delimiter. Now, you can create string by passing delimiters like comma(,), hyphen(-) etc  Example  **import** java.util.StringJoiner;  **public** **class** Example {  **public** **static** **void** main(String[] args) {  StringJoiner stringJoiner = **new** StringJoiner(","); // passing comma(,) as delimiter  // Adding values to StringJoiner  stringJoiner.add("Ram");  stringJoiner.add("Shyam");  stringJoiner.add("ashok");  stringJoiner.add("ajay");  System.***out***.println(stringJoiner);  }  }  Output  Ram,Shyam,ashok,ajay  **Hibernate :**  Hibernate frameworks is mediator through which java application is communicated with database. It is open source frameworks. It is Object Relational Mapping (ORM) tool.  **Why?**   * In JDBC, if we open a database connection we need to write in try, and if any exceptions occurred catch block will takers about it, and finally used to close the connections. * We must close the connection, or we may get a chance to get connections error message. * Actually if we didn’t close the connection in the finally block, then jdbc doesn’t responsible to close that connection. * In JDBC we need to write Sql commands in various places, after the program has created if the table structure is modified then the JDBC program doesn’t work, again we need to modify and compile and re-deploy required, which is tedious. * To overcome above drawbacks we should go for Hibernate framework.   **Advantages of Hibernate-**   1. It is open source frameworks. 2. Faster performance-                It uses cache concept hence the performance is fast.   1. Database independent query-                It generates the database independent query.   1. Automatic table creation-                It has facility to create the database tables automatically. There is no need to create the database tables manually.   1. Simplifies the complex join-                It is easy to fetch the data from multiple tables in hibernate framework.  **Hibernate Cache Support-**  **Why?**  To reduce the number of calls from database and improve the application performance.  **How it works?**              It will fetch only one time data from database and store it on local cache (temporary storage).               For n no. of user retrieve data only once from database and store it to cache and return it to user.  There are **three types** of cache in the hibernate as  Session level cache  SessionFactory level cache  Query level cache  Problem without cache-              Suppose I have Java study material application or site. It is the constant data, suppose 1 lakh user daily visit to site to read the concepts, so it will hit 1 lakh times  to database due to this your application will slow and lot of processing time it will takes. To overcome this issue, we should go for cache supports.  **Session level cache-**  It is called as first level cache.  It is apply for single user or one user only.  It is default level cache.  The first level cache data will not be available to entire application because application can use many session objects.  **When to use?**  Example- Login to Gmail application, if you want to retrieve the inbox mails at first time login. It will load the data from database. If you trying to refresh and if you do not have new mails. The data instead of reading every time from database, it will load from cache itself until doing logout. It will do only one select operation.  Note-  It will fetch data only one time from database and store it on session objects next time when user request some data, so it will retrieve data from session objects.  **Query level cache-**  **Why?**        If I want to execute some query again and again or multiple times (Example select max salary from employee) then you should go for query level cache.  **SessionFactory level cache-**  It is called as second level cache.  It will apply for all the users.  Program-2 Using SessionFactory  **Why?**  Problem with session level cache  https://lh6.googleusercontent.com/GXh0Odbo2osxXGz9TF02-ACi5-_OjuPRdQKAznBNCfg4pnwhNlITDf7JrzLZFM7xKPRShcdU4NUrn7PHnZ1M3e68PspGQTXUhgeWa-6V1JnG8o-QF6yLYXMzcrMgeJkHBCg3DLZmaznE4mBxppwidHnbQiT9V3F2PxmNCw1nk7jXs9RRNTD6wM2co7275fn-LW1QCg              Here admin1 and admin2 are the trying to access the same data, if we are using session, it will create the two session objects for some data or common data but if we use sessionfactory cache, data will fetch from database only one(data store under the sessionfactory) and data will available to any user.  **When?**  It is useful if you have multiple session objects from sessionfactory. |  |