**Java**

**Immutability**

1. What do you mean by immutability?

Immutable class or object, Immutable objects are those, whose state cannot be changed

1. What is the use of immutable objects

* Immutable object not only guarantees safe publication of object’s state, but also can be shared among other threads without any external synchronization.
* Immutable objects are by default thread safe, can be shared without synchronization in concurrent environment.

1. Example of Immutable classes in Java.

All wrapper classes in java.lang are immutable –   
String, Integer, Boolean, Character, Byte, Short, Long, Float, Double, BigDecimal, BigInteger

1. How to construct Immutable objects.

* All fields of Immutable class should be final.
* Object should be final in order to restrict sub-class for altering immutability of parent class.

1. Will constructor be public or private for Immutable class.
2. Example of**simple** immutable class in java

public final class Contacts {

private final String name;

private final String mobile;

public Contacts(String name, String mobile) {

this.name = name;

this.mobile = mobile;

}

public String getName(){

return name;

}

public String getMobile(){

return mobile;

}

}

1. Example of Immutable class with complex object like **Date** or **HashMap**

Its advised to **return copy of original object**

publicfinalclassImmutableReminder{

privatefinalDateremindingDate;

publicImmutableReminder (DateremindingDate) {

if(remindingDate.getTime() <System.currentTimeMillis()){

thrownewIllegalArgumentException("Can not set reminder” +

“ for past time: "+remindingDate);

}

this.remindingDate=newDate(remindingDate.getTime());

}

publicDategetRemindingDate() {

return (Date) remindingDate.clone();

}

}

1. Why String is popular HashMap key in Java?

Since String is immutable, its hashcode is cached at the time of creation and it doesn’t need to be calculated again. This makes it a great candidate for key in a Map and it’s processing is fast than other HashMap key objects. This is why String is mostly used Object as HashMap keys.

**Exception**

1. What is the purpose of the throw and throws keywords?

The **throws** keyword is used to specify that a method may raise an exception during its execution. It enforces explicit exception handling when calling a method:

public void simpleMethod() throws Exception {

// ...

}

The **throw** keyword allows us to throw an exception object to interrupt the normal flow of the program. This is most commonly used when a program fails to satisfy a given condition:

if (task.isTooComplicated()) {

throw new TooComplicatedException("The task is too complicated");

}

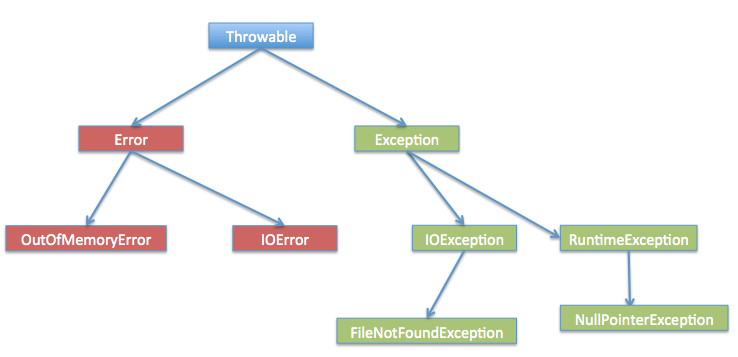
1. What are checked and unchecked exceptions, with Examples.

* A checked exception must be handled within a try-catch block or declared in a throws clause; whereas an unchecked exception is not required to be handled nor declared.
* Checked and unchecked exceptions are also known as compile-time and runtime exceptions respectively.
* All exceptions are checked exceptions, except those indicated by Error, RuntimeException, and their subclasses.

1. Can we create our own Exception class, if yes how.
2. What is a stacktrace and how does it relate to an exception?

* A stack trace provides the names of the classes and methods that were called, from the start of the application to the point an exception occurred.
* It’s a very useful debugging tool since it enables us to determine exactly where the exception was thrown in the application and the original causes that led to it.

1. Difference between final, finally and finalize
2. Try-catch and try-finally combination, what is required.
3. Exception hierarchy



1. Can we have an empty catch block?

Yes

1. Can we create our own error class, if Yes how.
2. Simples program to throw an error

**Collections**

1. ***What type of Collections have you used.***
2. ***Explain the difference between LinkedList and ArrayList.***

* **ArrayList** is an implementation of the List interface that is based on an array.
* ArrayList internally handles resizing of this array when the elements are added or removed. You can access its elements in constant time by their index in the array.
* However, inserting or removing an element infers shifting all consequent elements which may be slow if the array is huge and the inserted or removed element is close to the beginning of the list.

1. ***What is the difference between HashSet and TreeSet?***

Both ***HashSet*** and ***TreeSet*** classes implement the *Set* interface and represent sets of distinct elements. Additionally, *TreeSet* implements the *NavigableSet* interface. This interface defines methods that take advantage of the ordering of elements.

*HashSet* is internally based on a *HashMap*, and *TreeSet* is backed by a *TreeMap* instance, which defines their properties:

*HashSet* does not keep elements in any particular order. Iteration over the elements in a *HashSet* produces them in a shuffled order. *TreeSet*, on the other hand, produces elements in order according to some predefined *Comparator*.

1. ***How is HashMap implemented in Java? How does its implementation use hashCode and equals methods of objects? What is the time complexity of putting and getting an element from such structure?***

The HashMap class represents a typical hash map data structure with certain design choices.

The HashMap is **backed by a resizable array that has a size of power-of-two**. When the element is added to a HashMap, **first its hashCode is calculated** (an int value). Then a **certain number of lower bits of this value are used as an array index**. This **index directly points to the cell of the array (called a bucket)** where this key-value pair should be placed. Accessing an element by its index in an array is a very **fast O(1) operation**, which is the main feature of a hash map structure.

A hashCode is not unique, however, and even for different hashCodes, we may receive the same array position. This is called a **collision**. There is more than one way of resolving collisions in the hash map data structures. In Java’s HashMap, **each bucket actually refers not to a single object, but to a red-black tree of all objects that landed in this bucket** (prior to Java 8, this was a linked list).

So when the HashMap has determined the bucket for a key, it has to traverse this tree to put the key-value pair in its place. If a pair with such key already exists in the bucket, it is replaced with a new one.

To retrieve the object by its key, the HashMap again has to calculate the hashCode for the key, find the corresponding bucket, traverse the tree, call equals on keys in the tree and find the matching one.

HashMap has O(1) complexity, or constant-time complexity, of putting and getting the elements. Of course, lots of collisions could degrade the performance to O(log(n)) time complexity in the worst case, when all elements land in a single bucket. This is usually solved by providing a good hash function with a uniform distribution.

When the HashMap internal array is filled (more on that in the next question), it is automatically resized to be twice as large. This operation infers rehashing (rebuilding of internal data structures), which is costly, so you should plan the size of your HashMap beforehand.

1. ***What is the purpose of the initial capacity and load factor parameters of a HashMap? What are their default values?***

The initialCapacity argument of the HashMap constructor affects the size of the internal data structure of the HashMap, but reasoning about the actual size of a map is a bit tricky. The HashMap‘s internal data structure is an array with the power-of-two size. So the initialCapacity argument value is increased to the next power-of-two (for instance, if you set it to 10, the actual size of the internal array will be 16).

The load factor of a HashMap is the ratio of the element count divided by the bucket count (i.e. internal array size). For instance, if a 16-bucket HashMap contains 12 elements, its load factor is 12/16 = 0.75. A high load factor means a lot of collisions, which in turn means that the map should be resized to the next power of two. So the loadFactor argument is a maximum value of the load factor of a map. When the map achieves this load factor, it resizes its internal array to the next power-of-two value.

The initialCapacity is 16 by default, and the loadFactor is 0.75 by default, so you could put 12 elements in a HashMap that was instantiated with the default constructor, and it would not resize. The same goes for the HashSet, which is backed by a HashMap instance internally.

1. ***What is the difference between fail-fast and fail-safe iterators?***

Iterators for different collections are either fail-fast or fail-safe, depending on how they react to concurrent modifications. The concurrent modification is not only a modification of collection from another thread but also modification from the same thread but using another iterator or modifying the collection directly.

**Fail-fast** iterators (those returned by HashMap, ArrayList, and other non-thread-safe collections) iterate over the collection’s internal data structure, and they throw ConcurrentModificationException as soon as they detect a concurrent modification.

**Fail-safe** iterators (returned by thread-safe collections such as ConcurrentHashMap, CopyOnWriteArrayList) create a copy of the structure they iterate upon. They guarantee safety from concurrent modifications. Their drawbacks include excessive memory consumption and iteration over possibly out-of-date data in case the collection was modified.

1. ***How can you use Comparable and Comparator interfaces to sort collections?***

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | **No.** | **Comparable** | **Comparator** | | 1) | Comparable provides only one sort of sequence. | Comparator provides multiple sort of sequences. | | 2) | It provides one method named compareTo(). | It provides one method named compare(). | | 3) | It is found in java.lang package. | it is found in java.util package. | | 4) | If we implement Comparable interface, actual class is modified. | Actual class is not modified. | |

1. [***Difference between Java Collection and Collections***](https://stackoverflow.com/questions/1796275/difference-between-java-collection-and-collections)

Collection is a base interface for most collection classes, whereas Collections is a utility class. I recommend you read the documentation.

1. ***What is the difference between Set and List in Java?***

Set doesn't allowed duplicate while List does and List maintains insertion order while Set doesn't.

1. ***What is CopyOnWriteArrayList, how it is different than ArrayList***

CopyOnWriteArrayList is new List implementation introduced in Java 1.5 which provides better concurrent access than Synchronized List. better concurrency is achieved by Copying ArrayList over each write and replace with original instead of locking. Also CopyOnWriteArrayList doesn't throw any ConcurrentModification Exception. Its different than ArrayList because its thread-safe and ArrayList is not thread-safe

# Working of Collections.synchronizedMap

* Collections class defines a private nested static class named SynchronizedMap.
* SynchronizedMap has two instance variables **mutex** and **m.**
* **mutex**serves the purpose of a mutex variable in working of [Collections.synchronizedMap](https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#synchronizedMap-java.util.Map-) and **m**server the purpose of holding original collection.

This is how SynchronizedMap is defined roughly.

private static class SynchronizedMap<K,V>

implements Map<K,V>, Serializable {

private final Map<K,V> m; // references original map

final Object mutex; // server the purpose of mutex in sychronized methods.

}

* Note that we are passing a **map** created as **HashMap**to **Collections.synchronizedMap(map);**
* It instantiates **SynchronizedMap** and passes **m** which refers to object referred by **map.**So, **m**is backing **map** now.
* **mutex**refers to current **SynrozniedMap** instance internally. We can see the same in source code itself.

## Calling map.put() method:

Note that **Collections.synchronizedMap** returns an instance of type **SynchronizedMap.**so when you call **map.put(1,null);** due to polymorphism, SynchronizedMap.put() method is called which is synchronized on **mutex,**i.e. current instance of **SynchronizedMap.**

public V put(K key, V value) {

synchronized (mutex) {return m.put(key, value);}

}

**ConcurrentHashMap:** It allows concurrent access to the map. Part of the map called Segment (internal data structure) is only getting locked while adding or updating the map. So ConcurrentHashMap allows concurrent threads to read the value without locking at all. This data structure was introduced to improve performance.

A ConcurrentHashMap is divided into number of segments, and the example which I am explaining here used default as 32 on initialization.

A ConcurrentHashMap has internal final class called Segment so we can say that ConcurrentHashMap is internally divided in segments of size 32, so at max 32 threads can work at a time. It means each thread can work on a each segment during high concurrency and atmost 32 threads can operate at max which simply maintains 32 locks to guard each bucket of the ConcurrentHashMap.

**Synchronized vs Concurrent Collections**  
Though both Synchronized and Concurrent Collection classes provide thread-safety, the differences between them comes  in**performance**, **scalability** and how they achieve thread-safety. Synchronized collections like synchronized HashMap, Hashtable, HashSet, Vector, and synchronized ArrayList are much slower than their concurrent counterparts e.g. ConcurrentHashMap, CopyOnWriteArrayList, and CopyOnWriteHashSet. Main reason for this slowness is **locking;** synchronized collections locks the whole collection e.g. whole Map or List while concurrent collection never locks the whole Map or List.

**Multi Threading**

### How can we make sure main() is the last thread to finish in Java Program?

We can use Thread join() method to make sure all the threads created by the program is dead before finishing the main function.

### Why wait(), notify() and notifyAll() methods have to be called from synchronized method or block?

When a Thread calls wait() on any Object, it must have the monitor on the Object that it will leave and goes in wait state until any other thread call notify() on this Object. Similarly when a thread calls notify() on any Object, it leaves the monitor on the Object and other waiting threads can get the monitor on the Object. Since all these methods require Thread to have the Object monitor, that can be achieved only by synchronization, they need to be called from synchronized method or block.

### Why Thread sleep() and yield() methods are static?

Thread sleep() and yield() methods work on the currently executing thread. So there is no point in invoking these methods on some other threads that are in wait state. That’s why these methods are made static so that when this method is called statically,

### What is ThreadLocal?

Java ThreadLocal is used to create thread-local variables. We know that all threads of an Object share it’s variables, so if the variable is not thread safe, we can use synchronization but if we want to avoid synchronization, we can use ThreadLocal variables.  
Every thread has it’s own ThreadLocal variable and they can use it’s get() and set() methods to get the default value or change it’s value local to Thread.

### What is BlockingQueue?

java.util.concurrent.BlockingQueue is a Queue that supports operations that wait for the queue to become non-empty when retrieving and removing an element, and wait for space to become available in the queue when adding an element.

### What is Callable and Future?

### What does join() method?

The join() method waits for a thread to die. In other words, it causes the currently running threads to stop executing until the thread it joins with completes its task.

**What is the difference between notify() and notifyAll()?**  
A) notify() wakes up the first thread that called wait() on the same object, whereas the notifyAll() method wakes up all the waiting threads.

**What is the difference between start and run method in Java Thread?**

**What is the difference between CountDownLatch and CyclicBarrier in Java?**

The key point to mention, while answering this question is that CountDownLatch is not reusable once the count reaches to zero, while CyclicBarrier can be reused even after the barrier is broken.

**What is the difference between submit() and execute() method of Executor and ExecutorService in Java?**  
The main difference between submit and execute method from ExecutorServiceinterface is that former return a result in the form of a Future object, while later doesn't return a result.

**What is atomic operation/classes and CAS**

**What are Reentrant Locks?**

The ReentrantLock class implements the Lock interface and provides synchronization to methods while accessing shared resources. The code which manipulates the shared resource is surrounded by calls to lock and unlock method. This gives a lock to the current working thread and blocks all other threads which are trying to take a lock on the shared resource.

ReentrantLock allow threads to enter into lock on a resource more than once. When the thread first enters into lock, a hold count is set to one. Before unlocking the thread can re-enter into lock again and every time hold count is incremented by one. For every unlock request, hold count is decremented by one and when hold count is 0, the resource is unlocked.

Reentrant Locks also offer a fairness parameter, by which the lock would abide by the order of the lock request i.e. after a thread unlocks the resource, the lock would go to the thread which has been waiting for the longest time. This fairness mode is set up by passing true to the constructor of the lock.

**What do we understand by fair locks?**

A fair lock takes the waiting time of the threads into account when choosing the next thread that passes the barrier to some exclusive resource. An example implementation of a fair lock is provided by the Java SDK: java.util.concurrent.locks.ReentrantLock. If the constructor with the boolean flag set to true is used, the ReentrantLock grants access to the longest-waiting thread.

**Which Java classes use the CAS operation?**

The SDK classes in the package java.util.concurrent.atomic like AtomicInteger or AtomicBoolean use internally the CAS operation to implement concurrent incrementation.

public class CounterAtomic {

privateAtomicLong counter = new AtomicLong();

public void increment() {

counter.incrementAndGet();

}

public long get() {

returncounter.get();

}

}

**What happens when you submit() a new task to an ExecutorService instance whose queue is already full?**

As the method signature of submit() indicates, the ExecutorService implementation is supposed to throw a RejectedExecutionException.

**Yield VS Sleep**

sleep() causes the thread to definitely stop executing for a given amount of time; if no other thread or process needs to be run, the CPU will be idle (and probably enter a power saving mode).

yield() basically means that the thread is not doing anything particularly important and if any other threads or processes need to be run, they should. Otherwise, the current thread will continue to run.

### ****What is a volatile field****

A read of a volatile variable is guaranteed to observe the last write to this variable, according to this order.

If you have a field that is accessed from multiple threads, with at least one thread writing to it, then you should consider making it volatile, or else there is a little guarantee to what a certain thread would read from this field.

### ****If two threads call a synchronized method on different object instances simultaneously, could one of these threads block? What if the method is static?****

If the method is an instance method, then the instance acts as a monitor for the method. Two threads calling the method on different instances acquire different monitors, so none of them gets blocked.

If the method is static, then the monitor is the Class object. For both threads, the monitor is the same, so one of them will probably block and wait for another to exit the synchronized method.

### What is Executors in java Executor Framework?

Executors is a factory that provides the methods to return ExecutorService, ScheduledExecutorService, ThreadFactory. Find some method details.

**newFixedThreadPool()**: It returns the pool with fixed number of size. We need to pass the number of threads to this method. If concurrently task are submitted more than the pool size, then rest of task need to wait in queue. It returns ExecutorService.

**newScheduledThreadPool**: This also creates a fixed size pool but it can schedule the thread to run after some defined delay. It is useful to schedule the task. It returns ScheduledExecutorService.

**newCachedThreadPool()**: There is no fixed size of this pool. Thread will be created at run time and if there is no task it will alive for 60 second and then die. For short lived threads this pool works good. It returns ExecutorService.

**Method Overloading Rules**

1. First and important rule to overload a method in java is to **change method signature**. Method signature is made of **number of arguments, type of arguments and order of arguments** if they are of different types.
2. Return type of method is never part of method signature, so only **changing the return type of method does not amount to method overloading**.
3. Thrown exceptions from methods are also not considered when overloading a method. So your overloaded method throws the same exception, a different exception or it simply does no throw any exception; **no effect at all on method loading**.

**Method Overriding Rules**

1. The method **argument list in overridden and overriding methods must be exactly same**If they don’t match, you will end up with an overloaded method.
2. The **return type of overriding method can be child class of return type declared in overridden method**.
3. Above all rules, **private, static and final methods can not be overridden** in java in any way. As simple as that !!
4. **Overriding method can not throw checked Exception higher in hierarchy** than thrown by overridden method
5. Also note that **overriding method can not reduce the access scope of overridden method**

**JAVA 8**

### forEach() method in Iterable interface

### default and static methods in Interfaces

### Functional Interfaces and Lambda Expressions

Functional interfaces are new concept introduced in Java 8. An interface with exactly one abstract method becomes Functional Interface. We don’t need to use @FunctionalInterface annotation to mark an interface as Functional Interface

### Java Stream API for Bulk Data Operations on Collections

Stream API will allow sequential as well as parallel execution.

### Java Time API

**Scenarios Based**

1. Several request are hitting your function, maintain a count in multi threading environment.
2. Method m1() calls m2(), has taken lock in m1(), will acquire new lock for m2()..??
3. What if several request hits your API.

**Database**

## How a database index work

So, what is an index? Well, an index is a data structure (most commonly a B- tree) that stores the values for a specific column in a table. An index is created on a column of a *table*.B- trees are the most commonly used data structures for indexes.

The reason B- trees are the most popular data structure for indexes is due to the fact that they are time efficient – because look-ups, deletions, and insertions can all be done in logarithmic time. And, another major reason B- trees are more commonly used is because the data that is stored inside the B- tree can be **sorted**.

B- trees are the most commonly used data structures for indexes. The reason B- trees are the most popular data structure for indexes is due to the fact that they are time efficient – because look-ups, deletions, and insertions can all be done in logarithmic time. And, another major reason B- trees are more commonly used is because the data that is stored inside the B- tree can be **sorted**.

## How does a database know when to use an index?

When a query like “SELECT \* FROM Employee WHERE Employee\_Name = ‘Jesus’ ” is run, the database will check to see if there is an index on the column(s) being queried. Assuming the Employee\_Name column does have an index created on it, the database will have to decide whether it actually makes sense to use the index to find the values being searched – because there are some scenarios where it is actually less efficient to use the database index, and more efficient just to scan the entire table.

## Can you force the database to use an index on a query?

Generally, you will not tell the database when to actually use an index – that decision will be made by the database itself.

## How to create an index in SQL:

Here’s what the actual SQL would look like to create an index on the Employee\_Name column from our example earlier:

CREATE INDEX name\_index

ON Employee (Employee\_Name)

## How to create a multi-column index in SQL:

We could also create an index on two of the columns in the Employee table , as shown in this SQL:

CREATE INDEX name\_index

ON Employee (Employee\_Name, Employee\_Age)

**Database Replication**

**1. IO thread**

This process called *IO thread* connects to a master, reads binary log events from the master as they come in and just copies them over to a local log file called **relay log**. That’s all.

Even though there’s only one thread reading binary log from the master and one writing relay log on the slave, very rarely copying of replication events is a slower element of the replication. There could be a network delay, causing a steady delay of few hundred milliseconds, but that’s about it.

**2. SQL thread**

The second process – *SQL thread* – reads events from a relay log stored locally on the replication slave (the file that was written by IO thread) and then applies them as fast as possible.

**Clustered Vs Non Clustered Index**

A clustered index alters the way that the rows are stored. When you create a clustered index on a column (or a number of columns), SQL server sorts the table’s rows by that column(s). It is like a dictionary, where all words are sorted in alphabetical order in the entire book.

A non-clustered index, on the other hand, does not alter the way the rows are stored in the table. It creates a completely different object within the table that contains the column(s) selected for indexing and a pointer back to the table’s rows containing the data.

**Miscellaneous**

**What are various types of Class loaders used by JVM ?**

Bootstrap - Loads JDK internal classes, java.\* packages.

Extensions - Loads jar files from JDK extensions directory - usually lib/ext directory of the JRE

System  - Loads classes from system classpath.

**JMS**

<https://dzone.com/articles/jms-activemq>

JMS short for Java Message Service provides a mechanism for integrating applications in a loosely coupled, flexible manner. JMS delivers data asynchronously across applications on a store and forward basis. Applications communicate through MOM (Message Oriented Middleware) which acts as an intermediary without communicating directly.

**JMS Architecture**

Main components of JMS are:

* JMS Provider: A messaging system that implements the JMS interfaces and provides administrative and control features
* Clients: Java applications that send or receive JMS messages. A message sender is called the Producer, and the recipient is called a Consumer
* Messages: Objects that communicate information between JMS clients
* Administered objects: Preconfigured JMS objects created by an administrator for the use of clients.

How Kafka works

**Hibernate**

**What is Java Persistence API (JPA)**

Hibernate provides reference implementation of Java Persistence API, that makes it a great choice as ORM tool with benefits of loose coupling.

**What are the advantages of Hibernate over JDBC?**

* Hibernate supports inheritance, associations and collections.
* Hibernate implicitly provides transaction management, in fact most of the queries can’t be executed outside transaction. In JDBC API, we need to write code for transaction management using commit and rollback.
* Hibernate supports caching that is better for performance, JDBC queries are not cached hence performance is low.
* Hibernate Query Language (HQL) is more object oriented and close to java programming language. For JDBC, we need to write native sql queries
* Hibernate supports JPA annotations, so code is independent of implementation and easily replaceable with other ORM tools. JDBC code is very tightly coupled with the application.

**SessionFactory (org.hibernate.SessionFactory):** SessionFactory is an immutable thread-safe cache of compiled mappings for a single database. We need to initialize SessionFactory once and then we can cache and reuse it. SessionFactory instance is used to get the Session objects for database operations.

**Session (org.hibernate.Session):** Session is a single-threaded, short-lived object representing a conversation between the application and the persistent store. It wraps JDBC java.sql.Connection and works as a factory for org.hibernate.Transaction. We should open session only when it’s required and close it as soon as we are done using it. Session object is the interface between java application code and hibernate framework and provide methods for CRUD operations.

1. **javax.persistence.Entity**: Used with model classes to specify that they are entity beans.
2. **javax.persistence.Table**: Used with entity beans to define the corresponding table name in database.
3. **javax.persistence.Access**: Used to define the access type, either field or property. Default value is field and if you want hibernate to use getter/setter methods then you need to set it to property.
4. **javax.persistence.Id**: Used to define the primary key in the entity bean.
5. **javax.persistence.EmbeddedId**: Used to define composite primary key in the entity bean.
6. **javax.persistence.Column**: Used to define the column name in database table.
7. **javax.persistence.GeneratedValue**: Used to define the strategy to be used for generation of primary key. Used in conjunction with javax.persistence.GenerationType enum.
8. **javax.persistence.OneToOne**: Used to define the one-to-one mapping between two entity beans. We have other similar annotations as OneToMany, ManyToOne and ManyToMany
9. **org.hibernate.annotations.Cascade**: Used to define the cascading between two entity beans, used with mappings. It works in conjunction with org.hibernate.annotations.CascadeType
10. **javax.persistence.PrimaryKeyJoinColumn**: Used to define the property for foreign key. Used with org.hibernate.annotations.GenericGenerator and org.hibernate.annotations.Parameter

### What is difference between openSession and getCurrentSession?

[Hibernate SessionFactory](https://www.journaldev.com/3522/hibernate-sessionfactory) getCurrentSession() method returns the session bound to the context. But for this to work, we need to configure it in hibernate configuration file. Since this session object belongs to the hibernate context, we don’t need to close it. Once the session factory is closed, this session object gets closed.

<property name="hibernate.current\_session\_context\_class">thread</property>

Hibernate SessionFactory openSession() method always opens a new session. We should close this session object once we are done with all the database operations. We should open a new session for each request in multi-threaded environment.

**What is difference between Hibernate Session get() and load() method?**

Hibernate session comes with different methods to load data from database. get and load are most used methods, at first look they seems similar but there are some differences between them.

1. get() loads the data as soon as it’s called whereas load() returns a proxy object and loads data only when it’s actually required, so load() is better because it support lazy loading.
2. Since load() throws exception when data is not found, we should use it only when we know data exists.
3. We should use get() when we want to make sure data exists in the database.

**What is hibernate caching? Explain Hibernate first level cache?**

As the name suggests, hibernate caches query data to make our application faster. Hibernate Cache can be very useful in gaining fast application performance if used correctly. The idea behind cache is to reduce the number of database queries, hence reducing the throughput time of the application.

Hibernate first level cache is associated with the Session object. Hibernate first level cache is enabled by default and there is no way to disable it.

**What are different states of an entity bean?**

An entity bean instance can exist is one of the three states.

1. **Transient**: When an object is never persisted or associated with any session, it’s in transient state. Transient instances may be made persistent by calling save(), persist() or saveOrUpdate(). Persistent instances may be made transient by calling delete().
2. **Persistent**: When an object is associated with a unique session, it’s in persistent state. Any instance returned by a get() or load() method is persistent.
3. **Detached**: When an object is previously persistent but not associated with any session, it’s in detached state. Detached instances may be made persistent by calling update(), saveOrUpdate(), lock() or replicate(). The state of a transient or detached instance may also be made persistent as a new persistent instance by calling merge().

**What is use of Hibernate Session merge() call?**

Hibernate merge can be used to update existing values, however this method create a copy from the passed entity object and return it. The returned object is part of persistent context and tracked for any changes, passed object is not tracked.

**Different types of cache in Hibernate.**

**What is the difference between session.save() and session.persist() method?**

|  |  |  |
| --- | --- | --- |
| **No.** | **save()** | **persist()** |
| 1) | returns the identifier (Serializable) of the instance. | return nothing because its return type is void. |
| 2) | Syn: public Serializable save(Object o) | Syn: public void persist(Object o) |

# Hibernate Inheritance Mapping Tutorial

1. [Table Per Hierarchy](https://www.javatpoint.com/hibernate-inheritance-mapping-tutorial)
2. [Table Per Concrete class](https://www.javatpoint.com/hibernate-inheritance-mapping-tutorial)
3. [Table Per Subclass](https://www.javatpoint.com/hibernate-inheritance-mapping-tutorial)

We can map the inheritance hierarchy classes with the table of the database. There are three inheritance mapping strategies defined in the hibernate:

1. Table Per Hierarchy
2. Table Per Concrete class
3. Table Per Subclass

#### Table Per Hierarchy

In table per hierarchy mapping, single table is required to map the whole hierarchy, an extra column (known as discriminator column) is added to identify the class. But nullable values are stored in the table .

#### Table Per Concrete class

In case of table per concrete class, tables are created as per class. But duplicate column is added in subclass tables.

#### Table Per Subclass

In this strategy, tables are created as per class but related by foreign key. So there are no duplicate columns.

**Hibernate 2nd level cache using ehcache/infinispan**

[**Transaction isolation levels relation with locks on table**](https://stackoverflow.com/questions/16162357/transaction-isolation-levels-relation-with-locks-on-table)

* READ UNCOMMITTED - no lock on table. You can read data in the table while writing on it. This means, A writes data (uncommited) and B can read this uncommited data and use it (for any purpose). If A executes a rollback, B still has read the data and used it. This is the fastest but most insecure way to work with data since can lead to data holes in not physically related tables (yes, two tables can be logically but not physically related in real world apps =\).
* READ COMMITTED - lock on committed data. You can read the data that was only commited. This means, A writes data and B can't read the data saved by A until A executes a commit. The problem here is that C can update data that was read and used on B and B client won't have the updated data.
* REPEATABLE READ - lock on block of sql(which is selected by using select query). This means, B reads the data under some condition i.e. WHERE aField > 10 AND aField < 20, A inserts data where aField value is between 10 and 20, then B reads the data again and get a different result.
* SERIALIZABLE - lock on full table(on which Select query is fired). This means, B reads the data and **no other transaction can modify the data** on the table. This is the most secure but slowest way to work with data. Also, since a simple read operation locks **the table**, this can lead to heavy problems on production: imagine that T table is an Invoice table, user X wants to know the invoices of the day and user Y wants to create a new invoice, so while X executes the read of the invoices, Y can't add a new invoice (and when it's about money, people get really mad, specially the bosses).

**Hibernate 5 changes**

## Support classes of the Date and Time API as BasicTypes

## Fetch multiple entities by their primary key

MultiIdentifierLoadAccess<Book> multi = session.byMultipleIds(Book.class);

List<Book> books = multi.multiLoad(1L, 2L, 3L);

## Requires Java 7

Annotations – Inverse, mappedby, cascade

**Spring**

**What are the types of Dependency Injection Spring supports?**

**Setter Injection**

Setter-based DI is realized by calling setter methods on the user’s beans after invoking a no-argument constructor or no-argument static factory method to instantiate their bean.

**Constructor Injection**

Constructor-based DI is realized by invoking a constructor with a number of arguments, each representing a collaborator.

**What is Spring IOC container?**

The Spring IOC creates the objects, wire them together, configure them, and manage their complete lifecycle from creation till destruction. The spring container uses dependency injection (DI) to manage the components that make up an application.

**What bean scopes does Spring support? Explain them.**

The Spring Framework supports following five scopes, three of which are available only if the users use a web-aware Application Context.

**Singleton:** This scopes the bean definition to a single instance per Spring IoC container.

**Prototype:** This scopes a single bean definition to have any number of object instances.

**Request:** This scopes a bean [definition to an HTTP request](https://intellipaat.com/interview-question/mysql-interview-questions/). Only valid in the context of a web-aware Spring ApplicationContext

**Session:** This scopes a bean definition to an HTTP session. Only valid in the context of a web-aware Spring ApplicationContext.

**Global-session:** This scopes a bean definition to a global HTTP session. Only valid in the context of a web-aware Spring ApplicationContext.

**Explain Bean lifecycle in Spring framework?**

Following is sequence of a bean lifecycle in Spring:

**Instantiate:** First the spring container finds the bean’s definition from the XML file and instantiates the bean.

**Populate properties:** Using the dependency injection, spring populates all of the properties as specified in the bean definition.

**Set Bean Name:** If the bean implements BeanNameAware interface, spring passes the bean’s id to setBeanName() method.

**Set Bean factory:** If Bean implements BeanFactoryAware interface, spring passes the beanfactory to setBeanFactory() method.

**Pre Initialization:** Also called post process of bean. If there are any bean BeanPostProcessors associated with the bean, Spring calls postProcesserBeforeInitialization() method.

**Initialize beans:** If the bean implements IntializingBean,its afterPropertySet() method is called. If the bean has init method declaration, the specified initialization method is called.

**Post Initialization:**– If there are any BeanPostProcessors associated with the bean, their postProcessAfterInitialization() methods will be called.

**Ready to use:** Now the bean is ready to use by the application

**Destroy:** If the bean implements DisposableBean , it will call the destroy() method

**What is Aspect, Advice, Pointcut, JointPoint and Advice Arguments in AOP?**

**Aspect**: Aspect is a class that implements cross-cutting concerns, such as transaction management.

**Advice**: Advice is the action taken for a particular join point. In terms of programming, they are methods that gets executed when a specific join point with matching pointcut is reached in the application.

**Pointcut**: Pointcut are regular expressions that is matched with join points to determine whether advice needs to be executed or not.

**Join Point**: A join point is the specific point in the application such as method execution, exception handling, changing object variable values etc. In Spring AOP a join points is always the execution of a method.

One of the shortcoming of Spring AOP is that it can be applied only to the beans created through Spring Context.

**Question** m1() calls M2(), will AOP execute in both

**Question**: Java Singleton vs Spring Singleton

**@Component, @Controller, @Repository & @Service annotations in Spring?**

**@Component** is used to indicate that a class is a component. These classes are used for auto detection and configured as bean, when annotation based configurations are used.

**@Controller** is a specific type of component, used in MVC applications and mostly used with RequestMapping annotation.

**@Repository** annotation is used to indicate that a component is used as repository and a mechanism to store/retrieve/search data. We can apply this annotation with DAO pattern implementation classes.

**@Service** is used to indicate that a class is a Service. Usually the business facade classes that provide some services are annotated with this.

**Global Exception Handler** – Exception Handling is a cross-cutting concern and Spring provides @ControllerAdvice annotation that we can use with any class to define our global exception handler.

**@RestController**, a specialized version of the controller which is a convenience annotation that does nothing more than add the @Controller and @ResponseBody annotations. By annotating the controller class with @RestController annotation, you no longer need to add @ResponseBody to all the request mapping methods. The @ResponseBody annotation is active by default.

**Spring AOP proxy**

Spring AOP uses either JDK dynamic proxies or CGLIB to create the proxy for a given target object.

If the target object to be proxied **implements at least one interface then a JDK dynamic proxy** will be used. All of the interfaces implemented by the target type will be proxied. If the target object does not implement any interfaces then a CGLIB proxy will be created.

Spring Hibernate Integration

<bean id="myEmf" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">

      <property name="dataSource" ref="dataSource" />

      <property name="packagesToScan" value="org.baeldung.persistence.model" />

      <property name="jpaVendorAdapter">

         <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter" />

      </property>

      <property name="jpaProperties">

         <props>

            <prop key="hibernate.hbm2ddl.auto">create-drop</prop>

            <prop key="hibernate.dialect">org.hibernate.dialect.MySQL5Dialect</prop>

         </props>

      </property>

   </bean>

   <bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">

      <property name="driverClassName" value="com.mysql.cj.jdbc.Driver" />

      <property name="url" value="jdbc:<mysql://localhost:3306/spring_jpa>" />

      <property name="username" value="tutorialuser" />

      <property name="password" value="tutorialmy5ql" />

   </bean>

   <bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager">

      <property name="entityManagerFactory" ref="myEmf" />

   </bean>

   <tx:annotation-driven />

**JSP**

**What are the JSP lifecycle phases?**

1. **Translation** – JSP container checks the JSP page code and parse it to generate the servlet source code. For example in Tomcat you will find generated servlet class files at **TOMCAT/work/Catalina/localhost/WEBAPP/org/apache/jsp** directory. If the JSP page name is home.jsp, usually the generated servlet class name is home\_jsp and file name is home\_jsp.java
2. **Compilation** – JSP container compiles the jsp class source code and produce class file in this phase.
3. **Class Loading** – Container loads the class into memory in this phase.
4. **Instantiation** – Container invokes the no-args constructor of generated class to load it into memory and instantiate it.
5. **Initialization** – Container invokes the init method of JSP class object and initializes the servlet config with init params configured in deployment descriptor. After this phase, JSP is ready to handle client requests. Usually from translation to initialization of JSP happens when first request for JSP comes but we can configure it to be loaded and initialized at the time of deployment like servlets using load-on-startup element.
6. **Request Processing** – This is the longest lifecycle of JSP page and JSP page processes the client requests. The processing is multi-threaded and similar to servlets and for every request a new thread is spawned and ServletRequest and ServletResponse object is created and JSP service method is invoked.
7. **Destroy** – This is the last phase of JSP lifecycle where JSP class is unloaded from memory. Usually it happens when application is undeployed or the server is shut down.

**What are JSP lifecycle methods**

1. **jspInit()**: This method is declared in JspPage and it’s implemented by JSP container implementations. This method is called once in the JSP lifecycle to initialize it with config params configured in deployment descriptor. We can override this method using JSP declaration scripting element to initialize any resources that we want to use in JSP page.
2. **\_jspService()**: This is the JSP method that gets invoked by JSP container for each client request by passing request and response object. Notice that method name starts with underscore to distinguish it from other lifecycle methods because we can’t override this method. All the JSP code goes inside this method and it’s overridden by default. We should not try to override it using JSP declaration scripting element. This method is defined in HttpJspPage interface.
3. **jspDestroy()**: This method is called by container when JSP is unloaded from memory such as shutting down application or container. This method is called only once in JSP lifecycle and we should override this method to release any resources created in JSP init method.

**Which JSP lifecycle methods can be overridden?**

We can override jspInit() and jspDestroy() methods using JSP declaration scripting element.

**What are JSP implicit objects?**

 out Object

 request Object

 response Object

 config Object

 application Object

 session Object

 pageContext Object

 page Object

 exception Object

## 1) The include directive:

<%@ include file="header.html" %>

**Static**: adds the content from the value of the file attribute to the current page **at translation time**. The directive was originally intended for static layout templates, like HTML headers.

## 2) The <jsp:include> standard action:

<jsp:include page="header.jsp" />

**Dynamic**: adds the content from the value of the page attribute to the current page **at request time**. Was intended more for dynamic content coming from JSPs.

**Servlets**

**When servlet object is created?**

At the time of first request.

**Who is responsible to create the object of servlet?**

The web container or servlet container.

**What is difference between GenericServlet and HttpServlet?**

The GenericServlet is protocol independent whereas HttpServlet is HTTP protocol specific. HttpServlet provides additional functionalities such as state management etc.

**Difference between forward() method and sendRedirect() method ?**

|  |  |
| --- | --- |
| **forward() method** | **sendRedirect() method** |
| 1) forward() sends the same request to another resource. | 1) sendRedirect() method sends new request always because it uses the URL bar of the browser. |
| 2) forward() method works at server side. | 2) sendRedirect() method works at client side. |

**Servlets are not thread safe, so in multi threaded environment, they should not access variables which are not thread safe.**

**We Can call destroy() from within the service(). It will do whatever logic you have in destroy() (cleanup, remove attributes, etc.) but it won't "unload" the servlet instance itself.**

**You do not manage the life cycle of servlets in the program; the servlet engine does.**

**Servers**

**CORS Error**