Difference between Executor, ExecutorService and Executers class in Java

The main difference between `Executor`, `ExecutorService`, and `Executors` class is that Executor is the core interface which is an abstraction for parallel execution. It separates task from execution, this is different from `java.lang.Thread` class which combines both task and its execution. You can read the difference between Thread and Executor to learn more differences between these two key classes of Java.
On the other hand, `ExecutorService` is an extension of Executor interface and provides a facility for returning a Future object and terminate, or shut down the thread pool. Once the shutdown is called, the thread pool will not accept new task but complete any pending task. It also provides a submit() method which extends Executor.execute() method and returns a Future.

The Future object provides the facility of asynchronous execution, which means you don't need to wait until the execution finishes, you can just submit the task and go around, come back and check if Future object has the result, if execution is completed then it would have result which you can access by using the `Future.get()` method. Just remember that this method is a blocking method i.e. it will wait until execution finish and the result is available if it's not finished already.

By using the Future object returned by `ExecutorService.submit()` method, you can also cancel the execution if you are not interested anymore. It provides cancel() method to cancel any pending execution.

Third one `Executors` is a utility class similar to `Collections`, which provides factory methods to create different types of thread pools e.g. fixed and cached thread pools. Let's see some more difference between these three classes.

Executor vs ExecutorService vs Executors in Java

As I told, all three classes are part of Java 1.5 Executor framework and it's very important for a Java programmer to learn and understand about these classes to make effective use of different types of thread pools provided by Java. Let's see some key differences between Executor, ExecutorService, and Executors in Java to understand them better:

1) One of the key difference between Executor and ExecutorService interface is that former is a parent interface while ExecutorService extends Executor i.e. it's a sub-interface of Executor.

2) Another important difference between ExecutorService and Executor is that Executor defines execute() method which accepts an object of the Runnable interface, while submit() method can accept objects of both Runnable and Callable interfaces.

3) The third difference between Executor and ExecutorService interface is that execute() method doesn't return any result, its return type is void but submit() method returns the result of computation via a Future object. This is also the key difference between submit() and execute() method, which is one of the frequently asked Java concurrency interview questions.

4) The fourth difference between ExecutorService and Executor interface is that apart from allowing a client to submit a task, ExecutorService also provides methods to control the thread pool e.g. terminate the thread pool by calling the shutDown() method. You should also read "Java Concurrency in Practice" to learn more about the graceful shutdown of a thread-pool and how to handle pending tasks.

5) Executors class provides factory methods to create different kinds of thread pools e.g. `newSingleThreadExecutor()` creates a thread pool of just one thread, `newFixedThreadPool(int numOfThreads)` creates a thread pool of fixed number of threads and `newCachedThreadPool()` creates new threads when needed but reuse the existing threads if they are available.

String Deduplication of G1 Garbage collector to Save Memory from Duplicate String in Java 8

You might not be aware that Java 8 update 20 has introduced a new feature called **"String deduplication"** which can be used to save memory from duplicate String object in Java application, which can improve the performance of your Java application and prevent `java.lang.OutOfMemoryError` if your application makes heavy use of String. If you have profiled a Java application to check which object is taking the bulk of memory, you will often find `char[]` object at the top of the list, which is nothing but internal character array used by String object. Some of the tools and profilers might show this as `java.lang.String[]` as well e.g. Java Flight Recorder, but they are essentially pointing to the same problem i.e. a major portion of memory is occupied with String objects.

Since from Java 7 onward, String has stopped sharing character array with sub-strings, the memory occupied by String object has gone higher, which had made the problem even worse. If you remember, earlier both substring and String share same character objects (see how Substring works in Java), which was actually a bug that had potential to cause a serious memory leak. The bug was fixed in JDK 7, but it created this new problem.

The **String deduplication** is trying to bridge that gap. It reduces the memory footprint of String object on the Java Heap space by taking advantage of the fact that many String objects are identical. Instead of each String object pointing to their own character array, identical String object can point to the same character array.

Btw, this is not exactly same as it was before Java 7 update 6, where substring also points to the same character array, but can greatly reduce memory occupied by duplicate String in JVM. Anyway, In this article, you will see how you can enable this feature in Java 8 to reduce memory consumed by duplicate String objects.

How to enable String deduplication in Java 8

String deduplication is *not enabled* by default in Java 8 JVM. You can enable String deduplication feature by using -XX:+UseStringDeduplication option. Unfortunately, String deduplication is only available for the G1 garbage collector, so if you are not using G1 GC then you cannot use the String deduplication feature. It means just providing -XX:+UseStringDeduplication will not work, you also need to turn on G1 garbage collector using -XX:+UseG1GC option.

String deduplication also doesn't consider relatively young String for processing. The minimal age of processed String is controlled by -XX:StringDeduplicationAgeThreshold=3 option. The default value of this parameter is 3.

Now, you might be thinking that how does this method compare with the traditional way to reduce memory due to duplicate String e.g. by using intern() method of java.lang.String class? Well, this approach has an advantage because you don't need to write a single line of code. Just enable this feature using JVM parameters and you are done.

If you have ever optimized your code by using `String.intern()` method then you know that it's not easy. It not only compromise readability by adding additional lines of code without adding any functionality but also increase the size of the code. Btw, if you are interested in learning more about G1 Garbage collector, I suggest reading Java Performance Companion by Charlie Hunt, which covers some good information about G1 Garbage Collector.

Important points

Here are some of the important points about String deduplication feature of Java 8:

1) This option is only available from Java 8 Update 20 JDK release.

2) This feature will only work along with G1 garbage collector, it will not work with other garbage collectors e.g. Concurrent Mark Sweep GC.

3) You need to provide both -XX:+UseG1GC and -XX:+StringDeduplication JVM options to enable this feature, first one will enable the G1 garbage collector and the second one will enable the String deduplication feature within G1 GC.

4) You can optionally use -XX:+PrintStringDeduplicationStatistics JVM option to analyze what is happening through the command-line.

5) Not every String is eligible for deduplication, especially young String objects are not visible, but you can control this by using -XX:StringDeduplicationAgeThreshold=3 option to change when Strings become eligible for deduplication.

6) It is observed in general this feature may decrease heap usage by about 10%, which is very good, considering you don't have to do any coding or refactoring.

7) String deduplication runs as a background task without stopping your application.

That's all about **how to use enable String deduplication in Java 8** to reduce memory consumed by duplicate String objects. This is one of the useful features to know about it but unfortunately, it is only available for G1 Garbage Collector. You also need Java 8 Update 20 to use enable this option. Hopefully, in Java 9, when G1 Garbage collector will become the default collector, it can use this feature to further improve performance. If we are lucky, we may also see this feature extended for other major garabage collector like Concurrent Mark Sweep Garbage collector.

5 Difference between Constructor and Static Factory method in Java- Pros and Cons

The common way to create objects in Java is by using public constructors. A class provides public constructor e.g. `java.lang.String` so anyone can create an instance of String class to use in their application, but, there is another technique which can be used to create objects in Java and every experienced Java programmer should know about it. A class can provide a public static factory method which can return an instance of the class e.g. `HashMap.newInstance()`. The factory method is a smart way to create objects in Java and provides several advantages over the traditional approach of creating objects using constructors in Java. It can also improve the quality of code by making the code more readable, less coupled, and improves performance by caching.

Let's analyze some difference between constructor and factory method to find out which one is better for creating objects. But before that, let's find learn what is factory method and what are shortcomings of constructor in Java? A factory method is nothing but a static method which returns an object of the class. The most common example of factory method is `getInstance()` method of any Singleton class, which many of you have already used.

The main problem with a constructor is that by definition they don't have names. As such, a class can have only one constructor with given signature, which limits your capability to provide a more useful constructor. For example, if you already have a constructor `IdGenerator(int max)` which generates integer Id up to that number and you want to add `IdGenerator(int min)` you cannot do that, the compiler will throw an error.

Constructor vs Factory Method in Java

The problem we saw in the previous paragraph is just one of the many problems you face when you decide to provide a public constructor for creating an instance of the class, you will learn more of such shortcomings of constructor in Java as we discuss actual differences one by one. Thankfully, Factory methods address many limitations of constructors in Java and help to write cleaner code.

Here is my list of some key differences between constructor and static factory method in Java. All these differences stem from the shortcoming of constructors and explain how static factory methods solves those problems. They will also explain relative pros and cons of different ways of creating objects in Java.

Readable Names

One of the serious limitation of the constructor is that you cannot give it an explicit name, the name must be same as the name of the class. If your class return two different types of object for a different purpose, you can use factory methods with more readable names.

A good example of this concept is `java.text.NumberFormat` class in JDK, which provides different factory methods to returns different objects e.g. `getCurrencyInstance()` to return an instance of `NumberFormat` which can format currency, `getPercentInstance()` to return a percentage format, and `getNumberInstance()` to return a general purpose number formatting.

If you have used new `NumberFormat()`, it would have been difficult to know that which kind of `NumberFormat` instance would have returned.

Polymorphism
Another serious limitation of a constructor is that it always return the same type of object. You cannot vary the type of constructed object, it must be same as the name of the contractor. But the factory method can return an object of different subclasses e.g. in above example, if you call `getNumberInstance()` then it return an object of `DecimalFormat` class.

The Polymorphism also allows static factory methods to return instances of non-public classes e.g. `RegularEnumSet` and `JumboEnumSet` in the case of `EnumSet` interface. When you call the `EnumSet.of()` method, which is a static factory method, it can return an instance of either of this class depending upon the number of enum constants in the Enum class you have provided.

This means, most suitable class is used for the job. This improves the performance of your Java code as well. You can further read, Java Performance The Definitive Guide By Scott Oaks to learn more about such specialized trick to improve performance.

Coupling
Factory methods promote the idea of coding using Interface then implementation which results in more flexible code, but constructor ties your code to a particular implementation.

On the other hand by using constructor you tightly any client (who uses that class) to the class itself. Any change in class e.g. introducing a new constructor or new implementation will require change almost everywhere.

Hence, it's very difficult to change and unit test the code which uses constructors to create objects all over.

It's also one of the best practice in Java to make the constructor private to ensure that objects are only created using public static factory methods provided. The standard JDK API uses this technique with many new classes e.g. EnumSet.

If you have ever used EnumSet then you might know that you cannot create instances of this class using a constructor, you must use the static factory method, EnumSet.of() to create an instance. This allows JDK to use choose the more relevant implementation of EnumSet depending upon the key sizes of provided Enum object. See RegularEnumSet vs JumboEnumSet to learn more about how EnumSet works in Java.
Type Inference
Until Java 7, the constructor doesn't provide automatic type inference, which means you need to declare generic types on both left and right side of variable declaration as shown below.

This results in unreadable and cluttered code. Java 7 improved this by introducing the diamond operator, which helps to infer types, but factory method has this type inference right from Java 1.5. Google Guava has created several static utility classes with lots of factory methods to take advantage of improved type inference provided by factory methods. here are a couple of examples.

Caching
A constructor always creates a new object in heap. It's not possible to return a cached instance of a class from Constructor. On other hand, Factory methods can take advantage of caching. It's, in fact, a common practice to return the same instance of Immutable classes from factory method instead of always creating a new one.

For example `Integer.valueOf()`, `Long.valueOf()` are factory methods to create instance of `Integer` and `Long` respectively and return *cached* value in range of -128 to 127.

They are also used during auto-boxing. Since we mostly used values in that range, it greatly improves performance by reducing memory consumption and pressure on garbage collection. You can read Effective Java Item 1 to learn more about how caching of `valueOf()` method works. The item is actually also the best source to learn why static factory methods are better than constructor for creating objects.

Disadvantages of Static Factory methods

Like all things in the world, static factor methods also has some disadvantages e.g. once you make the constructor private to ensure that the class can only be instantiated using the constructor, you lost the power of inheritance because classes without public or protected constructor cannot be extended. But, if you look deep, this is not such a bad thing because it encourages you to favor Composition over Inheritance for code reuse, which results in more robust and flexible software.

In summary, both static factory methods and constructor have their usage and it's important for an experienced developer to understand their relative merits. In most cases, static factories are better choices so avoid the nature of providing public constructors without first considering static factories for creating objects.

That's all about the **difference between factory method and constructor in Java**. You can see that factory method provides several advantages over constructor and should be preferred for object creation in Java. In fact, this is advised by the great Java developer and author of popular Java APIs e.g. Collection framework and several useful classes of java.lang package, Joshua Bloch on his all-time classic Java book, Effective Java. You can read the very first item from this book to learn more about why factory method is better than constructor in Java.

Difference between First and Second Level Cache in Hibernate

If you have used Hibernate in past then you know that one of the strongest points of Hibernate framework is caching, which can drastically improve the performance of Java application's persistence layer if configured and used correctly. Hibernate provides caching at many levels e.g. first level cache at `Session` level, second level cache at the `SessionFactory` level, and query cache to cache frequently executed SQL queries. The first level cache minimizes database access for the same object. For example, if you call the `get()` method to access Employee object with id = 1 from one session, it will go the database and load the object into memory, but it will also cache the object in the first level cache.

When you will call the `get()` method again for the same object from the same session, even after doing some updates on the object, it will return the object from the cache without accessing the database. You can confirm this from Hibernate's log file by observing how many queries are executed. This is also one of the frequently asked Hibernate Interview Questions, so it will not only help to improve the performance of your Java application but also help you to do well on your next interview.

This session level cache greatly improves the performance of Java application by minimizing database roundtrips and executing less number of queries. For example, if an object is modified several times within the same transaction, then Hibernate will only generate one SQL UPDATE statement at the end of the transaction, containing all the modification.

But, since this cache is associated with the `Session` object, which is a short-lived object in Hibernate, as soon as you close the session, all the information held in the cache is lost. So, if you try to load the same object using the `get()` method, Hibernate will go to the database again and fetch the record.

This poses significant performance challenge in an application where multiple sessions are used, but you don't need to worry. Hibernate provides another application level cache, known as second level cache, which can be shared among multiple sessions. This means a request for the same object will not go to the database even if it is executed from multiple session, provided object is present in the second level cache.

The second level cache is maintained at the `SessionFactory` level, which is used to open sessions, hence every session is linked to `SessionFactory`. This cache is opposite to first level cache which is by default enabled in Hibernate, this one is by default disabled and you need to configure the second level cache in Hibernate configuration file to enable it.

The second level cache is provided with the help of caching providers e.g. `EhCache` and `OSCache`. If you look at the cache package in Hibernate, you can see the implementation of Caching related interfaces by these providers. Depending upon which cache you want to use, you can configure them in the Hibernate Configuration file.

Once configured, every request for an object will go to the second level cache if it is not found in the first level cache. It won't hit the database without consulting second level cache, which means improved performance.

It's very important for a Java and Hibernate developer to know about Caching in Hibernate. It's not just important from Interview point of view but also from the application development and performance improvement point of view. You will often face performance related challenges in a real world application which contain millions of records, by correctly configuring Hibernate sessions and writing code which make use of caching, your Java application can float above water even in the case of a significant load. If you want to learn more about Hibernate performance, I suggest reading I suggest reading High-Performance Java Persistence by Vlad Mihalcea, one of the best and up-to-date resources on hibernate performance at the moment.

Difference between First and Second Level Cache in Hibernate

Now that we know what is first level and second level cache in Hibernate, let's revise some key differences between them from interview point of view.

Scope
First level cache is associated with Session Object, while the Second level cache is associated with the SessionFactory object. This means first level cache's scope is limited to session level while second level cache's scope is at the application level. Since Session object is created on demand from the SessionFactory and it's destroyed once the session is closed, the same query if run from two different sessions will hit the database twice if the second level cache is not configured. On the other hand, second level cache remains available throughout the application's life-cycle.

Configuration
First level cache is by default enabled in Hibernate, while the second level cache is optional. If you need it then you need to explicitly enable the second level cache on Hibernate configuration file i.e. the hibernate.cfg.xml file.

You can use
the hibernate.cache.provider_class and hibernate.cache.use_second_level_cache properties to enable the second level cache in Hibernate. The first one is the name of the class which implements Second level cache and could be different, depending upon which cache you use
e.g. **EhCache** or **OSCache**.

By default, hibernate.cache.provider_class is set to org.hibernate.cache.NoCacheProvider class, which means the second level cache is disabled. You can enable it by setting something like org.hibernate.cache.EhCacheProvider if you want to use EhCache as the second level cache.

Here is a sample configuration to configure Second level cache with EhCache:

```
<prop key="hibernate.cache.use_second_level_cache">true</prop>

<prop key="hibernate.cache.provider_class">org.hibernate.cache.EhCacheProvider</prop>
```

Don't forget to include hibernate-ehcache.jar into your classpath. This class comes from that JAR. You can also see Java Persistence with Hibernate, 2nd edition to learn more about other configuration options available to second level cache.

Availability
First level cache is available only until the session is open, once the session is closed, the first level cache is destroyed. On the other hand, second level cache is available through the application's life-cycle, it is only destroyed and recreated when you restart your application.

Order
If an entity or object is loaded by calling the `get()` method then Hibernate first checked the first level cache, if it doesn't found the object then it goes to the second level cache if configured. If the object is not found then it finally goes to the database and returns the object, if there is no corresponding row in the table then it return null. When an object is loaded from the database is put on both second level and first level cache, so that other session who request the same object can now get it from the second level cache.

In case if the object is not found in the first level cache but found in the second level cache because some other sessions have loaded the object before then it is not only returned from first level cache but also cached at first level cache, so that next time if your code request the same object, it should be returned from 1st level cache rather than going to the 2nd level cache.

In general. When an object is pass to `save()`, `update()`, or `saveOrUpdate()` method and retrieved by `load()`, `get()`, `list()`, `iterate()`, or `scroll()` method, that object is added to the internal cache of the Session and when the `flush()` is subsequently called, the state of the object is sychronized with the database.

Second level cache can also be configured on a per-class and per-collection basis, which means it can cache a class or a collection. You can use class-cache and colleection-cache elements in `hibernate.cfg.xml` to specify which class or collection to cache at 2nd level cache. You should remember that second level cache by default doesn't cache any entitty until you configure it.

You can also use JPA Annoation `@Cacheable` to specify which entity is cacheable. and Hibernate annoation `@Cache` to specify caching startegy e.g. `CacheConcurrencyStrategies` like `READ_WRITE` or `READ_ONLY` to tell Hibernate how the second level cache should behave.

That's all about difference between first and second level cache in Hibernate. It's not just an ORM tool, hibernate is much more than that and in-built caching is one of the biggest benefit of using Hibernate to implement psersistence of DAO layer in Hibernate. By correctly configuring second level cache and writing code to leverage both first level and second level cache in Hibernate, you can get improved performance in Java application.

Difference between PriorityQueue and TreeSet in Java?

The `PriorityQueue` and `TreeSet` collection classes has a lot of similarities e.g. both provide `O(log(N))` time complexity for adding, removing, and searching elements, both are non-synchronized and you can get element from both `PriorityQueue` and `TreeSet` in sorted order, but there is fundamental difference between them, TreeSet is a Set and doesn't allow a duplicate element, while `PriorityQueue` is a queue and doesn't have such restriction. It can contain multiple elements with equal values and in that case head of the queue will be arbitrarily chosen from them. Another key difference between `TreeSet` and `PriorityQueue` is *iteration order*, though you can access elements from the head in a sorted order e.g. head always give you lowest or highest priority element depending upon your `Comparable` or `Comparator` implementation but iterator returned by `PriorityQueue` doesn't provide any ordering guarantee.

Only guarantee `PriorityQueue` gives that head will always be smallest or largest element. On the other hand, `TreeSet` keeps all elements in sorted order and iterator returned by `TreeSet` will allow you to access all elements in that sorted order.

This is one of the frequently asked Collection interview questions and what makes it interesting is the subtle difference between a lot of similarities between PriorityQueue and TreeSet. You can use one in place of another in some scenarios but not in all scenarios and that's what Interviewer is looking for when he ask this question to you on Interview.

Similarity between PriorityQueue and TreeSet

Before looking at the difference between PriorityQueue and TreeSet, let's first understand the similarities between them. This will help you to think of scenarios where you can use a PriorityQueue in place of TreeSet or vice-versa.

ThreadSafety

First similarities between PriorityQueue and TreeSet is that both are not thread-safe, which means you cannot share them between multiple threads. If multiple threads are required to modify the TreeSet at the same time, then you must synchronize their access externally.

Ordering

The third similarity between PriorityQueue and TreeSet is that both can be used to access elements in a particular order. TreeSet is a SortedSet, hence it always keeps the elements in the order defined by their Comparator or natural order if there is no Comparator defined, while PriorityQueue will always make sure that head contains the lowest or highest value depending upon the order imposed by Comparator or Comparable.

Eligibility

When I say eligibility, which means which objects can be stored in PrioritySet and TreeSet? Is there any restriction or all objects are allowed? Well, there is, you can only store objects which implement Comparable or Comparator in both PriorityQueue and TreeSet because the collection classes are responsible for keeping their commitment i.e. PriorityQueue must adjust after every insertion or deletion to keep the lowest or highest element in head position. Similarly, TreeSet must re-arrange elements so that they remain the sorted order specified by Comparator or natural order imposed by Comparable.

Performance

This is one point where you will see both similarity and difference between PriorityQueue and TreeSet in Java i.e. both provides `O(log(N))` complexity for adding, removing and searching elements, but when you want to remove the highest or lowest priority element then PriorityQueue gives `O(1)` performance because it always keep the element in head, much like a heap data structure i.e. minimum heap where root is always the minimum value. If you are not familiar with heap data structure, I suggest you reading a good book on data structure e.g. Algorithms 4th Edition by Robert Sedgewick.

Difference between PriorityQueue and TreeSet

Now, that you know and understand similarities between TreeSet and PrioritySet, let's see how different they are? and why you cannot use PrioritySet in place of TreeSet in Java?

Underlying Data Structure

This first and foremost difference is underlying data structure. PriorityQueue is a Queue and that's why it provides the functionality of FIFO data structure, while TreeSet is a Set and doesn't provide the Queue API.

Duplicate Elements

The second difference between them can be derived from the first difference i.e. properties of their underlying data structure. Since TreeSet is a Set it **doesn't allow duplicate elements** but PriorityQueue may contain duplicates. In the case of ties, the head of the priority queue is chosen arbitrarily.

Performance

The third difference between TreeSet and PrirityQueue comes from their relative performance. The PriorityQueue provides largest or smallest element in O(1) time, which is not possible by TreeSet. Since TreeSet is backed by a red black tree, the search operation will take O(logN) time. This is why if you are developing an application where priority matters e.g. a job scheduling algorithm where you always want to execute the job with the highest priority, you should use PriorityQueue data structure.

Availability

The 5th and last difference between PriorityQueue and TreeSet class are that former was added in JDK 1.5 while TreeSet was available from JDK 1.4 itself. This is not a very significant difference in the age of Java 8, but if you are working with legacy systems still running on Java 1.5, a point worth remembering.

Ordering
The fourth difference, which is more subtle than you think because in similarities I have told that both are responsible for keeping some ordering. The key point is that in TreeSet **all elements remain in the sorted order,** while in priority queue apart from root, which is guaranteed to be smallest or largest depending upon Comparing logic, rest of element may or may not follow any ordering.

This means if you store same elements in the TreeSet and PriorityQueue and iterate over them, then their order will be different. TreeSet will print them in sorted order but PriorityQueue will not until you are always getting the element from the head. You can read a good core Java book e.g. Java: The Complete Reference, Ninth Edition to learn more about PriorityQueue implementation in Java.

Using PriorityQueue and TreeSet in Java Program

Now, let's some code to highlight the difference between PriorityQueue and TreeSet in Java. If you remember the most subtle difference I mention in the previous paragraph was that TreeSet keeps all elements in the sorted order, while PriorityQueue only keeps the root or head into order. I mean the lowest element will always be at root in PriorityQueue. If you use poll() which consumes the head and removes it, you will always retrieve the elements in increasing order from PriorityQueue, as shown in following Java program.

```
import java.util.Collections;
import java.util.Date;
import java.util.Iterator;
import java.util.List;
import java.util.PriorityQueue;
import java.util.Queue;
import java.util.Set;
import java.util.TreeSet;

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

      Set setOfNumbers = new TreeSet<>();
      Queue queueOfNumbers = new PriorityQueue<>();

      // inserting elements into TreeSet and PriorityQueue
      setOfNumbers.add(202);
      setOfNumbers.add(102);
      setOfNumbers.add(503);
      setOfNumbers.add(33);

      queueOfNumbers.add(202);
      queueOfNumbers.add(102);
      queueOfNumbers.add(503);
      queueOfNumbers.add(33);

      // Iterating over TreeSet
      System.out.println("Iterating over TreeSet in Java");
      Iterator itr = setOfNumbers.iterator();
      while(itr.hasNext()){
        System.out.println(itr.next());
      }
      // Iterating over PriorityQueue
      System.out.println("Iterating over PriorityQueue in Java");
      itr = queueOfNumbers.iterator();
      while(itr.hasNext()){
        System.out.println(itr.next());
      }

      // Consuming numbers using from head in PriorityQueue
      System.out.println("polling a PriorityQueue in Java");
      while(!queueOfNumbers.isEmpty()){
        System.out.println(queueOfNumbers.poll());
      }
  }
}
```

```
Output:
Iterating over TreeSet in Java
33
102
202
503
Iterating over PriorityQueue in Java
33
102
503
202
polling a PriorityQueue in Java
33
102
202
503
```

You can see that the order is different from the Iterator returned by `PriorityQueue` and `HeadSet` (highlighted by red) but when you use the `poll()` method to consume all elements in PriorityQueue, you have got them in the same order they were present in TreeSet i.e. lowest to highest. This proves the point that *TreeSet keeps all elements in sorted order* while `PriorityQueue` only care about the root or head position. This kind of details are very important from Java certification perspective as well, you will find a lot of questions based upon such subtle details in mock exams as well as on the real exam.

Difference between Transient, Persistent, and Detached Objects in Hibernate

In Hibernate framework, an entity can be in three states, transient, persistent, and detached. When an object is in transient state, it is commonly refereed as transient object, similarly if it is in persistence and detached state, it is known as persistent and detached object. When an entity is first created using the new operator e.g. new User() and not associated with Hibernate session e.g. you haven't called session.save(user) method then it is known as **transient object**. At this stage, Hibernate doesn't know anything about this object and the object doesn't have any representation in database e.g. a corresponding row in the User table. Hibernate will not run any SQL query to reflect any changes on this object. You can move this object into persistent state by associating it with an hibernate session e.g. by calling save() or saveOrUpdate() method from an hibernate Session.

When an entity object moved to Persistence state it become responsibility of Hibernate. Now if you make any change on entity object e.g. change any attribute like user.setName("Mike"), Hibernate will automatically run the update queries to persist the change into database. A persistence object has corresponding representation on the database.

When you close the hibernate session or call the evict() method then the object moves to the detached state. In this state, hibernate doesn't track the object but you can re-attach a detached object to Hibernate session by calling the update() or saveOrUpdate(), or merge() method. Once reattached, the detached object will move to **Persistent** state.

Difference between Transient vs Persistent vs Detached Object in Hibernate

'This is also one of the frequently asked Hibernate Interview questions and even though, both Transient and Detached object is not associated with hibernate session, there is a key difference between them. First, detached object was associated with Hibernate session in past and it has representation in database, on the other hand, Transient object is never associated with hibernate and it doesn't have any representation in database.

But, both can be moved to Persistent state by associating them with session e.g. you can move an entity from transient to persistent state by calling Session.save() method. Similarly, you can move a detached entity to Persistent state by calling Session.update() or Session.saveOrUpdate(), or Session.merge() methods.

Database Representation

The main difference between transient, persistent, and detached object comes from representation in database. When an entity is first created, it goes to transient state and this time it doesn't have a representation in database i.e. there will be no row corresponding to this object in Entity table. On the other hand, both Persistent and Detached objects has corresponding representation in database.

Association with Hibernate

Another key difference between transient, persistent, and detached objects comes from the fact that whether they are associated with session or not. The transient object is not associated with session, hibernate knows nothing about them. Similarly detached object is also not associated with session, but Persistent object is associated with session.

Hence any changes in the Persistent object will reflect in database because Hibernate will automatically run update queries to save changes on Persistent object. See Java Persistence with Hibernate for more details.

mpact of GC

Both transient and detached objects are eligible for garbage collection because they are not associated with session, when GC will run they can be collected, but persistent object is not eligible to garbage collection until session is open because Hibernate Session object keep a reference of Persistent object.

State transition

When an entity is first created in application using the `new()` operator, it remains in transient state. It can move to Persistent state when you associate it with a session by calling `Session.save()` method. When you `close()` the session or `evict()` that object from session, it moves to detached state. You can again move a detached object to Persistent state by calling `Session.update()` or `Session.saveOrUpdate()` method.

Here is a nice Hibernate state diagram which shows how state transition happens in Hibernate by calling different methods in an hibernate entity objects life-cycle:

You can see that when a new object is created it goes to Transient state and then when you call save() or saveOrUpdate() method it goes to Persistent state, and when you call the evict(), clear(), or close() method, it goes to Detached state. Similarly, when you call get() or load() method the object goes to Persistent state because it has representation in database.

Difference between @Autowired and @Inject annotation in Spring?

What is the difference between `@Autowired` and `@Inject` annotation in Spring is one of the frequently asked questions on Spring interviews? Since everybody is now moved or moving to annotation driven, Java configuration in Spring, this question has become even more important for prospective candidates looking for a Java web development job using Spring framework. The `@Autowired`annotation is used for auto-wiring in Spring framework. The Autowiring is a process on which Spring framework figure out dependencies of a Spring bean, instead of you, a developer, explicitly specifying them in the application context file. You can annotate fields and constructor using `@Autowired` to tell Spring framework to find dependencies for you.

The `@Inject` annotation also serves the same purpose, but the main difference between them is that `@Inject` is a **standard annotation** for dependency injection and `@Autowired` is **spring specific**.

Since Spring is not the only framework which provides dependency injection, in future if you change your container and moves to another DI framework like **Google Guice**, you need to reconfigure your application.

You can potentially avoid that development effort by using standard annotations specified by `JSR-330` e.g. `@Inject`, `@Named`, `@Qualifier`, `@Scope` and `@Singleton`. A bean declared to be auto-wired using @Inject will work in both Google Guice and Spring framework, and potentially any other DI container which supports JSR-330 annotations.

Difference between @Autowired vs @Inject Annotation

If you have worked with Hibernate and JPA in past then JSR-330 annotation is nothing but like JPA annotations which standardize the Object Relational mapping across the framework. When you use the JPA annotations like `@Entity`, your code will not only work on Hibernate but also on other ORM tools and framework e.g. TopLink.

Btw, like all similar things in world, even though both `@Autowired` and `@Inject` serve the same purpose there are couple of differences between them, let's examine them briefly

1) The first and most important difference between `@Autowired` and `@Inject` annotation is that the `@Inject` annotation is only available from Spring 3.0 onwards, so if you want to use annotation-driven dependency injection in Spring 2.5 then you have to use the `@Autowired` annotation.

2) The second difference between these two annotations is that unlike Spring's `@Autowired`, the `@Inject` does require the **`'required'`** attribute.

3) The third most common difference between `@Autowired` and `@Inject` annotation is that former is Spring specific while later is the standard for Dependency Injection, specified in JSR-330. In general, I recommend the use of JSR 330 annotation for DI, the `@Inject` annotation is as capable as Spring's `@Autowired` and if you want you can also mix and match this with Spring's `@Value` and`@Lazy` annotations.

4) The `@Autowired` annotation was added on Spring 2.5 and used for annotation driven dependency injection. It works in conjunction with `@Component` annotation and `<context:component-scan />` to streamline development cycle. From **Spring 3.0**, Spring offers support for JSR-330 dependency injection annotations e.g. `@Inject`, `@Named`, and `@Singleton`. It also added more Spring specific annotations e.g. `@Primary`, `@Lazy`, and `@DependsOn` annotation. You can read Spring in Action 4th Edition to learn more about Spring specific annotations.

5) The `@Inject` annotation is good from the portability point of view. Since `@Autowired` is specific to Spring framework, if you ever decided to move to Google Guice or any other dependency injection framework then you need to re-implement your dependency injection logic, even though your application remains same. All bean creation logic needs to be changed to match with Google Guice's implementation.