**HIBERNATE**

1. **Session Vs Session Factory**

SessionFactory is a factory class for Session objects.

* **Availability** :- It is available for the whole application while a session is only available for particular transaction.
* **Life** :- Life Session is short lived while SessionFactory objects are long lived.
* **Caching level** :- SessionFactory provides a second level cache and Session provides a first level cache.
* **Thread Safe** :- Session Factory thread safe while session is not thread safe. Factory is thread safe because its immutable class
* **Singleton Session Factory:-**

The main difference is that "There will be only one session factory object per hibernate client application. Because the implementation class of SessionFactory interface is singleton Java class".

A SessionFactory is usually only built once at startup.

A session obtains a database connection lazily

* **Weight**

**Session Factory heavyweight** object and can create during startup of application one session factory for the one database configuration.

**Session :-** Light weight create for the per transaction

**2. Transient , Persist , and Detached**

* **Transient –** 
  + An object is transient if it has just been instantiated using the new operator,
  + It is not associated with any of the Hibernate Session(s).
  + It has no persistent representation in the database and no identifier value has been assigned.
  + Transient instances will be destroyed by the garbage collector if the application does not hold a reference anymore.
  + Use the Hibernate Session to make an object persistent and let Hibernate take care of the SQL statements that need to be executed for this transition.
* **Persistent –**
  + A persistent instance has a representation in the database and an identifier value.
  + It might just have been saved or loaded; however, it is by definition in the scope of a Session.
  + Hibernate will detect any changes made to an object in persistent state and synchronize the state with the database when the unit of work completes.
  + Developers do not execute manual UPDATE or DELETE statements when an object should be made transient.
* **Detached –**
  + A detached instance is an object that has been persistent, but its Session has been closed.
  + The reference to the object is still valid, of course, and the detached instance might even be modified in this state.
  + A detached instance can be reattached to a new Session at a later point of time, making it persistent again.
  + This feature enables a programming model for long running units of work that require user think-time.
  + We call them application transactions, i.e., a unit of work from the point of view of the user.

**3. First Level Cache vs Second Level Cache**

* + **First Level Cache**:
    - 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.
    - However hibernate provides methods through which we can delete selected objects from the cache or clear the cache completely.
    - Any object cached in a session will not be visible to other sessions and when the session is closed, all the cached objects will also be lost.
  + **Second Level Cache**:
    - Hibernate Second Level cache is disabled by default but we can enable it through configuration.
    - Currently EHCache and Infinispan provides implementation for Hibernate Second level cache and we can use them. We will look into this in the next tutorial for hibernate caching.
  + **Query Cache**:
    - Hibernate can also cache result set of a query.
    - Hibernate Query Cache doesn’t cache the state of the actual entities in the cache;
    - it caches only identifier values and results of value type. So it should always be used in conjunction with the second-level cache.
  + **Type of caches**
  + What is the default cache service of hibernate?
  + Hibernate supports multiple cache services like
  + EHCache
  + OSCache
  + SWARMCache and TreeCache and
  + default cache service of hibernate is EHCache.

**4. Criteria**

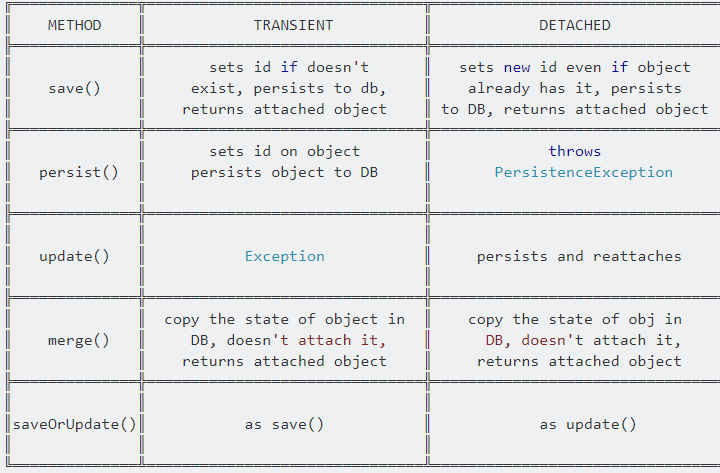
* Hibernate Criteria query is only used to fetch the results from the database using object oriented approach.

1. Hibernate Criteria API provides Projection that we can use for aggregate functions such as sum(), min(), max() etc.
2. Hibernate Criteria API can be used with ProjectionList to fetch selected columns only.
3. Criteria in Hibernate can be used for join queries by joining multiple tables, useful methods for Hibernate criteria join are createAlias(), setFetchMode() and setProjection()
4. Criteria in Hibernate API can be used for fetching results with conditions, useful methods are add() where we can add Restrictions.
5. Hibernate Criteria API provides addOrder() method that we can use for ordering the result

**5. Convert object detach to persistent**

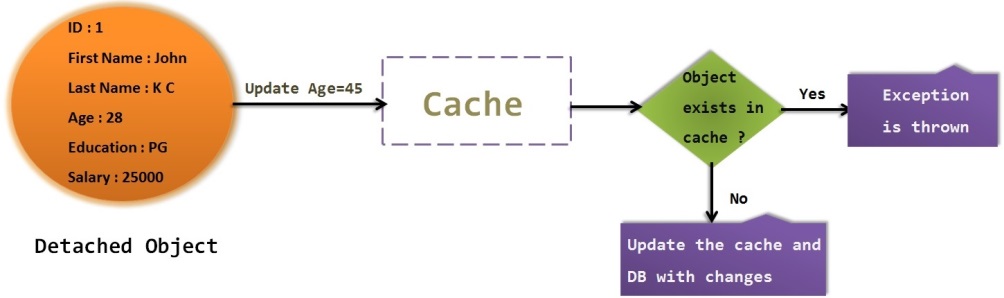
Update() and merge() methods in hibernate are used to convert the object which is in detached state into persistence state.

6. Difference Between Merge And Update Methods In Hibernate



When we call update() method, if the object already existed in cache update() method will throw exception whereas merge() method copies the changes in to cache.

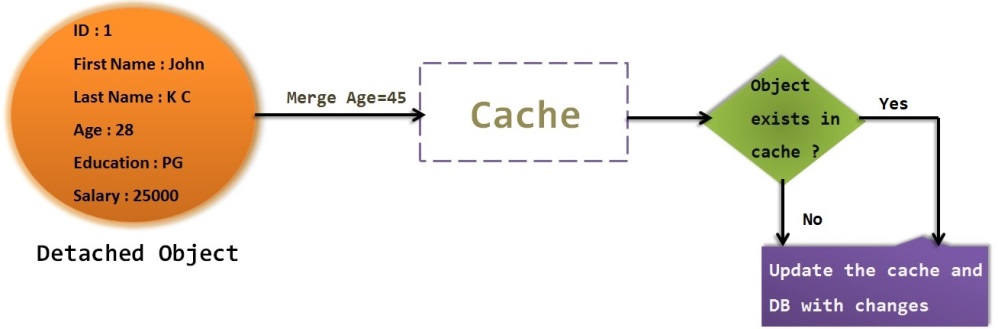
Lets see how update() method works with example

****

When we call update() method, ***if the same object is not exist in session cache then the update() method will update the object.***

When we call update() method, ***if the same object is already exist in session cache then the update() method throws an exception called “NonUniqueObjectException”.***

Lets see how merge() method works with example

****

Similar to update() method, ***merge() method also changes the state of detached state to persistent state.***

When we call merge() method, ***It first checks the same object exist in cache***

***If exist then it will update the cache with the changes, else if object is not exist in cache then it will load the values to cache.***

So in either way,**unlike update()** method, **merge() method will not throw any exception**.

7.What’s the difference between load() and get() method in hibernate?

**Load** :- Result in exception when not found data

**Get** :- Return Null

**Load** :- Use when sure the object exist

**Get** :- Use when not sure

**Load**

Get get will hit the database if object is not found in the cache and returned completely initialized object

**8.What’s the difference between session.save() and session.saveOrUpdate() methods in hibernate?**

* Sessionsave() method saves a record only if it’s unique with respect to its primary key and will fail to insert if primary key already exists in the table.
* saveOrUpdate() method inserts a new record if primary key is unique and will update an existing record if primary key exists in the table already.

**9.What’s transaction management in hibernate? How it works?**

Transaction management is the process of managing a set of statements or commands. In hibernate; transaction management is done by transaction interface as shown in below code:

**Session s = null;**

**Transaction tr = null;**

**try {**

**s = sessionFactory.openSession();**

**tr = s.beginTransaction();**

**doTheAction(s);**

**tr.commit();**

**} catch (RuntimeException exc) {**

**tr.rollback();**

**} finally {**

**s.close();**

**}**

**10 Save SaveorUpdate & Persis**t**What is the difference between save, persist and saveOrUpdate methods in Hibernate?**

* All three methods i.e. save(), saveOrUpdate() and persist() is used to save objects into database, but has subtle differences
* e.g. save() can only INSERT records but saveOrUpdate() can either INSERT or UPDATE records.
* The return type of save() is a Serializable object, while return type of persist() method is void.

**11. What is named SQL query in Hibernate?**

* This Hibernate Interview question is related to query functionality provided by Hibernate. Named queries are SQL queries which are defined in mapping document using <sql-query> tag and called using Session.getNamedQuery() method.
* Named query allows you to refer a particular query by the name you provided, by the way, you can define named query in hibernate either by using annotations or XML mapping file, as I said above. @NameQuery is used to define single named query and @NameQueries is used to define multiple named query in hibernate

@NameQueries

Session.getNamedQuery()

<sql-query> tag

**12. Call Procedure query Hibernate.**

* **Hibernate allows to express queries in native SQL format directly.**
* **Therefore, we can straightforwardly create a native SQL query, and use the *CALL* statement to call the *getAllFoos()* stored procedure:**

|  | **Query query = session.createSQLQuery("CALL GetAllFoos()").addEntity(Foo.class);**  **List<Foo> allFoos = query.list();**  **13. HQL (Hibernate Query Language)**   * HQL or Hibernate Query Language is the object-oriented query language of Hibernate Framework. * HQL is very similar to SQL except that we use Objects instead of table names, that makes it more close to object oriented programming. |
| --- | --- |

SessionFactory sessionFactory = HibernateUtil.getSessionFactory();  
 Session session = sessionFactory.getCurrentSession();  
 Transaction tx = session.beginTransaction();  
 Query query = session.createQuery("from Employee");  
 List<Employee> empList = query.list();

**SPRING**

**Dependency Injection**

* Dependency Injection (DI) is a design pattern.
* Removes the dependency from the programming code so that it can be easy to manage and test the application.
* Loosely coupled
* We are provide dependency through xml configuration file

**Spring framework provides two ways to inject dependency**

* By Constructor

<bean id="e" **class**="com.javatpoint.Employee">

<constructor-arg value="10" type="int"></constructor-arg>

</bean>

* By Setter method

<bean id="obj" **class**="com.javatpoint.Employee">

<property name="id"> <value>20</value></property>

</bean>

**What is Inversion of Control?**

**Inversion of Control is a principle in software engineering by which the control of objects or portions of a program is transferred to a container or framework.**

**It’s most often used in the context of object-oriented programming.**

**By contrast with traditional programming, in which our custom code makes calls to a library, IoC enables a framework to take control of the flow of a program and make calls to our custom code.**

**To enable this, frameworks use abstractions with additional behavior built in. If we want to add our own behavior, we need to extend the classes of the framework or plugin our own classes.**

**The advantages of this architecture are:**

* **decoupling the execution of a task from its implementation**
* **making it easier to switch between different implementations**
* **greater modularity of a program**
* **greater ease in testing a program by isolating a component or mocking its dependencies and allowing components to communicate through contracts**

**Inversion of Control can be achieved through various mechanisms such as: Strategy design pattern, Service Locator pattern, Factory pattern, and Dependency Injection (DI).**

**What bean scopes are supported by Spring and what do they mean? Which is used by default?**

* **singleton (used by default)**: This means a single instance per Spring container; not thread-safe
* **prototype**: This means any number of object instances.
* **request**: This scopes a bean definition to an HTTP request. 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.

**What is dependency injection (DI)? What are the types of DI?**

* **Dependency injection is the concept where you do not create your objects but describe how they should be created, and then expect pre-created objects to be passed in.**
* **Likewise, you don’t directly connect your components together but describe which components are needed with either a configuration file or an annotation. The Spring container is responsible for the rest.**
* **DI can be either constructor based or setter based. Constructor based DI is accomplished when the container invokes a class constructor with a number of arguments,**
* **each representing a dependency on other classes. Setter based dependency injection is accomplished when the container calls setter methods on a bean after instantiating it.**

**Describe the Spring bean lifecycle.**

**The lifecycle of a Spring bean consists the following steps:**

* **Instantiation**
* **Properties population**
* **Call of setBeanName() method of BeanNameAware**
* **Call of setBeanFactory() method of BeanFactoryAware**
* **Call of setApplicationContext() of ApplicationContextAware**
* **Pre-initialization with BeanPostProcessor**
* **Call of afterPropertiesSet() method of InitializingBean**
* **Custom init method**
* **Post-initialization with BeanPostProcessor**
* **Bean is ready to use**
* **Call of destroy() method of DisposableBean**
* **Custom destroy method**

Numbers 11-12 are actual for all scopes except prototype, since Spring does not manage the complete lifecycle of a prototype bean: the container instantiates, configures, and otherwise assembles a prototype object and hands it to the client with no further record of that prototype instance.

**Which steps of beans lifecycle can be overridden or controlled?**

* The init() method is called when the bean is loaded to the container via the init-method attribute in the xml configuration with the @PostConstruct annotation.
* The destroy() method is called when the bean is unloaded from the container, through the destroy-method attribute in the xml configuration with the @PreDestroy annotation.
* If a bean is a prototype-scoped, the client code must clean up objects and release expensive resources that the prototype beans are holding.
* To get the Spring container to release resources held by prototype-scoped beans, try using a custom BeanPostProcessor, which holds a reference to beans that need to be cleaned up.
* A developer can implement various interfaces to invoke specific behavior during a bean’s life cycle, such as InitializingBean and DisposableBean, as well as BeanNameAware,BeanFactoryAware and ApplicationContextAware.

**What is a Spring Bean Factory? What are some of the implementations available?**

* **A BeanFactory is the actual container which instantiates, configures and manages all Spring beans together with their dependencies.**
* **Bean factories are represented by the interface org.springframework.beans.factory.BeanFactory and its sub-interfaces including:**
  + **ApplicationContext**
  + **WebApplicationContext**
  + **AutowireCapableBeanFactory**
* **All of which are implemented with:**
  + **AnnotationConfigWebApplicationContext**
  + **XmlWebApplicationContext**
  + **ClassPathXmlApplicationContext**
  + **FileSystemXmlApplicationContext**
* **It’s important to note that implementations can correspond to multiple interfaces.**

What is a Spring Application Context? What are some example usages of one?

An ApplicationContext is an interface extending BeanFactory’s functionality. In addition to the BeanFactory’s methods, ApplicationContext provides the ability to:

Load file resources by extending the ResourcePatternResolver interface

Publish events to registered listeners (via the ApplicationEventPublisher interface)

Resolve messages supporting internationalization (with the MessageSource interface).

It’s read-only while the application is running.

The easiest way to create an ApplicationContent instance is:

ApplicationContext ctx = new FileSystemXmlApplicationContext("application.xml");

Loading resources is done with:

ctx.getResources(String locationPattern);

ctx.getResource(String location);

Publishing events is as simple as:

ctx.publishEvent(ApplicationEvent event);

ctx.publishEvent(Object event);

Internationalization support messages can be done by:

ctx.getMessage(String code, Object[] args, String defaultMessage, Locale locale);

ctx.getMessage(String code, Object[] args, Locale locale);

ctx.getMessage(MessageSourceResolvable resolvable, Locale locale);

In the context of Spring, what is a “stereotype”? What are the existing stereotypes and what is the difference between them?

Stereotype is a class-level annotation denoting the roles of types or methods in the overall architecture (at a conceptual level, rather than implementation). In Spring, these annotations live in the package org.springframework.stereotype.

Currently, this package has the following annotations:

@Component indicates that an annotated class is a “component”. Such classes are considered as candidates for auto-detection when using annotation-based configuration and classpath scanning.

@Controller indicates that an annotated class is a “Controller” (e.g. a web controller).

@Repository indicates that an annotated class is a “Repository”, originally defined by Domain-Driven Design (Evans, 2003) as “a mechanism for encapsulating storage, retrieval, and search behavior which emulates a collection of objects”.

@Service indicates that an annotated class is a “Service”, originally defined by Domain-Driven Design (Evans, 2003) as “an operation offered as an interface that stands alone in the model, with no encapsulated state.” May also indicate that a class is a Business Service Facade (in the Core J2EE patterns sense) or something similar.

These different types primarily allow a developer easily distinguish the purpose of the annotated classes. Starting with Spring 2.5, @Controller, @Repository and @Service serve as a specialization of @Component, allowing for implementation classes to be autodetected through classpath scanning.

**How do you load and inject properties into a Spring Bean?**

Let’s say we have a custom.properties file that defines a database connection timeout property called connection.timeout. To load this property into a Spring context, we need to define a propertyConfigurer bean:

<bean id="propertyConfigurer" class="org.springframework.context.support.PropertySourcesPlaceholderConfigurer">

<property name="location" value="custom.properties" />

</bean>

After that we can use Spring Expression Language to inject properties into other beans:

<bean class="com.toptal.spring.ConnectionFactory">

<property name="timeout" value="${connection.timeout}"/>

</bean>

The same is available in the annotation based configuration, like so:

@Value("${connection.timeout}")

private int timeout;

**What are the different ways to configure a class as Spring Bean?**

* **Java Based Configuration**

Any object can be put into Spring Context and be reused later as a usual bean.

ConfigurableApplicationContext context;

context.getBeanFactory().registerSingleton(name, obj);

* What is a Spring Application Context? What are some example usages of one?
* An ApplicationContext is an interface extending BeanFactory’s functionality. In addition to the BeanFactory’s methods, ApplicationContext provides the ability to:
* Load file resources by extending the ResourcePatternResolver interface
* Publish events to registered listeners (via the ApplicationEventPublisher interface)
* Resolve messages supporting internationalization (with the MessageSource interface).
* It’s read-only while the application is running.
* The easiest way to create an ApplicationContent instance is:
* ApplicationContext ctx = new FileSystemXmlApplicationContext("application.xml");
* Loading resources is done with:

ctx.getResources(String locationPattern);

ctx.getResource(String location);

* Publishing events is as simple as:

ctx.publishEvent(ApplicationEvent event);

ctx.publishEvent(Object event);

* Internationalization support messages can be done by:

ctx.getMessage(String code, Object[] args, String defaultMessage, Locale locale);

ctx.getMessage(String code, Object[] args, Locale locale);

ctx.getMessage(MessageSourceResolvable resolvable, Locale locale);

* In the context of Spring, what is a “stereotype”? What are the existing stereotypes and what is the difference between them?
* Stereotype is a class-level annotation denoting the roles of types or methods in the overall architecture (at a conceptual level, rather than implementation). In Spring, these annotations live in the package org.springframework.stereotype.
* Currently, this package has the following annotations:
* @Component indicates that an annotated class is a “component”. Such classes are considered as candidates for auto-detection when using annotation-based configuration and classpath scanning.
* @Controller indicates that an annotated class is a “Controller” (e.g. a web controller).
* @Repository indicates that an annotated class is a “Repository”, originally defined by Domain-Driven Design (Evans, 2003) as “a mechanism for encapsulating storage, retrieval, and search behavior which emulates a collection of objects”.
* @Service indicates that an annotated class is a “Service”, originally defined by Domain-Driven Design (Evans, 2003) as “an operation offered as an interface that stands alone in the model, with no encapsulated state.” May also indicate that a class is a Business Service Facade (in the Core J2EE patterns sense) or something similar.
* These different types primarily allow a developer easily distinguish the purpose of the annotated classes. Starting with Spring 2.5, @Controller, @Repository and @Service serve as a specialization of @Component, allowing for implementation classes to be autodetected through classpath scanning.
* How do you load and inject properties into a Spring Bean?
* Let’s say we have a custom.properties file that defines a database connection timeout property called connection.timeout. To load this property into a Spring context, we need to define a propertyConfigurer bean:
* <bean id="propertyConfigurer" class="org.springframework.context.support.PropertySourcesPlaceholderConfigurer">
* <property name="location" value="custom.properties" />
* </bean>
* After that we can use Spring Expression Language to inject properties into other beans:
* <bean class="com.toptal.spring.ConnectionFactory">
* <property name="timeout" value="${connection.timeout}"/>
* </bean>
* The same is available in the annotation based configuration, like so:
* @Value("${connection.timeout}")
* private int timeout;
* What are the different ways to configure a class as Spring Bean?
* Java Based Configuration
* Any object can be put into Spring Context and be reused later as a usual bean.
* ConfigurableApplicationContext context;
* context.getBeanFactory().registerSingleton(name, obj);
* Annotation Based Configuration
* A Spring Bean can be configured with the @Bean annotation, which is used together with @Configuration classes.
* @Configuration
* public class MyConfiguration {
* @Bean
* public MyService getService(){
* return new MyService();
* }
* }
* The annotations @Component, @Service, @Repository and @Controller can also be used with classes to configure them as Spring Beans. In this case, the base package location has to be provided to scan for these classes, like so:
* <context:component-scan base-package="com.toptal.spring" />
* Which dependency injection method is better: Constructor-based or setter-based?
* You can use both Constructor-based and Setter-based Dependency Injection. The best solution is using constructor arguments for mandatory dependencies and setters for optional dependencies.
* What is Bean wiring? How does autowiring work?
* Bean wiring is the process of injection Spring Bean dependencies while initializing. It’s usually best practice to wire all dependencies explicitly, (with XML configuration, for example), but Spring also supports autowiring with the @Autowired annotation. To enable this annotation we need to put the context:annotation-config element into the Spring configuration file. To avoid conflicts in bean mapping while autowiring, the bean name has to be provided with the @Qualifier annotation.
* There are different ways to autowire a Spring Bean:
* byName - to use this type setter method for dependency injection, the variable name should be the same in both the class where the dependency will be injected and in the Spring configuration file.
* byType - in order for this to function, there should be only one bean configured for that specific class.
* Via constructor - similar to byType, but type is applied to constructor arguments.
* Via autodetect - now obsolete, used in Spring 3.0 and earlier, this was used to autowire by constructor or byType.

A Spring Bean can be configured with the @Bean annotation, which is used together with @Configuration classes.

@Configuration

public class MyConfiguration {

@Bean

public MyService getService(){

return new MyService();

}

}

* **The annotations**

@Component,

@Service,

@Repository

@Controller can also be used with classes to configure them as Spring Beans.

In this case, the base package location has to be provided to scan for these classes, like so:

<context:component-scan base-package="com.toptal.spring" />

**Which dependency injection method is better: Constructor-based or setter-based?**

* You can use both Constructor-based and Setter-based Dependency Injection.
* The best solution is using constructor arguments for mandatory dependencies and setters for optional dependencies.

**What is Bean wiring? How does autowiring work?**

* Bean wiring is the process of injection Spring Bean dependencies while initializing.
* It’s usually best practice to wire all dependencies explicitly, (with XML configuration, for example), but Spring also supports autowiring with the @Autowired annotation.
* To enable this annotation we need to put the context:annotation-config element into the Spring configuration file.
* To avoid conflicts in bean mapping while autowiring, the bean name has to be provided with the @Qualifier annotation.

**There are different ways to autowire a Spring Bean:**

* **byName** - to use this type setter method for dependency injection, the variable name should be the same in both the class where the dependency will be injected and in the Spring configuration file.
* **byType** - in order for this to function, there should be only one bean configured for that specific class.
* **Via constructor** - similar to byType, but type is applied to constructor arguments.
* **Via autodetect -** now obsolete, used in Spring 3.0 and earlier, this was used to autowire by constructor or byType.

**Is Java platform independent?**

* Yes. Java is a platform independent language.
* We can write java code on one platform and run it on another platform.
* e.g. we can write and compile the code on windows and can run it on Linux or any other supported platform. This is one of the main features of java.

**What is javac ?**

* It produces the java byte code from \*.java file.
* It is the intermediate representation of your source code that contains instructions.

**What is class?**

* Class is nothing but a template that describes the data and behavior associated with instances of that class

**What is the base class of all classes?**

java.lang.Object

**What is Unicode?**

* Java uses Unicode to represent the characters.
* Unicode defines a fully international character set that can represent all of the characters found in human languages.

**What is Type casting in Java?**

* To create a conversion between two incompatible types,
* we must use a cast.
* There are two types of casting in java: automatic casting (done automatically) and explicit casting (done by programmer).

**Abstract class?**

An abstract class is a class which can’t be instantiated (we cannot create the object of abstract class), we can only extend such classes. It provides the generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details. We can achieve partial abstraction using abstract classes, to achieve full abstraction we use interfaces.

**Q) What is Interface in java?**

An interface is a collection of abstract methods. A class implements an interface, thereby inheriting the abstract methods of the interface. [Read more about interface here](https://beginnersbook.com/2013/05/java-interface/).

**Q) What is the difference between abstract class and interface?**

1) abstract class can have abstract and non-abstract methods. An interface can only have abstract methods.

2) An abstract class can have static methods but an interface cannot have static methods.

3) abstract class can have constructors but an interface cannot have constructors.

What is Collection ? What is a Collections Framework ? What are the benefits of Java Collections Framework ?

Collection : A collection (also called as container) is an object that groups multiple elements into a single unit.

Collections Framework : Collections framework provides unified architecture for manipulating and representing collections.

Benefits of Collections Framework :

1. Improves program quality and speed

2. Increases the chances of reusability of software

3. Decreases programming effort.

Root Element in collection Hierarchy

Collection interface

What is the difference between Collection and Collections ?

Collection is an interface while Collections is a java class

Which collection classes are synchronized or thread-safe ?

Stack, Properties , Vector and Hashtable

The list of core collection interfaces are : just mention the important ones

Important : Collection , Set , Queue , List , Map

Other interface also in the list : SortedSet, SortedMap , Deque, ListIterator etc

What is the difference between List and Set ?

Set contain only unique elements while List can contain duplicate elements.

Set is unordered while List is ordered . List maintains the order in which the objects are added .

What is the difference between Map and Set ?

Map object has unique keys each containing some value, while Set contain only unique values.

Class implementing List interface : ArrayList , Vector , LinkedList ,

Class implementing Set interface : HashSet , TreeSet

Iterator is an interface . It is found in java.util package. It provides methods to iterate over any Collection.

Q10 What is the difference between Iterator and Enumeration ?

The main difference between Iterator and Enumeration is that Iterator has remove() method while Enumeration doesn't.

Hence , using Iterator we can manipulate objects by adding and removing the objects from the collections.

Enumeration behaves like a read only interface as it can only traverse the objects and fetch it

Which methods you need to override to use any object as key in HashMap ?

To use any object as key in HashMap , it needs to implement equals() and hashCode() method .

How to reverse the List in Collections ?

There is a built in reverse method in Collections class . reverse(List list) accepts list as parameter.

Convert array of string String[] wordArray = {"Love Yourself" , "Alive is Awesome" , "Be in present"};

List wordList = Arrays.asList(wordArray);

Difference between ArrayList and Vector

Synchronized :- Vector synchronized

ArrayList NotSync

Speed :- Vector slow

Arraylist Fast

Difference HashMap & HashSet

Null :- HashMap allows one null key and any number of null values while Hashtable does not allow null keys and null values.

Synchronized :- HashMap synchronized and Thread Safe

HashSet NotSync and Not Thread safe

Iterating the values: Hashmap object values are iterated by using iterator .

HashTable is the only class other than vector which uses enumerator to iterate the values of HashTable object.

Performance : Hashmap is much faster and uses less memory than Hashtable as former is unsynchronized

Queue :- Peek() , Poll() and remove()

Both poll() and remove() method is used to remove head object of the Queue.

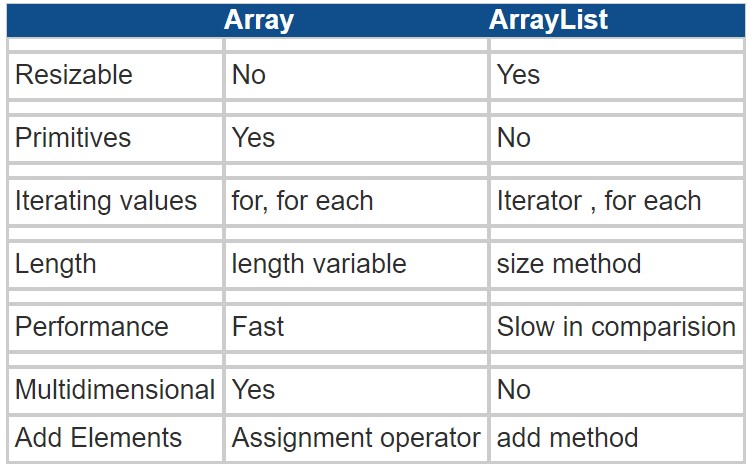
Return Type

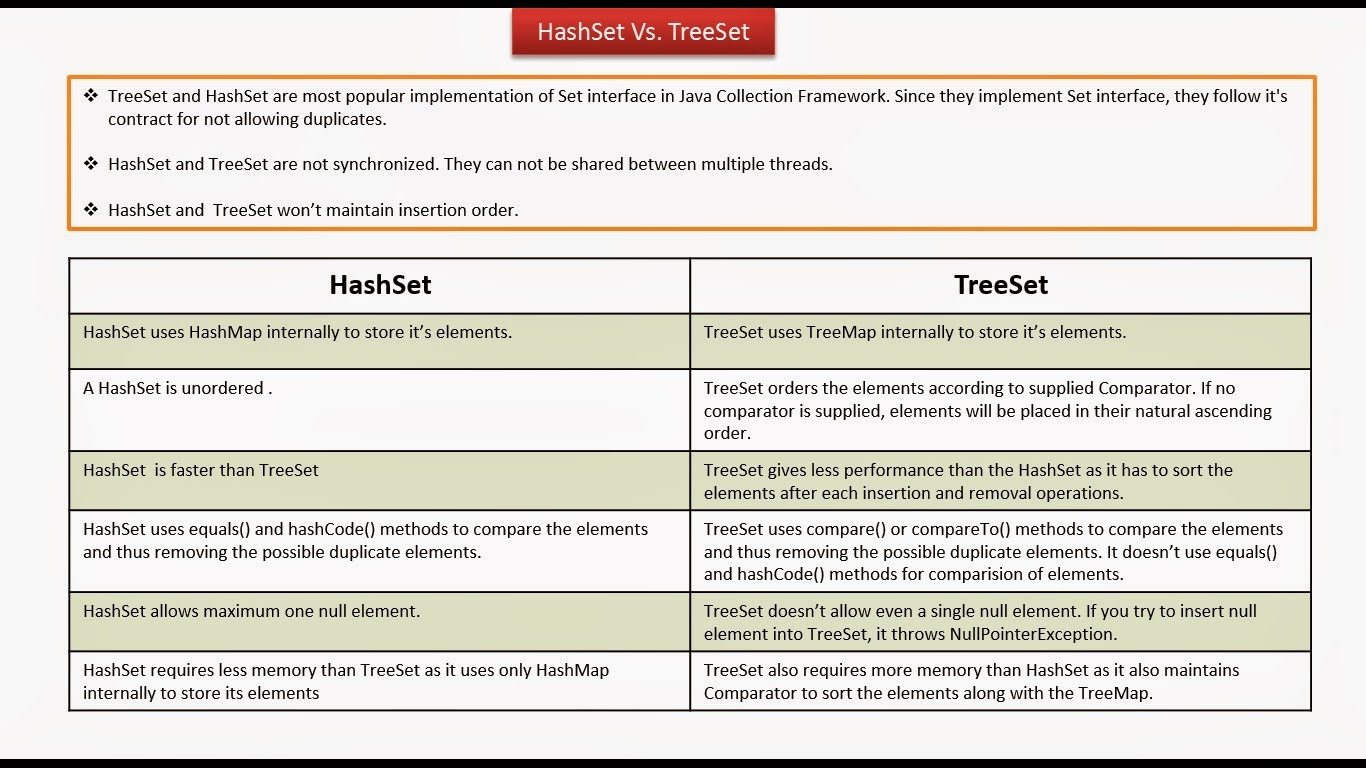
queque empty :-

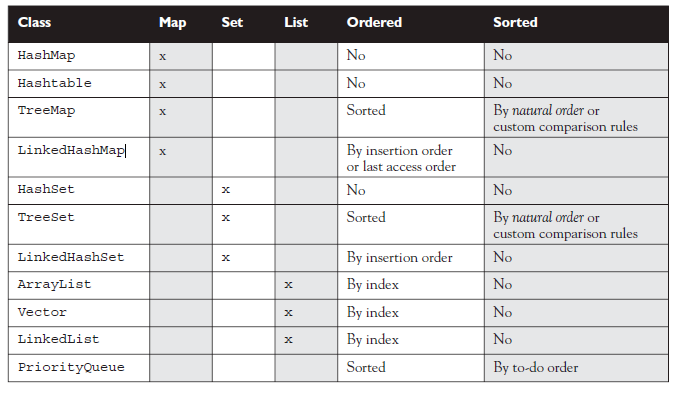
Poll() ==> Null

remove() ==> remove() method will throw NoSuchElementException .

peek() method==> retrieves but does not remove the head of the Queue. If queue is empty then peek() method also returns null.







How to make Map Ordered

* LinkedHashMap

Iteration order for above implementations:

* HashSet - undefined
* HashMap - undefined
* LinkedHashSet - insertion order
* LinkedHashMap - insertion order of keys (by default), or 'access order'
* ArrayList - insertion order
* LinkedList - insertion order
* TreeSet - ascending order, according to Comparable / Comparator
* TreeMap - ascending order of keys, according to Comparable / Comparator
* Collections.synchronizedList(new ArrayList<YourClassNameHere>())

**Principal features of non-primary implementations:**

* HashMap has slightly better performance than LinkedHashMap, but its iteration order is undefined
* HashSet has slightly better performance than LinkedHashSet, but its iteration order is undefined
* TreeSet is ordered and sorted, but slower
* TreeMap is ordered and sorted, but slower
* LinkedList has fast adding to the start of the list, and fast deletion from the interior via iteration

### **How HashMap works in Java?**

* HashMap stores key-value pair in Map.Entry static nested class implementation.
* HashMap works on hashing algorithm and uses hashCode() and equals() method in put and get methods.
* When we call put method by passing key-value pair, HashMap uses Key hashCode() with hashing to find out the index to store the key-value pair
* The Entry is stored in the LinkedList, so if there are already existing entry, it uses equals() method to check if the passed key already exists, if yes it overwrites the value else it creates a new entry and store this key-value Entry.
* When we call get method by passing Key, again it uses the hashCode() to find the index in the array and then use equals() method to find the correct Entry and return it’s value. Below image will explain these detail clearly.
* The other important things to know about HashMap are capacity, load factor, threshold resizing. HashMap initial default capacity is **16** and load factor is 0.75. Threshold is capacity multiplied by load factor and whenever we try to add an entry,
* if map size is greater than threshold, HashMap rehashes the contents of map into a new array with a larger capacity. The capacity is always power of 2,
* so if you know that you need to store a large number of key-value pairs, for example in caching data from database, it’s good idea to initialize the HashMap with correct capacity and load factor.

### **What is the importance of hashCode() and equals() methods?**

HashMap uses Key object hashCode() and equals() method to determine the index to put the key-value pair. These methods are also used when we try to get value from HashMap. If these methods are not implemented correctly, two different Key’s might produce same hashCode() and equals() output and in that case rather than storing it at different location, HashMap will consider them same and overwrite them.

Similarly all the collection classes that doesn’t store duplicate data use hashCode() and equals() to find duplicates, so it’s very important to implement them correctly. The implementation of equals() and hashCode() should follow these rules.

* If o1.equals(o2), then o1.hashCode() == o2.hashCode()should always be true.
* If o1.hashCode() == o2.hashCode is true, it doesn’t mean that o1.equals(o2) will be true.

https://www.journaldev.com/1330/java-collections-interview-questions-and-answers