**DEPENDENCY INJECTION(ref : JavaPapers.com)**

http://www.journaldev.com/2696/spring-interview-questions-and-answers#aspect-advice-pointcut-joinpoint

In object oriented design, objects have relationship with one another. A class (A) can have attributes (B) and methods. Those attributes are again instances of another class (C). If class (A) wants to work and perform its objective, attributes (B) should be instantiated.

There are different ways to instantiate an object. A simple and direct way is to use the “new” operator and call the constructor of Class (C) where we need that instance in class (A). This is class A has obsolute control over creation of attribute (B). It decides which class (C) to call and how to call etc.

Now, if we outsource that ‘instantiation and supplying an instance’ job to some third party. Class (A) needs instance of class (C) to operate, but it outsources that responsibility to some third party. The designated third party, decides the moment of creation and the type to use to create the instance. The dependency between class (A) and class (C) is injected by a third party. Whole of this agreement involves some configuration information too. This whole process is called dependency injection.

Dependency Injection is a design pattern on which dependency of is injected by framework rather than created by [Object](http://javarevisited.blogspot.ca/2012/12/what-is-object-in-java-or-oops-example.html) itself.

One of the implementation of DI is Inversion of Control (IOC) on which framework like Spring controls object’s dependency.

As the name implies **Inversion of control** means now we have inverted the control of creating the object from our own using new operator to container or framework. Now it’s the responsibility of container to create an object as required. We maintain one XML file where we configure our components, services, all the classes and their property. We just need to mention which service is needed by which component and container will create the object for us.

**Inversion of Control** (IoC) is the mechanism to achieve loose-coupling between Objects dependencies. To achieve loose coupling and dynamic binding of the objects at runtime, the objects define their dependencies that are being injected by other assembler objects. Spring IoC container is the program that injects dependencies into an object and make it ready for our use.

We can implement dependency injection pattern to move the dependency resolution from compile-time to runtime.

**Difference between Dependency Injection and Factory Pattern**

Even if we use a factory the dependent class has the responsibility of creating the instance but the core of dependency injection is separating that responsibility to external component.

WITH FACTORY

Class A {

private C obj;

public void someMethod() {

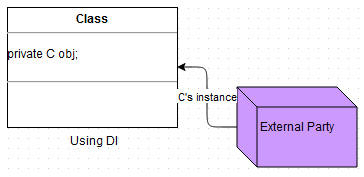
...

this.obj = MyObjectFactory.getC();

...

}

WITH DI



With DI the contract is different, pass C’s instance to get the job done. So the responsibility is with an external person to decide.

Spring’s IOC container is light-weight and it manages the dependency between objects using configurations. It wires the related objects together, instantiates and supplies them based on configuration. In Spring, these objects are called managed objects

Two most popular ways in Spring for DI is, constructor and setter injection. Configuration for dependency are done using Spring-configuration XML or using annotations. Spring manages these objects using [BeanFactory](http://static.springsource.org/spring/docs/2.5.x/api/org/springframework/beans/factory/BeanFactory.html).

**Difference between setter and constructor injection**

1) The fundamental difference between setter and constructor injection, as their name implies is How dependency is injected. Setter injection in Spring uses setter methods like setDependency() to inject dependency on any bean managed by Spring's IOC container. On the other hand constructor injection uses [constructor](http://javarevisited.blogspot.sg/2012/01/what-is-constructor-overloading-in-java.html) to inject dependency on any Spring-managed bean.

2) **Readability** Because of using setter method, setter Injection in more readable than constructor injection in Spring configuration file usually applicationContext.xml . Since setter method has name e.g. setReporotService() by reading Spring XML config file you know which dependency you are setting. While in constructor injection, since it uses an index to inject the dependency, it's not as readable as setter injection and you need to refer either Java documentation or code to find which index corresponds to which property.

3)**Circular Dependency** If Object A and B are dependent each other i.e A is depends ob B and vice-versa. Spring throws ObjectCurrentlyInCreationException while creating objects of A and B bcz A object cannot be created until B is created and vice-versa. So spring can resolve circular dependencies through setter-injection. Objects constructed before setter methods invoked.

1. **Partial dependency**: can be injected using setter injection but it is not possible by constructor. Suppose there are 3 properties in a class, having 3 arg constructor and setters methods. In such case, if you want to pass information for only one property, it is possible by setter method only.
2. **Overriding**: Setter injection overrides the constructor injection. If we use both constructor and setter injection, IOC container will use the setter injection.
3. **Changes**: We can easily change the value by setter injection. It doesn't create a new bean instance always like constructor. So setter injection is flexible than constructor injection.

Use Setter injection when a number of dependencies are more or you need readability. Use Constructor Injection when Object must be created with all of its dependency. If any dependency is mandatory for an Object to perform its duty then you should use Constructor Injection, having said that optional dependency can be injected using Setter injection.

|  |  |
| --- | --- |
|  | In **setter injection** strategy, we trust the IoC container that it will first create the bean first but will do the injection right before using the bean using the setter methods. And the injection is done according to your configuration. If you somehow misses to specify any beans to inject in the configuration, the injection will not be done for those beans and your dependent bean will not function accordingly when it will be in use!  But in **constructor injection** strategy, container imposes (or must impose) to provide the dependencies properly while constructing the bean |

**Spring Core Aliases vs name attribute of bean tag in XML**

Aliases have different behavior if declared in name attribute in a bean definition or declared using the alias tag.

* aliases declared with the alias tag hides any later bean definition with the same name.
* aliases declared in name attribute prevent any other bean definition to use the same name by throwing a BeanDefinitionParsingException.

For example:

<bean id="foo" name="bar" class="Foo" />

<bean id="bar" class="Bar" /> -- throw Exception (name bar is in use)

but

<bean id="foo" class="Foo" />

<alias name="foo" alias="bar" />

<bean id="bar" class="Bar" /> -- Hidden by alias no exception thrown

Following steps explain their life cycle inside the container.

1. The container will look the bean definition inside configuration file (e.g. bean.xml).

2 using reflection container will create the object and if any property is defined inside the bean definition then it will also be set.

3. If the bean implements the BeanNameAware interface, the factory calls setBeanName() passing the bean’s ID.  
4. If the bean implements the BeanFactoryAware interface, the factory calls setBeanFactory(), passing an instance of itself.  
5. If there are any BeanPostProcessors associated with the bean, their post- ProcessBeforeInitialization() methods will be called before the properties for the Bean are set.

6. If an init() method is specified for the bean, it will be called.  
7. If the Bean class implements the DisposableBean interface, then the method destroy() will be called when the Application no longer needs the bean reference.

8. If the Bean definition in the Configuration file contains a 'destroy-method' attribute, then the corresponding method definition in the Bean class will be called.

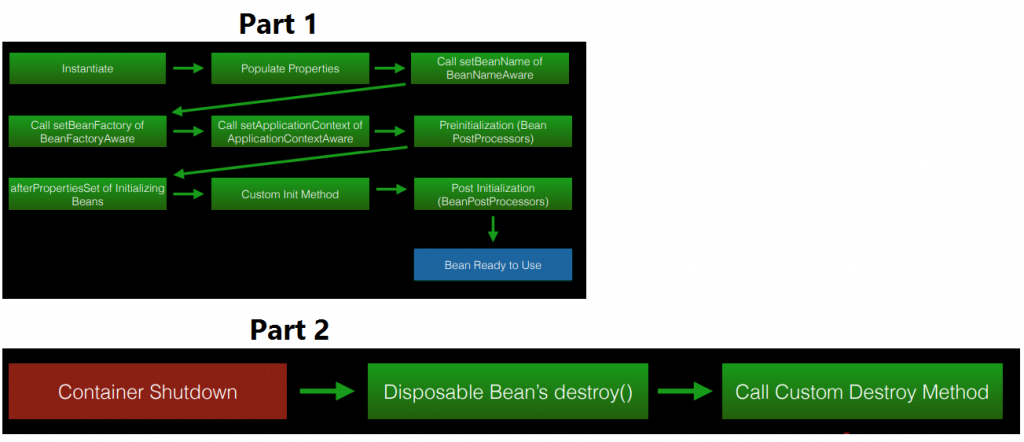


\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

BeanFactory factory = new XmlBeanFactory(new FileInputStream("beans.xml"));

Or

ClassPathResource resorce = new ClassPathResource("beans.xml");   
XmlBeanFactory factory = new XmlBeanFactory(resorce);



**The first difference in these two interfaces** is that they both provide the ability to retrieve the object we need in Spring’s container using the getBean() method, but the BeanFactory only produces the object we need when we call its getBean() method and ApplicationContext will create all the objects we need as soon as we call to Spring container.(application context is initialized)

|  |  |
| --- | --- |
| **ApplicationContext.** | **BeanFactory** |
| Here we can have more than one config files possible | In this only one config file or .xml file |
| Application contexts can publish events to beans that are registered as listeners(event handling) | Doesn’t support. |
| Support internationalization (I18N) messages | It’s not |
| Support application life-cycle events, and validation. | Doesn’t support. |
| Supports  many enterprise services such JNDI access, EJB integration, remoting | Doesn’t support. |

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Aspect**: Aspect is a class that implements cross-cutting concerns, such as transaction management. Aspects can be a normal class configured and then configured in Spring Bean configuration file or we can use Spring AspectJ support to declare a class as Aspect using @Aspect annotation.

**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. You can think of Advices as [Spring interceptors](http://www.journaldev.com/2676/spring-mvc-interceptor-example-handlerinterceptor-handlerinterceptoradapter) or [Servlet Filters](http://www.journaldev.com/1933/java-servlet-filter-example-tutorial).

**Pointcut**: Pointcut are regular expressions that is matched with join points to determine whether advice needs to be executed or not. Pointcut uses different kinds of expressions that are matched with the join points. Spring framework uses the AspectJ pointcut expression language for determining the join points where advice methods will be applied.

**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.

**Advice Arguments**: We can pass arguments in the advice methods. We can use args() expression in the pointcut to be applied to any method that matches the argument pattern. If we use this, then we need to use the same name in the advice method from where argument type is determined.

###  What is the difference between Spring AOP and AspectJ AOP?

AspectJ is the industry-standard implementation for Aspect Oriented Programming whereas Spring implements AOP for some cases. Main differences between Spring AOP and AspectJ are:

* Spring AOP is simpler to use than AspectJ because we don’t need to worry about the weaving process.
* Spring AOP supports AspectJ annotations, so if you are familiar with AspectJ then working with Spring AOP is easier.
* Spring AOP supports only proxy-based AOP, so it can be applied only to method execution join points. AspectJ support all kinds of pointcuts.
* One of the shortcoming of Spring AOP is that it can be applied only to the beans created through Spring Context.

Spring Framework IoC container classes are part of org.springframework.beans and org.springframework.context packages and provides us different ways to decouple the object dependencies.

Some of the useful ApplicationContext implementations that we use are;

* AnnotationConfigApplicationContext: For standalone java applications using annotations based configuration.
* ClassPathXmlApplicationContext: For standalone java applications using XML based configuration.
* FileSystemXmlApplicationContext: Similar to ClassPathXmlApplicationContext except that the xml configuration file can be loaded from anywhere in the file system.
* AnnotationConfigWebApplicationContext and XmlWebApplicationContext for web applications.

###  How to get ServletContext and ServletConfig object in a Spring Bean?

There are two ways to get Container specific objects in the spring bean.

1. Implementing Spring \*Aware interfaces, for these ServletContextAware and ServletConfigAware interfaces, for complete example of these aware interfaces, please read [Spring Aware Interfaces](http://www.journaldev.com/2637/spring-bean-life-cycle)
2. Using @Autowired annotation with bean variable of type ServletContext and ServletConfig. They will work only in servlet container specific environment only though.
3. @Autowired

ServletContext servletContext;

* **ServletContextAware** - to inject ServletContext object in MVC application, example usage is to read context parameters and attributes.
* **ServletConfigAware** - to inject ServletConfig object in MVC application, example usage is to get servlet config parameters.

### What Is The Difference Between @Inject And @Autowired? Which One to Use Under What Circumstances?

The short answer: There is no different and can be used interchangeably.

In more detail the **@Inject** annotation is part of Java EE 7’s Context and Dependency Injection framework ([JSR 346](https://jcp.org/en/jsr/detail?id=346) also see [JSR 365](https://www.jcp.org/en/jsr/detail?id=365) for Java 2.0) while **@Autowired** is the Spring Frameworks own implementation (see [Java doc](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/beans/factory/annotation/Autowired.html)).

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

### How would you achieve Transaction Management in Spring?

ANSWER:

1. Support for Declarative Transaction Management. In this model, Spring uses AOP over the transactional methods to provide data integrity. This is the preferred approach and works in most of the cases.
2. Support for most of the transaction APIs such as JDBC, Hibernate, JPA, JDO, JTA etc. All we need to do is use proper transaction manager implementation class. For example org.springframework.jdbc.datasource.DriverManagerDataSource for JDBC transaction management and org.springframework.orm.hibernate3.HibernateTransactionManager if we are using Hibernate as ORM tool.
3. Support for programmatic transaction management by using TransactionTemplate or PlatformTransactionManager implementation.

Ref-- [http://www.journaldev.com/2603/spring-transaction-management-jdbc-example#](http://www.journaldev.com/2603/spring-transaction-management-jdbc-example)

public class CustomerDAOImpl implements CustomerDAO {

private DataSource dataSource;

public void setDataSource(DataSource dataSource) {

this.dataSource = dataSource;

}

@Override

public void create(Customer customer) {

String queryCustomer = "insert into Customer (id, name) values (?,?)";

String queryAddress = "insert into Address (id, address,country) values (?,?,?)";

JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

jdbcTemplate.update(queryCustomer, new Object[] { customer.getId(),

customer.getName() });

System.out.println("Inserted into Customer Table Successfully");

jdbcTemplate.update(queryAddress, new Object[] { customer.getId(),

customer.getAddress().getAddress(),

customer.getAddress().getCountry() });

System.out.println("Inserted into Address Table Successfully");

}

}

CustomerDAO implementation is not taking care of transaction management. This way we are achieving separation of concerns because sometimes we get DAO implementations from third party and we don’t have control on these classes.

Let’s create a Customer Service that will use the CustomerDAO implementation and provide transaction management when inserting records in the customer and address tables in a single method.

public class CustomerManagerImpl implements CustomerManager {

private CustomerDAO customerDAO;

public void setCustomerDAO(CustomerDAO customerDAO) {

this.customerDAO = customerDAO;

}

@Override

**@Transactional**

public void createCustomer(Customer cust) {

customerDAO.create(cust);

}

}

If you notice the CustomerManager implementation, it’s just using CustomerDAO implementation to create the customer but provide declarative transaction management through annotating createCustomer() method with @Transactional annotation. That’s all we need to do in our code to get the benefits of Spring transaction management.

What spring internally does

Spring framework is using Around advice to generate a proxy class for CustomerManagerImpl and only committing the transaction if the method returns successfully. If there is any exception, it’s just rolling back the whole transaction

Make below change in spring.xml

<!-- Enable Annotation based Declarative Transaction Management -->

<tx:annotation-driven proxy-target-class="true"

transaction-manager="transactionManager" />

<!-- Creating TransactionManager Bean, since JDBC we are creating of type

DataSourceTransactionManager -->

<bean id="transactionManager"

class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

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

</bean>

<!-- MySQL DB DataSource -->

<bean id="dataSource"

class="org.springframework.jdbc.datasource.DriverManagerDataSource">

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

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

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

<property name="password" value="pankaj123" />

</bean>

<bean id="customerDAO" class="com.journaldev.spring.jdbc.dao.CustomerDAOImpl">

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

</bean>

<bean id="customerManager" class="com.journaldev.spring.jdbc.service.CustomerManagerImpl">

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

</bean>

</beans>

 **tx:annotation-driven** element is used to tell Spring context that we are using annotation based transaction management configuration. **transaction-manager** attribute is used to provide the transaction manager bean name. transaction-manager default value is *transactionManager* but I am still having it to avoid confusion. **proxy-target-class** attribute is used to tell Spring context to use class based proxies, without it you will get runtime exception with message such as *Exception in thread “main” org.springframework.beans.factory.BeanNotOfRequiredTypeException: Bean named ‘customerManager’ must be of type [com.journaldev.spring.jdbc.service.CustomerManagerImpl], but was actually of type [com.sun.proxy.$Proxy6]*

 Since we are using JDBC, we are creating transactionManager bean of type org.springframework.jdbc.datasource.DataSourceTransactionManager. This is very important and we should use proper transaction manager implementation class based on our transaction API use.

 *dataSource* bean is used to create the DataSource object and we are required to provide the database configuration properties such as driverClassName, url, username and password. Change these values based on your local settings.

 We are injecting *dataSource* into *customerDAO* bean. Similarly we are injecting *customerDAO* bean into *customerManager* bean definition.

**Programmatic transaction management** is usually a good idea only if you have a small number of transactional operations. For example, if you have a web application that require transactions only for certain update operations, you may not want to set up transactional proxies using Spring or any other technology. In this case, using the TransactionTemplate may be a good approach. Being able to set the transaction name explicitly is also something that can only be done using the programmatic approach to transaction management.

On the other hand, if your application has numerous transactional operations, declarative transaction management is usually worthwhile. It keeps transaction management out of business logic, and is not difficult to configure.

**EmployeeDaoImpl.java**

1. **package** com.dineshonjava.sdnext.dao.impl;
2. **import** java.util.List;
3. **import** org.springframework.dao.DataAccessException;
4. **import** org.springframework.jdbc.core.support.JdbcDaoSupport;
5. **import** org.springframework.stereotype.Component;
6. **import** org.springframework.transaction.PlatformTransactionManager;
7. **import** org.springframework.transaction.TransactionDefinition;
8. **import** org.springframework.transaction.TransactionStatus;
9. **import** org.springframework.transaction.support.DefaultTransactionDefinition;
10. **import** com.dineshonjava.sdnext.dao.EmpDao;
11. **import** com.dineshonjava.sdnext.domain.Employee;
12. **import** com.dineshonjava.sdnext.jdbc.utils.EmployeeMapper;
13. /\*\*
14. \* @author Dinesh Rajput
15. \*
16. \*/
17. @Component
18. **public** **class** EmployeeDaoImpl **extends** JdbcDaoSupport **implements** EmpDao {
19. // @Autowired
20. // private JdbcTemplate jdbcTemplateObject;
21. //
22. // /\*\*
23. //  \* @param jdbcTemplateObject the jdbcTemplateObject to set
24. //  \*/
25. // public void setJdbcTemplateObject(JdbcTemplate jdbcTemplateObject) {
26. //  this.jdbcTemplateObject = jdbcTemplateObject;
27. // }
28. **private** PlatformTransactionManager transactionManager;
29. **public** **void** setTransactionManager(
30. PlatformTransactionManager transactionManager) {
31. **this**.transactionManager = transactionManager;
32. }
33. @Override
34. **public** **void** create(String name, Integer age, Long salary) {
35. TransactionDefinition **def** = **new** DefaultTransactionDefinition();
36. TransactionStatus status = transactionManager.getTransaction(**def**);
37. **try** {
38. String SQL = "INSERT INTO Employee (name, age, salary) VALUES (?, ?, ?)";
39. getJdbcTemplate().update(SQL, **new** Object[]{name, age, salary} );
40. transactionManager.commit(status);
41. System.**out**.println("Created Record Name = " + name + " Age = " + age+ " Salary = " + salary);
42. // to simulate the exception.
43. **throw** **new** RuntimeException("simulate Error condition") ;
44. } **catch** (DataAccessException e) {
45. System.**out**.println("Error in creating record, rolling back");
46. transactionManager.rollback(status);
47. **throw** e;
48. }
49. }

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Using HibernateDaoSupport/HibernateTemplate is not recommended since it unnecessarily ties your code to Spring classes.

Using these classes was inevitable with older versions of Hibernate in order to integrate support of Spring-managed transactions.

However, since Hibernate 3.0.1 you don't need it any more - you can write a code against a plain Hibernate API while using Spring-managed transactions. All you need is to configure Spring transaction support, inject SessionFactory and call getCurrentSession() on it when you need to work with session.

Another benefit of HibernateTemplate is exception translation. Without HibernateTemplate the same functionality can be achieved by using @Repository annotation

Good exception questions-

http://www.journaldev.com/2167/java-exception-interview-questions-and-answers#java-exception-programming-questions  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

getCurrentSession() in hibernate

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.

HOW TO CONFIGURE IT IN Hibernate config file

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

* **ApplicationContextAware** – to inject ApplicationContext object, example usage is to get the array of bean definition names.
* **BeanFactoryAware** – to inject BeanFactory object, example usage is to check scope of a bean.

@Override

public void setBeanFactory(BeanFactory beanFactory) throws BeansException {

System.out.println("setBeanFactory called");

System.out.println("setBeanFactory:: employee bean [singleton](http://www.journaldev.com/1377/java-singleton-design-pattern-best-practices-examples)="

+ beanFactory.isSingleton("employee"));

}

* **BeanNameAware** – to know the bean name defined in the configuration file.
* **ResourceLoaderAware** – to inject ResourceLoader object, example usage is to get the input stream for a file in the classpath.
* **ServletContextAware** – to inject ServletContext object in MVC application, example usage is to read context parameters and attributes.
* **ServletConfigAware** – to inject ServletConfig object in MVC application, example usage is to get servlet config parameters.

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:context="http://www.springframework.org/schema/context"

xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.0.xsd"

**default-autowire="byName" default-autowire-candidates="\*"** >

<bean name="employee" class="com.journaldev.spring.autowiring.model.Employee">

<property name="name" value="Pankaj"></property>

</bean>

<bean name="employee1" class="com.journaldev.spring.autowiring.model.Employee" **autowire-candidate="false">**

<property name="name" value="Dummy Name"></property>

</bean>

</beans>

 **beans** element default-autowire is used to define the default autowiring method. Here I am defining the default autowiring method to be byName.

 **beans** element default-autowire-candidates is used to provide the pattern for bean names that can be used for autowiring. For simplicity I am allowing all the bean definitions to be eligible for autowiring, however if we can define some pattern for autowiring. For example, if we want only DAO bean definitions for autowiring, we can specify it as default-autowire-candidates="\*DAO".

 autowire-candidate="false" is used in a bean definition to make it ineligible for autowiring. It’s useful when we have multiple bean definitions for a single type and we want some of them not to be autowired. For example, in above spring bean configurations “employee1” bean will not be used for autowiring.

AOP implementations are provided by:

1. AspectJ
2. Spring AOP
3. JBoss AOP

**Spring AOP allows aspects to apply to beans declared in the IoC container. If you want to use additional pointcut types or apply your aspects “to objects created outside the Spring IoC container“, you have to use the AspectJ framework in your Spring application and use it’s weaving feature.**

Weaving is the process of linking aspects with other outsider application types or objects to create an advised object. This can be done at compile time (using the AspectJ compiler, for example), load time, or at runtime.

**AspectJ compile-time weaving** is done through a special AspectJ compiler called ajc. It can weave aspects into your Java source files and output woven binary class files. It can also weave aspects into your compiled class files or JAR files. This process is known as post-compile-time weaving. You can perform compile-time and post-compile-time weaving for your classes before declaring them in the Spring IoC container. Spring is not involved in the weaving process at all.

**AspectJ load-time weaving** (also known as LTW) happens when the target classes are loaded into JVM by a class loader. For a class to be woven, a special class loader is required to enhance the bytecode of the target class. Both AspectJ and Spring provide load-time weavers to add load-time weaving capability to the class loader. You need only simple configurations to enable these load-time weavers.

Spring AOP, like other pure Java AOP frameworks, performs weaving at runtime.

In Spring AOP makes it possible to modularize and separate logging, transaction like services and apply them declaratively to the components Hence programmer can focus on specific concerns. Aspects are wired into objects in the spring XML file in the way as JavaBean. This process is known as 'Weaving'.

Most aspects are a *combination of advice* that defines the aspect’s behavior and a *pointcut* defining where the aspect should be executed.

AdvisorSpring recognizes this and offers advisors, which combine advice and pointcuts into one object.

More specifically, the PointcutAdvisor does this.

public interface PointcutAdvisor {

Pointcut getPointcut();

Advice getAdvice();

}

Most of Spring’s built-in pointcuts also have a corresponding PointcutAdvisor. This is convenient if you want to define a pointcut and the advice it is managing in one place.

* XML mapping file comes handy when we are using third party classes and we can’t use annotations.
* SessionFactory is responsible to read the hibernate configuration parameters and connect to the database and provide Session objects.
* Internal state of SessionFactory is immutable, so it’s thread safe. Multiple threads can access it simultaneously to get Session instances.
* Hibernate Session object is not thread safe, every thread should get it’s own session instance and close it after it’s work is finished.

Hibernate SessionFactory provides three methods through which we can get Session object – getCurrentSession(), openSession() and openStatelessSession().

1. 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 like below.

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

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. Hibernate Session objects are not thread safe, so we should not use it in multi-threaded environment. We can use it in single threaded environment because it’s relatively faster than opening a new session.

1. 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. For web application frameworks, we can choose to open a new session for each request or for each session based on the requirement.
2. Hibernate SessionFactory openStatelessSession() method returns instance of StatelessSession. StatelessSession in Hibernate does not implement first-level cache and it doesn’t interact with any second-level cache. Since it’s stateless, it doesn’t implement transactional write-behind or automatic dirty checking or do cascading operations to associated entities. It’s more like a normal JDBC connection and doesn’t provide any benefits that come from using hibernate framework. However, stateless session can be a good fit in certain situations. For example where we are loading bulk data into database and we don’t want hibernate session to hold huge data in first-level cache memory.

The key unit of modularity in OOP is the class, whereas in AOP the unit of modularity is the aspect. Aspects enable the modularization of concerns such as transaction management that cut across multiple types and objects. (Such concerns are often termed crosscutting concerns in AOP literature.)

A proxy is a well-used design pattern. To put it simply, **a proxy is an object that looks like another object, but adds special functionality behind the scene**.

Spring AOP is proxy-based. AOP proxy is an object created by the AOP framework in order to implement the aspect contracts in runtime.

Get vs Load

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.
4. get() returns the object by fetching it from database or from [hibernate cache](http://www.journaldev.com/2969/hibernate-caching-first-level-cache) whereas load() just returns the reference of an object that might not actually exists, it loads the data from database or cache only when you access other properties of the object.

Lets say id 100 does not exists in employee table.

try{

Employee emp1 = (Employee) session.load(Employee.class, new Long(100));

System.out.println("Employee load called");

System.out.println("Employee LOAD ID= "+emp1.getId());

System.out.println("Employee load Details:: "+emp1+"\n");

}catch(Exception e){

e.printStackTrace();

}

1. when we use get() to retrieve data that doesn’t exists, it returns **null**. That makes sense because it try to load the data as soon as it’s called.

With load(), we are able to print the id but as soon as we try to access other fields, it fires database query and throws org.hibernate.ObjectNotFoundException if there is no record found with the given identifier. It’s hibernate specific [Runtime Exception](http://www.journaldev.com/1696/exception-handling-in-java), so we don’t need to catch it explicitly.

Important methods for first level cache

1. session.contains(emp) returns true/false depending upon whether session/cache contains the object or not
2. session.clear(); to clear everything in the current session cache
3. session.evict(emp);- to remove a particular object from first level cache/session.

Statistics for second level cache

SessionFactory sessionFactory = HibernateUtil.getSessionFactory();

Statistics stats = sessionFactory.getStatistics();

stats.setStatisticsEnabled(true);

some of the methods are:-

stats.getEntityFetchCount()

stats.getSecondLevelCacheHitCount()

stats.getSecondLevelCachePutCount()

stats.getSecondLevelCacheMissCount()

Hibernate uses [Reflection API](http://www.journaldev.com/1789/java-reflection-example-tutorial) to create instance of Entity beans, usually when you call get() or load() methods. The method Class.newInstance() is used for this and it requires no-args constructor. So if you won’t have no-args constructor in entity beans, hibernate will fail to instantiate it and you will get HibernateException.

Sorted lists- When we use Collection API sorting algorithms to sort a collection, it’s called sorted list. For small collections, it’s not much of an overhead but for larger collections it can lead to slow performance and OutOfMemory errors.

Ordered Lists

List<Employee> empList = session.createCriteria(Employee.class)

.addOrder(Order.desc("id")).list();

Ordered list is better than sorted list because the actual sorting is done at database level, that is fast and doesn’t cause memory issues.

Bag,set,List,Array,map – hibernate collection types

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*8

There are various ways to implement joins in hibernate.

* Using associations such as one-to-one, one-to-many etc.
* Using JOIN in the HQL query. There is another form “join fetch” to load associated data simultaneously, no lazy loading.
* We can fire native sql query and use join keyword.

HQL Join example

***//HQL join examples***

***query = session.createQuery("select e.name, a.city from Employee e "***

***+ "INNER JOIN e.address a");***

***List<Object[]> list = query.list();***

***for(Object[] arr : list){***

***System.out.println(Arrays.toString(arr));***

***}***

***//HQL Delete Employee, we need to take care of foreign key constraints too***

***query = session.createQuery("delete from Address where id= :id");***

***query.setLong("id", 4);***

***result = query.executeUpdate();***

***System.out.println("Address Delete Status="+result);***

***query = session.createQuery("delete from Employee where id= :id");***

***query.setLong("id", 4);***

***result = query.executeUpdate();***

***System.out.println("Employee Delete Status="+result);***

Once delete operation is performed, further operations will not show that record data

If we use HQL query, we need to take care of table mappings in our code. On the other hand, If we use Session to delete the Employee object, it will delete the record from both the tables.

Hibernate Native SQL Queries: Hibernate SQL Query is very handy when we have to execute database vendor specific queries that are not supported by Hibernate API. For example query hints or the CONNECT keyword in Oracle Database.

For normal scenarios, Hibernate SQL query is not the recommended approach because we loose benefits related to hibernate association and [hibernate first level cache](http://www.journaldev.com/2969/hibernate-caching-first-level-cache).

For Hibernate Native SQL Query, we use Session.createSQLQuery(String query) to create the SQLQuery object and execute it

SQLQuery query = session.createSQLQuery("select emp\_id, emp\_name, emp\_salary from Employee");

List<Object[]> rows = query.list();

for(Object[] row : rows){

Employee emp = new Employee();

emp.setId(Long.parseLong(row[0].toString()));

emp.setName(row[1].toString());

emp.setSalary(Double.parseDouble(row[2].toString()));

System.out.println(emp);

}

our query is not returning Address data, whereas if we use HQL query "from Employee", it returns the associated table data too.

If we use below then there is slight performance wise improvement

query = session.createSQLQuery("select emp\_id, emp\_name, emp\_salary from Employee")

.addScalar("emp\_id", new LongType())

.addScalar("emp\_name", new StringType())

.addScalar("emp\_salary", new DoubleType());

i.e we used addScalar() method to define the data type of the column.

Hibernate Named Query helps us in grouping queries at a central location rather than letting them scattered all over the code.  
Hibernate Named Query syntax is checked when the hibernate session factory is created, thus making the application fail fast in case of any error in the named queries.  
Hibernate Named Query is global, means once defined it can be used throughout the application.

Hibernate Named Queries can be defined in Hibernate mapping files or through the use of JPA annotations @NamedQuery and @NamedNativeQuery.

Hibernate provides Criteria API that is more object oriented for querying the database and getting results. We can’t use Criteria to run update or delete queries or any DDL statements. It’s only used to fetch the results from the database using more object oriented approach.

criteria = session.createCriteria(Employee.class)

.add(Restrictions.eq("id", new Long(3)));

Employee emp = (Employee) **criteria.uniqueResult();**

System.out.println("Name=" + emp.getName() + ", City="

+ emp.getAddress().getCity());

//Join example for selecting few columns

criteria = session.createCriteria(Employee.class, "employee");

criteria.setFetchMode("employee.address", FetchMode.JOIN);

criteria.createAlias("employee.address", "address"); // inner join by default

ProjectionList columns = Projections.projectionList()

.add(Projections.property("name"))

.add(Projections.property("address.city"));

criteria.setProjection(columns);

List<Object[]> list = criteria.list();

ManyToMany unidirectional relationship example. Cart has many items. And an item can be present in many carts.

Cart.java

@ManyToMany(targetEntity = Item1.class, cascade = { CascadeType.ALL })

@JoinTable(name = "CART\_ITEMS",

joinColumns = { @JoinColumn(name = "cart\_id") },

inverseJoinColumns = { @JoinColumn(name = "item\_id") })

private Set<Item1> items;

Test Program

public static void main(String[] args) {

Item1 item1 = new Item1();

item1.setDescription("samsung"); item1.setPrice(300);

Item1 item2 = new Item1();

item2.setDescription("nokia"); item2.setPrice(200);

Cart1 cart = new Cart1();

cart.setTotal(500);

Set<Item1> items = new HashSet<Item1>();

items.add(item1); items.add(item2);

cart.setItems(items);

SessionFactory sessionFactory = null;

try{

sessionFactory = HibernateAnnotationUtil.getSessionFactory();

Session session = sessionFactory.getCurrentSession();

Transaction tx = session.beginTransaction();

session.save(cart);

System.out.println("Before committing transaction");

tx.commit();

sessionFactory.close();

System.out.println("Cart ID="+cart.getId());

System.out.println("Item1 ID="+item1.getId());

System.out.println("Item2 ID="+item2.getId());

Output

Hibernate: insert into CART (cart\_total) values (?)

Hibernate: insert into ITEM (item\_desc, item\_price) values (?, ?)

Hibernate: insert into ITEM (item\_desc, item\_price) values (?, ?)

Before committing transaction

Hibernate: insert into CART\_ITEMS (cart\_id, item\_id) values (?, ?)

Hibernate: insert into CART\_ITEMS (cart\_id, item\_id) values (?, ?)

Cart ID=5

Item1 ID=6

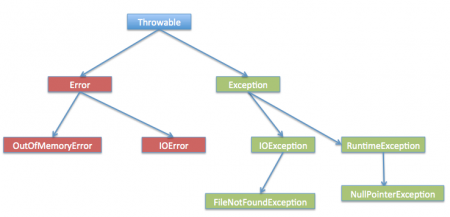
Item2 ID=5

For collections, try to use Lists, maps and sets. Avoid array because you don’t get benefit of lazy loading.

Throwable is the parent class of Java Exceptions Hierarchy and it has two child objects – Error and Exception.

Exception is the parent class of all Checked Exceptions.

RuntimeException is the parent class of all runtime exceptions.



* Checked exception if handled by catch block, then must be specified in throws clause, otherwise compile time error.
* if a method declares unchecked exception in throws clause, it is not mandatory to handle that in the program.
* If super class method is not throwing any exceptions, then it can be overridden with any unchecked type of exceptions, but cannot be overridden with checked type of exceptions.
* If a super class method is throwing unchecked exception, then it can be overridden in the sub class with same exception or any other unchecked exceptions but cannot be overridden with checked exceptions.
* If super class method is throwing checked type of exception, then it can be overridden with same exception or with it’s sub class exceptions i.e you can decrease the scope of the exception, but can not be overridden with it’s super class exceptions i.e you can not increase the scope of the exception.

**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 .

ArrayList Vs Vector

1. **A Vector defaults to doubling size of its array .** While when you insert an element into the **ArrayList** ,     **it increases** **its Array size by 50%  .**  
   Java ArrayList class uses a**dynamic array** for storing the elements. It is like an array, but there is no size limit.  
     
   By default ArrayList size is 10 .

The size of ArrayList grows automatically which is fully based on **load factor** and **current capacity.**Basically, the load factor is the measure that decides when to increase the capacity of the ArrayList. The default load factor of an ArrayList is **0.75f**.

It checks whether it reaches the       last  element then it will create the new array ,copy the new data of last array to new array ,then old array     is garbage collected by the Java Virtual Machine (JVM) .

For example, if the **ArrayList**has 10 element and load factor of 0.75f, then threshold will be,

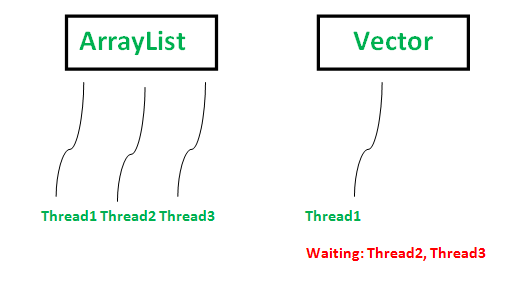
Threshold = 10 \* 0.75 = 7

Let’s take an example and understand how ArrayList grows in memory. Suppose we have an ArrayList with default capacity 10. We can add more than 10 elements in ArrayList.  
Whenever we perform add operation, it checks whether the threshold value is equal to the number of elements present in ArrayList. If the size of the current element is greater than the threshold. The JVM creates a new ArrayList and copies the old ArrayList into new. It creates the ArrayList size 10 to 15.

int newCapacity = oldCapacity + (oldCapacity >> 1);

For example, if the Array size is 10 and already all the rooms were filled by the elements, while we are adding a new element now the array capacity will be increased as 10+ (10>>1) => 10+ 5 => 15. Here the size is increased from 10 to 15. To increase the size by **50%** we use the [right shift operator.](https://www.geeksforgeeks.org/bitwise-shift-operators-in-java/)While in **Java 6** it’s totally different from the above calculation on increasing the size of the Array, in java 6 the capacity increases by the amount to **1.5X**

* **Synchronization:** Vector is **synchronized**, which means only one thread at a time can access the code, while ArrayList is**not synchronized**, which means multiple threads can work on ArrayList at the same time. For example, if one thread is performing an add operation, then there can be another thread performing a remove operation in a multithreading environment. If multiple threads access ArrayList concurrently, then we must synchronize the block of the code which modifies the list structurally or allow simple element modifications. Structural modification means the addition or deletion of element(s) from the list. Setting the value of an existing element is not a structural modification.



* **Performance:** ArrayList is faster. Since it is non-synchronized, while vector operations give slower performance since they are synchronized (thread-safe), if one thread works on a vector, it has acquired a lock on it, which forces any other thread wanting to work on it to have to wait until the lock is released.
* **Data Growth:**ArrayList and Vector **both grow and shrink dynamically**to maintain optimal use of storage – but the way they resize is different. ArrayList increments 50% of the current array size if the number of elements exceeds its capacity, while vector increments 100% – essentially doubling the current array size.
* **Traversal:**Vector can use both [**Enumeration and Iterator**](https://www.geeksforgeeks.org/iterators-in-java/) for traversing over vector elements, while ArrayList can only use **Iterator** for traversing.
* **Applications:**Most of the time, programmers prefer ArrayList over Vector because ArrayList can be synchronized explicitly using [Collections.synchronizedList](https://www.geeksforgeeks.org/collections-synchronizedlist-method-in-java-with-examples/).

Collection framework hierarchy:

Diagram

Description automatically generated

Iterator vs Enumeration

Iterators allow the caller to remove elements from the underlying collection during the iteration with well-defined semantics

Enumeration methods :

hasMoreElements()  
\*nextElement()

Iterator methods :

\*hasNext()  
\*next()  
\*remove()

Iterator is fail fast. if the Hashtable is structurally modified at any time after the iterator is created in any way except the iterator's own remove method , then the iterator will throw ConcurrentModification

Where as Enumeration is fail-safe in nature. It doesn’t throw any exceptions if a collection is modified while iterating. Enumeration is not safe and secured due to it’s fail-safe nature. Iterator is safer

Exception. both Enumeration and Iterator will give successive elements, but Iterator is new and improved version where method names are shorter, and has new method called remove.

Iterator interface is introduced from JDK 1.2 where as Enumeration interface is there from JDK 1.0.

Enumeration is a legacy interface used to traverse only the legacy classes like Vector, HashTable and Stack. Where as Iterator is not a legacy code which is used to traverse most of the classes in the collection framework. For example, ArrayList, LinkedList, HashSet, LinkedHashSet, TreeSet, HashMap, LinkedHashMap, TreeMap etc.

**What is the difference between peek(),poll() and remove() method of the Queue interface ?**  
  
Both poll() and remove() method is used to remove head object of the Queue. The main difference lies when the Queue is empty().  
If Queue is empty then poll() method will return null . While in similar case , 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.

Array vs Arraylist

|  |  |
| --- | --- |
| **Array** | **ArrayList** |
| Arrays are static in nature. Arrays are fixed length data structures. You can’t change their size once they are created. | ArrayList is dynamic in nature. Its size is automatically increased if you add elements beyond its capacity. |
| Arrays can hold both primitives as well as objects. | ArrayList can hold only objects. If you try to insert primitive data into ArrayList, data is automatically boxed into corresponding wrapper class. |
| Arrays can be iterated only through *for* loop or *for-each* loop. | ArrayList provides iterators to iterate through their elements. |
| The size of an array is checked using *length* attribute. | The size of an ArrayList can be checked using *size()* method. |
| Array gives constant time performance for both add and get operations. | ArrayList also gives constant time performance for both add and get operations provided adding an element doesn’t trigger resize. |
| Arrays don’t support generics. | ArrayList supports generics. |
| Arrays are not type safe. | ArrayList are type safe. |
| Arrays can be multi-dimensional. | ArrayList can’t be multi-dimensional. |
| Elements are added using assignment operator. | Elements are added using add() method. |

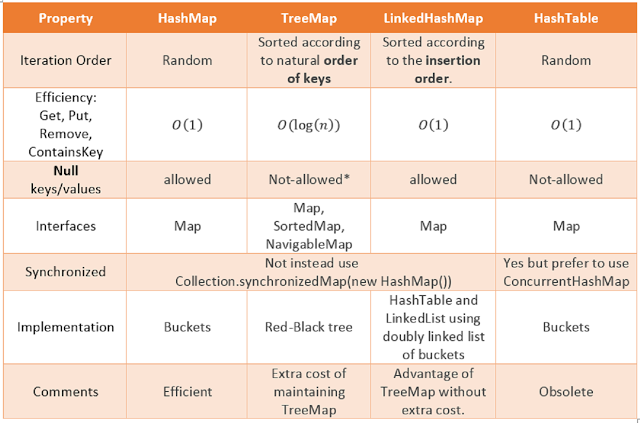
**Differences Between ArrayList And LinkedList In Java:**

|  |  |  |
| --- | --- | --- |
|  | **ArrayList** | **LinkedList** |
| Structure | ArrayList is an index based data structure where each element is associated with an index. | Elements in the LinkedList are called as nodes, where each node consists of three things – Reference to previous element, Actual value of the element and Reference to next element.  (This class uses a [doubly linked list](https://www.geeksforgeeks.org/doubly-linked-list/) to store the elements in it.) |
| Insertion And Removal | Insertions and Removals in the middle of the ArrayList are very slow. Because after each insertion and removal, elements need to be shifted. | Insertions and Removals from any position in the LinkedList are faster than the ArrayList. Because there is no need to shift the elements after every insertion and removal. Only references of previous and next elements are to be changed. |
| Insertion and removal operations in ArrayList are of order O(n). | Insertion and removal in LinkedList are of order O(1). |
| Retrieval(Searching or getting an element) | Retrieval of elements in the ArrayList is faster than the LinkedList . Because all elements in ArrayList are index based. | Retrieval of elements in LinkedList is very slow compared to ArrayList. Because to retrieve an element, you have to traverse from beginning or end (Whichever is closer to that element) to reach that element. |
| Retrieval operation in ArrayList is of order of O(1). | Retrieval operation in LinkedList is of order of O(n). |
| Random Access | ArrayList is of type Random Access. i.e elements can be accessed randomly. | LinkedList is not of type Random Access. i.e elements can not be accessed randomly. you have to traverse from beginning or end to reach a particular element. |
| Usage | ArrayList can not be used as a Stack or Queue. | LinkedList, once defined, can be used as ArrayList, Stack, Queue, Singly Linked List and Doubly Linked List. This class implements both the List interface and the [Deque interface](https://www.geeksforgeeks.org/deque-interface-java-example/). Therefore, it can act as a list and a deque. |
| Memory Occupation | ArrayList requires less memory compared to LinkedList. Because ArrayList holds only actual data and it’s index. | LinkedList requires more memory compared to ArrayList. Because, each node in LinkedList holds data and reference to next and previous elements. |
| When To Use | If your application does more retrieval than the insertions and deletions, then use ArrayList. | If your application does more insertions and deletions than the retrieval, then use LinkedList. |

## Similarities Between ArrayList And LinkedList In Java :

* Both ArrayList and LinkedList implement **List interface**.
* Both ArrayList and LinkedList are **Cloneable** and **Serializable**.
* Both ArrayList and LinkedList maintain **insertion order**.
* Both are **non synchronized**.

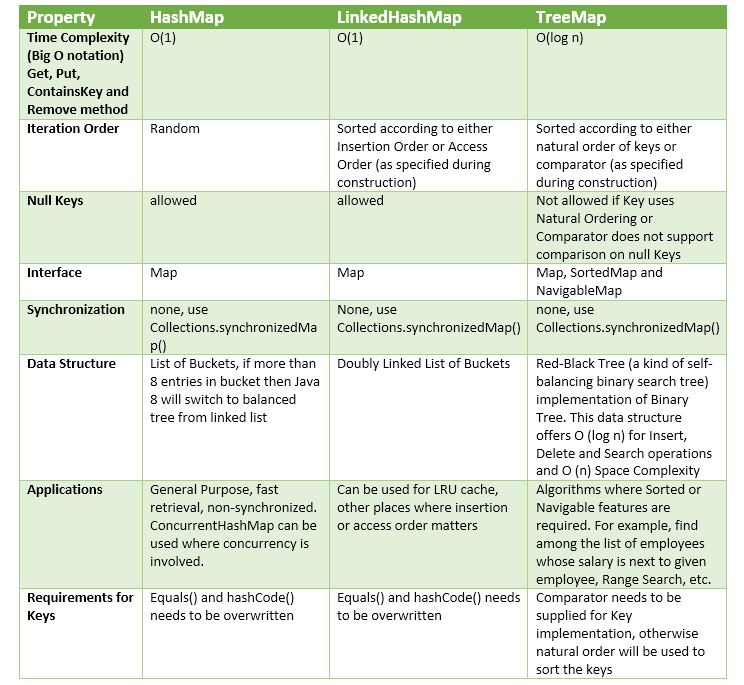
Difference between Hashmap, Treemap and LinkedHashMap



TreeMap is Red-Black tree based NavigableMap implementation while HashMap is internally backed by an array. It uses index[0] to store entries corresponding to null keys.  
  
 LinkedHashMap actually extends HashMap and implements Map interface. 

HashMap allows one null key and multiple null values. It keeps null key based entries on index[0] on an internal bucket. If you look at the put() method of HashMap, you can see, it doesn't throw [NullPointerException for null keys](http://javarevisited.blogspot.sg/2012/06/common-cause-of-javalangnullpointerexce.html). Since LinkedHashMap is a subclass of HashMap, it also allows null keys and values.  
  
On the other hand, TreeMap, which sorts elements in natural order doesn't allow null keys because compareTo() method throws NullPointerException if compared with null. If you are using TreeMap with [user defined Comparator](http://java67.blogspot.sg/2014/11/java-8-comparator-example-using-lambda-expression.html) than it depends upon the implementation of compare() method.

HashMap has o(1) only when key overrides equals and hashcode method properly. i.e provided if mappings are distributed uniformly across bucket location. In the real world, you always have collision and HashMap handles collision by using a linked list to store collided elements. This can reduce worst case performance of HashMap up to O(n).

To mitigate the above performance issue, JDK 8 has introduced balanced tree instead of linked list in case of frequent collision in HashMap. It internally switches to balanced tree from linked list if there are more than 8 entries in one bucket  
  


LinkedHashMap also provides a great starting point for creating a LRU Cache object by overriding the removeEldestEntry() method, as shown in the following code snippet.

1. private static final int MAX\_ENTRIES = 100;
2. protected boolean removeEldestEntry(Map.Entry eldest) {
3. return size() > MAX\_ENTRIES;
4. }

It supports O(1) get/put operations. Keys must have [consistent implementations of hashCode() and equals()](http://java.sun.com/javase/6/docs/api/java/lang/Object.html#hashCode%28%29) for this to work.

[Iteration over Map](http://java67.blogspot.sg/2014/05/3-examples-to-loop-map-in-java-foreach.html) is directly proportional to the "capacity" + "size" of HashMap, that's why it's important to set the initial capacity high enough if iteration performance is important.

looping over Map in the case of LinkedHashMap is slightly faster than HashMap because the time required is proportional to size only.

TreeMap and HashMap classes were added in java 1.2 , while LinkedHashMap was added to jdk in java version 1.4

LinkedHashMap implementation differs from HashMap in that it maintains a doubly linked list running through all of its entries.

In the LinkedHasMap implementation, the LinkedHashMap.Entry class extends the HashMap.Entry class, by adding before and after fields. These fields are used to assemble the LinkedHashMap.Entry objects into an independent doubly-linked list that records the insertion order. So, in the LinkedHashMap class, the entry objects are in two distinct chains:

* a singly linked hash chain that is accessed via the main hash array, and
* a separate doubly linked list of all entries that is kept in entry insertion order.

**class LinkedHashMapCustom<K, V>** {

**private** Entry<K,V>[] table;   //Array of Entry.

**private** **int** capacity= 4;  //Initial capacity of HashMap

**private** Entry<K,V> header; //head of the doubly linked list.

**private** Entry<K,V> last; //last of the doubly linked list.

   /\*

    \* before and after are used for maintaining insertion order.

    \*/

**static class Entry<K, V>** {

       K key;

       V value;

       Entry<K,V> next;

**Entry<K,V> before,after;**

**public** Entry(K key, V value, Entry<K,V> next){

**this**.key = key;

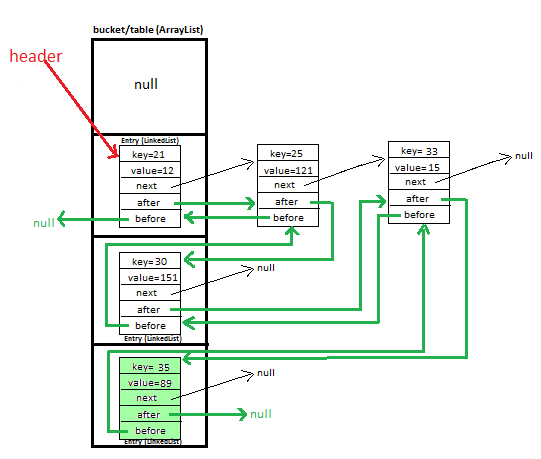
**this**.value = value;

**this**.next = next;

       }

   }

1. HashMap is a map based on [hashing](http://en.wikipedia.org/wiki/Hash_function) of the keys. It supports O(1) get/put operations. Keys must have [consistent implementations of hashCode() and equals()](http://java.sun.com/javase/6/docs/api/java/lang/Object.html#hashCode()) for this to work.
2. LinkedHashMap is very similar to HashMap, but it adds awareness to the order at which items are added (or accessed), so the iteration order is the same as insertion order (or access order, depending on construction parameters).
3. TreeMap is a tree based mapping. Its put/get operations take O(log n) time. It requires items to have some comparison mechanism, either with Comparable or Comparator. The iteration order is determined by this mechanism.



When second element with key=25 and value=121 comes We will calculate hash by using our **hash(K key)** method - in this case it returns

**key/capacity= 25%4= 1.**

**So, 1** will be the **index of bucket** on which **newEntry object** will be stored.

We will go to **1st** index, it contains **entry with key=21**, we will compare two keys(i.e. **compare 21 with 25** by using **equals method**), as **two keys are different** we check whether entry with key=21’s **next is null or not**, **if next is null** we will **put** our **newEntry object** on **next.**

**Additionally, for maintaining insertion order-**

Update **header.after**, it will start pointing to **newEntry object** (i.e make Entry with key=21’s after point to **newEntry object**], and also make  **newEntry object’s** before point to header(Entry with key=21’)

**HASHMAP IMPLEMENTATION**

 simple implementation of HashMaps in Java using an array of a linked list.

When an element is added to hashmap,

Key->calculate hashcode->index.

hashCode (key) % array\_length

Get the linked list at this index calculated above. Store the element in this index.

 The use of a linked list is important because of collisions: you could have two different keys with the same hash code or two different hash codes that map to the same index.

The term **bucket** used here is actually each index of the array. By default HashMap array is of length 16 so there are 16 buckets in a HashMap. Since the array is named table so table[0] is bucket0, table[1] is bucket1 and so on till bucket15.

**class** **HashMapCustom**<K, V> {

**private** Entry<K,V>[] table;   //Array of Entry.

**private** **int** capacity= 4;  //Initial capacity of HashMap

**static class Entry<K, V>** {

       K key;

       V value;

       Entry<K,V> next;

**public** Entry(K key, V value, Entry<K,V> next){

**this**.key = key;

**this**.value = value;

**this**.next = next;

       }

   }

**public** HashMapCustom(){

      table = **new** Entry[capacity];

   }

## Time Complexity

Since different keys can be mapped to the same index, there is a chance of collision. If the number of collisions is very high, the worst case runtime is O(n), where n is the number of keys.  
However, we generally assume a good implementation that keeps collisions to a minimum, in which case the lookup time is O(1).

How HashSet works in java?

**public** **class** **HashSet**<E>

**extends** AbstractSet<E>

**implements** Set<E>, Cloneable, java.io.Serializable

{

**private** **transient** HashMap<E,Object> map;

// Dummy value to associate with an Object in the backing Map

**private** **static** **final** Object PRESENT = **new** Object();

**public** **HashSet**() {

*map =* ***new*** *HashMap<>();*

}

// SOME CODE ,i.e Other methods in Hash Set

**public** **boolean** **add**(E e) {

**return** map.put(e, PRESENT)==**null**;

}

// SOME CODE ,i.e Other methods in Hash Set

}

So , we are achieving uniqueness in Set,internally in java  through HashMap . As we know in HashMap each key is unique . So what we do in the set is that we pass the argument in the add(Elemene E) that is E as a key in the HashMap . Now we need to associate some value to the key , so what Java apis developer did is to pass the Dummy  value that is ( new Object () ) which is referred by Object reference PRESENT .

Now if you see the code of the HashMap put(Key k,Value V) method , you will find something like this  
  
 public V put(K key, V value) {  
//Some code  
}  
  
The main point to notice in above code is that put (key,value) will return  
  
1.  null , if key is unique and added to the map  
2.  Old Value of the key , if key is duplicate  
  
So , in HashSet add() method ,  we check the return value of map.put(key,value) method with null value   
i.e.  
  
   public boolean add(E e) {  
            return map.put(e, PRESENT)==null;  
       }  
  
So , if map.put(key,value) returns null ,then  
map.put(e, PRESENT)==null      will return true and element is added to the HashSet.  
  
  
  
So , if map.put(key,value) returns old value of the key ,then  
map.put(e, PRESENT)==null      will return false and element is  not added to the HashSet .

HashTable is obsolete and the corresponding ConcurrentHashMap class should be used.

HashSet,LinkedhashSet,TreeSet

**Differences Between HashSet, LinkedHashSet and TreeSet In Java :**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **HashSet** | **LinkedHashSet** | **TreeSet** |
| How they work internally? | HashSet uses HashMap internally to store it’s elements. | LinkedHashSet uses  LinkedHashMap internally to store it’s elements. | TreeSet uses TreeMap internally to store it’s elements. |
| Order Of Elements | HashSet doesn’t maintain any order of elements. | LinkedHashSet maintains insertion order of elements. i.e elements are placed as they are inserted. | TreeSet orders the elements according to supplied Comparator. If no comparator is supplied, elements will be placed in their natural ascending order. |
| Performance | HashSet gives better performance than the LinkedHashSet and TreeSet. | The performance of LinkedHashSet is between HashSet and TreeSet. It’s performance is almost similar to HashSet. But slightly in the slower side as it also maintains LinkedList internally to maintain the insertion order of elements. | TreeSet gives less performance than the HashSet and LinkedHashSet as it has to sort the elements after each insertion and removal operations. |
| Insertion, Removal And Retrieval Operations | HashSet gives performance of order O(1) for insertion, removal and retrieval operations. | LinkedHashSet also gives performance of order O(1) for insertion, removal and retrieval operations. | TreeSet gives performance of order O(log(n)) for insertion, removal and retrieval operations. |
| How they compare the elements? | HashSet uses equals() and hashCode() methods to compare the elements and thus removing the possible duplicate elements. | LinkedHashSet also uses equals() and hashCode() methods to compare the elements. | TreeSet uses compare() or compareTo() methods to compare the elements and thus removing the possible duplicate elements. It doesn’t use equals() and hashCode() methods for comparision of elements. |
| Null elements | HashSet allows maximum one null element. | LinkedHashSet also allows maximum one null element. | TreeSet doesn’t allow even a single null element. If you try to insert null element into TreeSet, it throws NullPointerException. |
| Memory Occupation | HashSet requires less memory than LinkedHashSet and TreeSet as it uses only HashMap internally to store its elements. | LinkedHashSet requires more memory than HashSet as it also maintains LinkedList along with HashMap to store its elements. | TreeSet also requires more memory than HashSet as it also maintains Comparator to sort the elements along with the TreeMap. |
| When To Use? | Use HashSet if you don’t want to maintain any order of elements. | Use LinkedHashSet if you want to maintain insertion order of elements. | Use TreeSet if you want to sort the elements according to some Comparator |

COPYONWRITEARRAYLIST vs Arraylist

1) First and foremost difference between CopyOnWriteArrayList and ArrayList in Java is that CopyOnWriteArrayList is a [thread-safe collection](http://javarevisited.blogspot.sg/2011/04/difference-between-concurrenthashmap.html) while ArrayList is not thread-safe and can not be used in multi-threaded environment.

2) Second difference between ArrayList and CopyOnWriteArrayList is that [Iterator of ArrayList is fail-fast](http://javarevisited.blogspot.sg/2012/02/fail-safe-vs-fail-fast-iterator-in-java.html) and throw ConcurrentModificationException once detect any modification in List once iteration begins but Iterator of CopyOnWriteArrayList is fail-safe and doesn't throw ConcurrentModificationException.

3) Third difference between CopyOnWriteArrayList vs ArrayList is that [Iterator](http://javarevisited.blogspot.sg/2011/10/java-iterator-tutorial-example-list.html) of former doesn't support remove operation while Iterator of later supports remove() operation.

As name suggest CopyOnWriteArrayList creates copy of underlying [ArrayList](http://javarevisited.blogspot.sg/2011/06/converting-array-to-arraylist-in-java.html) with every mutation operation e.g. add or set. Normally CopyOnWriteArrayList is very expensive because it involves **costly Array copy** with every write operation but its very efficient if you have a [List](http://javarevisited.blogspot.sg/2012/04/difference-between-list-and-set-in-java.html) where Iteration outnumber mutation e.g. you mostly need to [iterate the ArrayList](http://java67.blogspot.sg/2012/08/how-to-traverse-iterate-or-loop-ArrayList-in-java-example-tutorial.html) and don't modify it too often

CopyOnWriteArrayList is recommended for the concurrent multi-threading environment as it is optimized for multiple concurrent reads and creates copy for the write operation. This was added in Tiger, aka JDK 1.5. It's part of java.util.concurrent package, along with ConcurrentHashMap and BlockingQueue.  
  
4)ArrayList class is present in java.util package , while CopyOnWriteArrayList class is present in java.util.concurrent package.

**5)**ArrayList class was added in java version 1.2 , while CopyOnWriteArrayList class was added in java version 1.5 (or java 5)   
Set Method of list- set(index,value)

Case 1 – when list is an Arraylist – we will get concurrentModificationException

while(it.hasNext()){

System.out.println("list is:"+list);

String str = it.next();

System.out.println(str);

if(str.equals("2"))list.remove("5");

if(str.equals("3"))list.add("3 found");

//below code don't throw ConcurrentModificationException

//because it doesn't change modCount variable of list

if(str.equals("4")) **list.set(1, "4");**

}

final void checkForComodification() {

if (modCount != expectedModCount)

throw new ConcurrentModificationException();

}

Here modCount is the ArrayList variable that holds the modification count and every time we use add, remove or trimToSize method, it increments. expectedModCount is the iterator variable that is initialized when we create iterator with same value as modCount. This explains why we don’t get exception if we use set method to replace any existing element.

So basically iterat`or throws ConcurrentModificationException if list size is changed.

If we change the implementation to CopyOnWriteArrayList, then we don’t get any exception and below is the output produced.

list is:[1, 2, 3, 4, 5]

1

list is:[1, 2, 3, 4, 5]

2

list is:[1, 2, 3, 4]

3

list is:[1, 2, 3, 4, 3 found]

4

list is:[1, 4, 3, 4, 3 found]

5

**ConcurrentHashMap**

The constructor of ConcurrentHashMap looks like this :

**public ConcurrentHashMap *(int initialCapacity, float loadFactor, int concurrencyLevel)***

So the above line  creates a new, empty map with the specified initial capacity, load factor and concurrency level.

where,

**initialCapacity** - the initial capacity. The implementation performs *internal sizing to accommodate this many elements.*

**concurrencyLevel** - the estimated number of concurrently updating threads. The implementation performs *internal sizing to try to accommodate this many threads.*

static final int DEFAULT\_INITIAL\_CAPACITY = 16;

static final int DEFAULT\_CONCURRENCY\_LEVEL = 16;

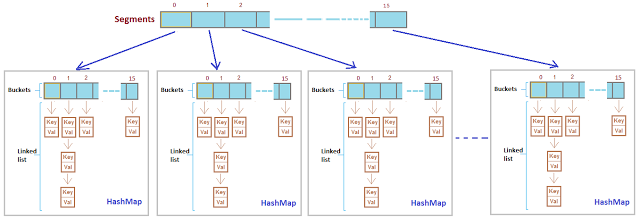
initial capacity parameter and concurrency level parameters of ConcurrentHashMap constructor (or Object) are set to 16 by default.

Thus, instead of a map wide lock, ConcurrentHashMap maintains a list of 16 locks by default ( number of locks equal to the initial capacity , which is by default 16

**Imagine a scenario where we have frequent reads(get) and less writes(put) and need thread safety,**  
  
**Can we use Hashtable in this scenario?**  
No. Hashtableis thread safe but give poor performance in case of multiple thread reading from hashtable because all methods of Hashtable including get() method is synchronized and due to which invocation to any method has to wait until any other thread working on hashtable complete its operation(get, put etc).

Unlike hashtable, we perform any sort of operation ( update ,delete ,read ,create) without locking on entire map in ConcurrentHashMap.

ConcurrentHashMap added one Array on top of it and each index of this additional array represents complete HashMap. Additional array is called Segment in ConcurrentHashMap.



In concurrentHashMap, the **difference lies in internal structure to store these key-value pairs**. ConcurrentHashMap has an addition concept of segments. It will be easier to understand it you think of one segment equal to one HashMap [conceptually]. A concurrentHashMap is divided into number of segments [default 16] on initialization. ConcurrentHashMap allows similar number (16) of threads to access these segments concurrently so that each thread work on a specific segment during high concurrency.

Ideally, you should choose a value to accommodate as many threads as will ever concurrently modify the table. Using a significantly higher value than you need can waste space and time, and a significantly lower value can lead to thread contention

ConcurrentHashMap synchronizes or locks on the certain portion of the Map . To optimize  
   the performance of ConcurrentHashMap , Map is divided into different partitions depending  
   upon the Concurrency level . So that we do not need to synchronize the whole Map Object.

  ConcurrentHashMap does not allow NULL values . So the key can not be null in  
     ConcurrentHashMap .While In HashMap there can only be one null key .

ConcurrentHashMap provides better Performance by replacing the Hashtable's map wide lock to Segment level lock.  
  
Hashtable is not efficient because it uses map wide lock, it means lock is applied on map object itself,

Ref- <http://javabypatel.blogspot.in/2016/09/concurrenthashmap-interview-questions.html>

### Question 1.  What is the need of ConcurrentHashMap when there is HashMap and Hashtable already present?

**Performance and Thread safety are 2 parameter on which ConcurrentHashMap is focused.**

**Imagine a scenario where we have frequent reads(get) and less writes(put) and need thread safety,**  
  
**Can we use Hashtable in this scenario?**  
No. Hashtableis thread safe but give poor performance in case of multiple thread reading from hashtable because all methods of Hashtable including get() method is synchronized and due to which invokation to any method has to wait until any other thread working on hashtable complete its operation(get, put etc).  
  
**Can we use HashMap in this scenario?**  
No. Hashmap will solve performance issue by giving parallel access to multiple threads reading hashmap simultaneously.  
But Hashmap is not thread safe, so what will happen if one thread tries to put data and requires Rehashing and at same time other thread tries to read data from hashmap, It will go in infinite loop.  
**Infinite loop problem discussed in detail:** [**Infinite loop in HashMap**](http://javabypatel.blogspot.in/2016/01/infinite-loop-in-hashmap.html)  
  
  
**ConcurrentHashMap combines good features of hashmap and hashtable and solves performance and thread safety problem nicely.**

**Difference b/w concurrenthashmap and collections.synchronisedmap**

The main difference between these two is that ConcurrentHashMap will lock only portion of the data which are being updated while other portion of data can be accessed by other threads. However, Collections.synchronizedMap() will lock all the data while updating, other threads can only access the data when the lock is released.

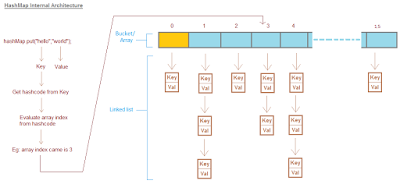
Also one other difference is that ConcurrentHashMap will not preserve the order of elements in the Map passed in. It is similar to HashMap when storing data. There is no guarantee that the element order is preserved. While Collections.synchronizedMap() will preserve the elements order of the Map passed in. For example, if you pass a TreeMap to ConcurrentHashMap, the elements order in the ConcurrentHashMap may not be the same as the order in the TreeMap, but Collections.synchronizedMap() will preserve the order.

Furthermore, ConcurrentHashMap can guarantee that there is no ConcurrentModificationException thrown while one thread is updating the map and another thread is traversing the iterator obtained from the map. However, Collections.synchronizedMap() is not guaranteed on this.

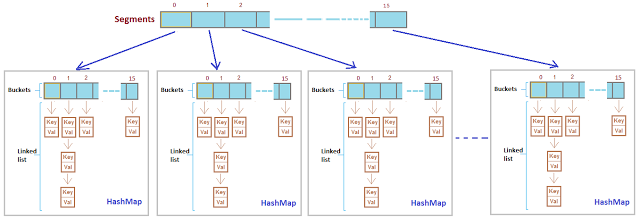
[ConcurrentHashMap vs Collections.synchronizedMap() | Pixelstech.net](https://www.pixelstech.net/article/1394026282-ConcurrentHashMap-vs-Collections-synchronizedMap%28%29)

### Question 2.  HashMap and Hashtable uses Array and Linkedlist as datastructure to store data, How is it different in ConcurrentHashMap?

If you are not familiar with HashMap and Hashtable, Please go through it first:  
[How HashMap works](http://javabypatel.blogspot.in/2015/09/hashmap-data-structure-and-hashcode.html)   
  
Below diagram shows how hashtable/hashmap look like,

[](https://2.bp.blogspot.com/-9JEpgm-_nVw/V-pS4HnlajI/AAAAAAAABPE/p3t6GfbM4hsYrJvRhbs0u2Y8yrCu8-4tACLcB/s1600/hashmap-internal-structure.png)

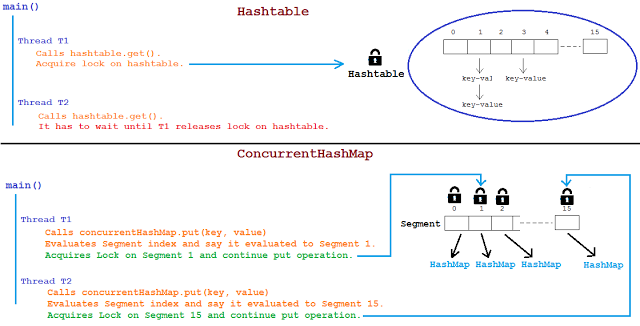
ConcurrentHashMap added one Array on top of it and each index of this additional array represents complete HashMap. Additional array is called Segment in ConcurrentHashMap.  
  
Architecture of ConcurrentHashMap looks like below,

[](https://4.bp.blogspot.com/-1W2vuBYf740/V-pnAFenccI/AAAAAAAABPU/lKSaVIfAzpMmrpRanTydHXSAA5uhC_p7wCLcB/s1600/consurrenthashmap-internal-structure.png)

**Putting key-value pair:**   
  
1. Putting key-value pair in ConcurrentHashMap requires first identifying exact index in   
    Segment array.   
    (Once Segment array index is identified, Now flow will be exactly same as putting the data in   
    hashmap/hashtable.)   
2. After identifying index in Segment array, next task is to identify index of internal bucket/array   
    present in internal hashmap as shown in figure above.   
 3. After identying bucket(internal array index), iterate key-value pairs and check each key with key   
    to store, wherever match is found replace stored value with value to store.  
    If there is no match, store key-value pair at the last of list.  
  
  
**Getting key-value pair:**  
  
1. Getting key-value pair in ConcurrentHashMap requires first identifying exact index in   
    Segment array.   
    (Once Segment array index is identified, Now flow will be exactly same as getting the data from   
    hashmap/hashtable.)   
2. After identifying index in Segment array, next task is to identify index of internal bucket/array   
    present in internal hashmap as shown in figure above.   
3. After identying bucket(internal array index), iterate key-value pairs and match each key with   
    given key, wherever match is found return value stored against key.  
    If there is no match, return null.

### Question 3.  How ConcurrentHashMap is efficient in terms of Performance and Thread safety?

ConcurrentHashMap provides better Performance by replacing the Hashtable's map wide lock to Segment level lock.  
  
Hashtable is not efficient beacause it uses map wide lock, it means lock is applied on map object itself,   
  
**So if 2 threads tries to call hashtable.get(key),**  
Thread T1 calls to get() method will acquire a lock on hashtable object and then execute get() method. (Lock is on complete 'hashtable object')  
  
Now if Thread T2 calls hashtable.get(key) method, then it will also try to acquire lock on hashtable object, but T2 will not able to acquire lock as lock on 'hashtable' is currently held by T1,   
  
So T2 waits until T1 finishes get() operation and release lock on hashtable object.

[](https://3.bp.blogspot.com/-aD_Z4JJtrhc/V-rLvoLHVaI/AAAAAAAABPk/M1VpuzyvZcY56vjBlnBGaczd-HifRuwvwCEw/s1600/hashtable-vs-concurrenthashmap-lock.png)

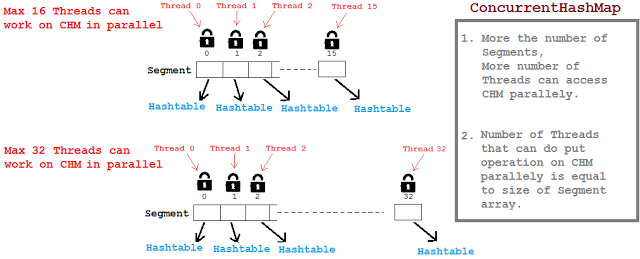
ConcurrentHashMap works bit different here and instead of locking complete map object it Locks per Segment.   
It means instead of single map wide lock, it has multiple Segment level lock.  
  
So 2 Threads can execute put operation simultaneously by acquiring lock on different Segments.  
  
Thread T1 calls concurrentHashMap.put(key, value), It acquires lock on say Segment 1 and invokes put method.  
Thread T2 calls concurrentHashMap.put(key, value), It acquires lock on say Segment 4 and invokes put method.  
  
Both threads doesn't interfere with each other and both can proceed simultaneously as they are working on separate Segment locks.  
  
**This is how ConcurrentHashMap improves Performance and provide Thread safety as well.**

### Question 4.  Can multiple threads read and write from same or different Segments of ConcurrentHashMap simultaneously?

**Read Operation: get(key)**   
**Same Segment/Different Segment : Yes.**   
Two threads T1 and T2 both can simultaneously read data from same Segment or different Segment of CHM simultaneously without blocking each other.  
  
  
**Write Operation: put(key, value)**   
**Different Segment :Yes**   
Multiple threads can write data to different Segment of CHM simultaneously without blocking each other.  
**Same Segment : No**    
Multiple threads CANNOT write data to same Segment of CHM simultaneously and need to wait for one thread to come write operation and then only other write operation can be proceed.   
     
  
**Read-Write Operation: get and put**  
**Say T1 is writing data in Segment 1 and T2 is reading data from same Segment 1, can read be allowed while writing operation is going on?**  
**YES.**  
Both operation that is T1 writing and T2 reading can be done parallely**.**  
  
**What data will T2 read if T1 is updating same data?**Retrieval operations (including get) generally do not block, so may overlap with update operations (including put and remove).   
Latest updated value present will be returned by get operation that is value updated by most recently completed update operations will be returned.  
  
**Note: Get operations are lock free and can be performed simulateneously irrespective of other thread writing data on same or different Segment of CHM.**

### Question 5.  What is the default size of Segment array? how it is tuned? What is ConcurrenyLevel in case of CHM?

**Default size of Segment array is 16.**  
   
ConcurrentHashMap differes from Hashtable in terms of Performance by introducing Segment array.  
Each index of Segment array is guarded by a lock for put operation.

[](https://4.bp.blogspot.com/-icREX-PLyG0/V_YsC15kxrI/AAAAAAAABRc/CDLvHf-xyRkeu8U_UaZbfos55t0E5NvAgCLcB/s1600/concurrency-level-concurrenthashmap.png)

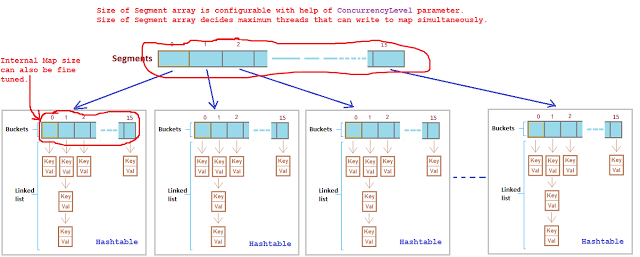
Threads working on separate Segments index doesn't affect each other.   
By default Segments array size is 16, So maximum 16 threads can simultaneously put data in map considering each thread is working on separate Segment array index.

### How Segment array size is tuned?

Segment size decides the number of Threads that can paralley write to a map.  
Segment array size is configured using ConcurrencyLevel parameter as shown below,

[?](http://javabypatel.blogspot.in/2016/09/concurrenthashmap-interview-questions.html)

|  |  |
| --- | --- |
| 1 | ConcurrentHashMap m = new ConcurrentHashMap(initialCapacity, loadFactor, concurrencyLevel) |

[](https://4.bp.blogspot.com/-0HRMmJEuDys/V_YtFEKQ8BI/AAAAAAAABRk/Z195nyfoOwgwBc53mHn9pkVsFAm4ORYMQCLcB/s1600/consurrenthashmap-concurrency-level.png)

**It takes 3 parameters,**  
  
**Example:**

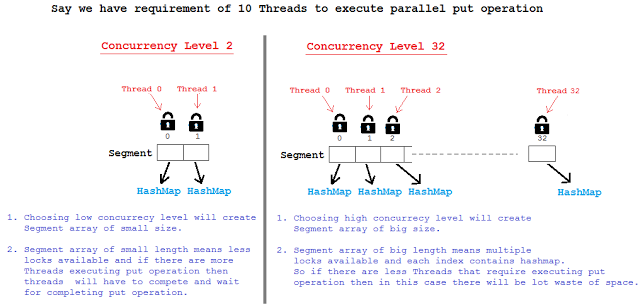
[?](http://javabypatel.blogspot.in/2016/09/concurrenthashmap-interview-questions.html)

|  |  |
| --- | --- |
| 1 | ConcurrentHashMap m = new ConcurrentHashMap(200 , 0.75f, 10); |

**Initial capacity** is 200, it means CHM make sure it has space for adding 200 key-value pairs after creation.  
  
**Load factor** is 0.75, it means when average number of elements per map exceeds 150 (intital capacity \* load factor = 200 \* 0.75 = 150) at that time map size will be increased and existing items in map are rehashed to put in new larger size map.  
For more details on Load Factor: [**Load factor in Map**](http://javabypatel.blogspot.in/2015/10/what-is-load-factor-and-rehashing-in-hashmap.html)  
  
**Concurrency level** is 10, it means at any given point of time Segment array size will be 10 or greater than 10, so that 10 threads can able to parallely write to a map.

### Question 6.  What will happen if the size of Segment array is too small or too large?

Choosing correct **ConcurrencyLevel** is very important because ConcurrencyLevel decides what will be the size of Segment array.  
  
Segment array size will decide how many parallel Threads will be able to execute put operation on map parallely.

[](https://2.bp.blogspot.com/-Dpl-QWxn8jM/V_CbnodjnOI/AAAAAAAABQc/TZwMkByf6RcEK6iC6BTcklkTnNIeiirnACLcB/s1600/concurrency-level-too-small-large.png)

So Segment array size should not be too big or should not be too small because,   
Using a significantly higher value than we will waste space and time, and a significantly lower value can lead to thread competition.

### If we choose ConcurrenyLevel as 10 then what will be size of Segment array? Is Segment array size exactly same as concurrenyLevel? If No, then how is the Segment array size calculated?

Segment array size is calculated based on concurrenyLevel specified but it doesn't mean it will be exactly same as concurrenyLevel.  
  
**If concurrenyLevel is 10 then Segment array size will be 16.**   
  
Segment array size = 2 to the power x, where result should be >= concurrenyLevel(in our case it is 10)  
Segment array size = 2 to the power x >= 10  
  
Segment array size = 2 ^ 1 = 2   >= 10 (False)  
Segment array size = 2 ^ 2 = 4   >= 10 (False)  
Segment array size = 2 ^ 3 = 8   >= 10 (False)  
Segment array size = 2 ^ 4 = 16 >= 10 **(True)**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

### Question 7.  What is HashEntry array and how is the size of HashEntry decided?

Default initial capacity of CHM is 16. It means CHM make sure there is sufficient space to accomodate 16 key-value pairs after CHM is created.

**In HashMap the bucket/array is of class Entry[] and in CHM the array is of class HashEntry[].**

[?](http://javabypatel.blogspot.in/2016/09/concurrenthashmap-interview-questions.html)

|  |  |
| --- | --- |
|  | static final class Segment<K,V> extends ReentrantLock implements Serializable {   transient volatile HashEntry<K,V>[] table;  }    static final class HashEntry<K,V> {   final int hash;   final K key;   volatile V value;   volatile HashEntry<K,V> next;  }  HashEntry[] array size  =   2 ^ x   >=  (initialCapacity / concurrenyLevel)  **Eg: ConcurrentHashMap(32,   0.75f,   4);**  HashEntry[] array size  =  2 ^ 1 = 2   >=  8(32/4) (False) HashEntry[] array size  =  2 ^ 2 = 4   >=  8 (False) HashEntry[] array size  =  2 ^ 3 = 8   >=  8 **(True)**  HashEntry[] array size =8**.** **It means there will always be capacity of 8 key-value pairs that can be put in CHM after its creation.**  **\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*** Question 8.  How does ConcurrentHashMap handle rehashing while another thread is still writing on another segment/partition? For understanding rehashing, please understand Load Factor first.  **Load Factor** is a measure, which decides when exactly to increase the HashMap/CHM capacity(buckets) to maintain get and put operation complexity of O(1).  Default load factor of Hashmap/CHM is 0.75f (i.e 75% of current map size).  **For more details on Load factor, Please refer:** [**Load Factor in HashMap**](http://javabypatel.blogspot.in/2015/10/what-is-load-factor-and-rehashing-in-hashmap.html)  [https://3.bp.blogspot.com/-eHGD8zneRbI/VhNYVf6_AaI/AAAAAAAAAd8/BaP2VrEfD7QEnLICoHoOl27LZoUNzE7ywCPcB/s640/hashmapRehashing.png](https://3.bp.blogspot.com/-eHGD8zneRbI/VhNYVf6_AaI/AAAAAAAAAd8/BaP2VrEfD7QEnLICoHoOl27LZoUNzE7ywCPcB/s1600/hashmapRehashing.png)  **In CHM, Every segment is separately rehashed so there is no collision between Thread 1 writing to Segment index 2 and Thread 2 writing to Segment index 5.**  **Example:** If say Thread 1 which is putting data in Segment[] array index 2 finds that HashEntry[] array needs to be rehashed due to exceed Load factor capacity then it will rehash HashEntry[] array present at Segment[] array index 2 only. HashEntry[] array at other Segment indexes will still be intact, unaffected and continue to serve put and get request parallely. |

**Can multiple threads read from the Hashtable concurrently ?**  
  
No multiple threads can not read simultaneously from Hashtable. Reason, the get() method of  Hashtable is synchronized. As a result , at a time only one thread can access the get() method .  
It is possible to achieve full  concurrency for reads (all the threads read at the same time) in  ConcurrentHashMap by using volatile keyword.

Count number of words in a string

**static** **int** **wordcount**(String s)

{

**char** ch[]= **new** **char**[s.length()]; //in string especially we have to mention the () after length

**for**(i=**0**;i<s.length();i++)

{

ch[i]= s.charAt(i);

**if**( ((i>**0**)&&(ch[i]!=' ')&&(ch[i-**1**]==' ')) || ((ch[**0**]!=' ')&&(i==**0**)) )

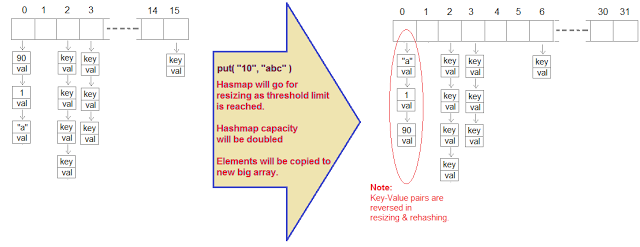
c++; }

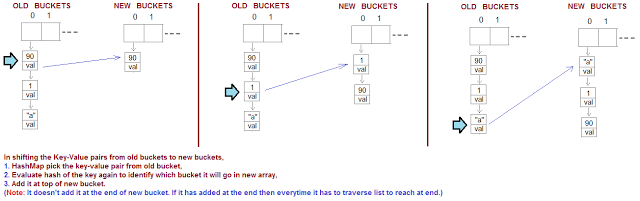
**return** c;

}

### Why HashMap should not be used in multi threaded environment? Can it cause infinite loop as well? When get method go to infinite loop in HashMap?

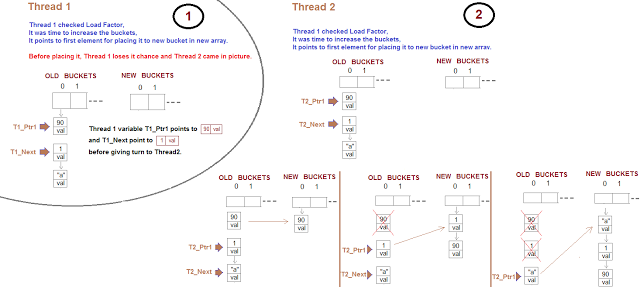
Default capacity of HashMap is 16 and Load factor is 0.75, which means HashMap will double its capacity when 12thKey-Value pair enters in map (**16 \* 0.75 = 12**).

[](http://4.bp.blogspot.com/-xnjGgQ1LP80/VolJc5dSUpI/AAAAAAAAA14/L0OlEphMJwA/s1600/hashmap-infinite-loop.png)

[](http://1.bp.blogspot.com/-jOf-vP71f3U/VolSr5kqAvI/AAAAAAAAA2g/KSQyZk6oj_4/s1600/hashmap-infinite-loop-working.png)

### What will happen when 2 Threads try to put 12th Key-Value pair in HashMap?

When 2 Thread tries to access HashMap simultaneously, then you may encounter infinite Loop.  
Let see how it happens,  
Thread 1 and Thread 2 tries to put 12th key-value pair.  
  
**Thread 1 got execution chance.**   
**1.** Thread 1 tries to put 12th key-value pair,   
**2.** Thread 1 founds that Threshold limit is reached and it creates new Buckets of increased capacity.  
    So map's capacity is increased from 16 to 32.  
**3.** Thread 1 now transfers all existing key-value pairs to new buckets.  
**4.** Thread 1 points to first key-value pair and next(second) key-value pair to start transfer process.    
  
**Thread 1 after pointing to key-value pairs and before starting the transfer process, loose the control and Thread 2 got a chance for execution.**  
**Thread 2 got execution chance.**   
**1.** Thread 2 tries to put 12th key-value pair,   
**2.** Thread 2 founds that Threshold limit is reached and it creates new Buckets of increased capacity.  
    So map's capacity is increased from 16 to 32.  
**3.** Thread 2 now transfers all existing key-value pairs to new buckets.  
**4.** Thread 2 points to first key-value pair and next(second) key-value pair to start transfer process.    
**5.** While transferring key-value pairs from old buckets to new buckets, key-value pairs will be   
    reversed in new buckets because hashmap will add key-value pairs at the start and not at the end.   
    Hashmap adds new key-value pairs at start to avoid traversing linked list every time and keep   
    constant performance.  
  
**6.** Thread 2 will transfer all key-value pairs from old buckets to new buckets and Thread 1 will get   
    chance for execution.  
 

[](http://4.bp.blogspot.com/-5XAvDdEKOxE/VolI3CPVvyI/AAAAAAAAA1w/EMaJZfDth-Q/s1600/hashmap-infinite-loop-in-multithreading-resizing-Part1.png)

**Thread 1 got execution chance.**   
**1.** Thread 1 before leaving control was pointing to first element and next element of old bucket.

**According to Thread 1, Next element of (1, val) is ("a", val) but actually it is not because Thread 2 has changed Next element of (1, val) to (90, val).**

**2.**  Now when Thread 1 started putting key-value pairs from old bucket to new bucket,   
     It successfully puts (90, val) and (1, val) in new Bucket.  
**3.** When it tries to add next element of (1, val) which is (90, val) into new Bucket, it will end up in   
     infinite loop.

Since TreeMaps are sorted by keys, the object for key has to be able to compare with each other, that's why it has to implement Comparable interface

A transient variable plays an important role in preventing an object from being serialized