**Spring-JDBC:**

\*Spring-JDBC provides the JdbcTemplate class.

\*That removes plumbing code and helps you concentrate on the SQL query and parameters. You just need to configure it with a DataSource, and you can then write code like this:

int nbRows = jdbcTemplate.queryForObject("select count(1) from person", Integer.class);

Person p = jdbcTemplate.queryForObject("select first, last from person where id=?",

rs -> new Person(rs.getString(1), rs.getString(2)), 134561351656L);

\*Spring-JDBC also provides a JdbcDaoSupport, that you can extend to develop your DAO.

\*It basically defines 2 properties: a DataSource and a JdbcTemplate that both can be used to implement the DAO methods.

\*It also provides an exceptions translator from SQL exceptions to spring DataAccessExceptions.

**Spring-ORM:**

Spring-ORM is an umbrella module that covers many persistence technologies, namely JPA, JDO, Hibernate and iBatis. For each of these technologies, Spring provides integration classes so that each technology can be used following Spring principles of configuration, and smoothly integrates with Spring transaction management.

For each technology, the configuration basically consists in injecting a DataSource (DBConfig) bean into some kind of SessionFactory or EntityManagerFactory etc. bean. For pure JDBC, there's no need for such integration classes (apart from JdbcTemplate), as JDBC only relies on a DataSource.

If you plan to use an ORM like JPA or Hibernate, you will not need spring-jdbc, but only this module.

<https://stackoverflow.com/questions/24990400/spring-dao-vs-spring-orm-vs-spring-jdbc>

**TransactionManagement:**

An important aspect of transaction management is defining the right transaction boundary for e.g when should a transaction start,when should it end,when data should be committed in DB and when it should be rolled back (in the case of exception).

1. Programmatically manage by writing custom code as below

i.Legacy implementation implementing directly on code

**2. Use Spring to manage transaction**

**i.  Programmatic transaction management :**

I.Using TransactionTemplet TransactionManager

II.Using PlatformTransactionManager

ii. **Declarative transaction management**

**1.Programmatically**

**Feature:**

1.The scope of the transaction is very clear in the code.

Drawback:

* It's repetitive and error prone.
* Any error can have a very high impact.
* A lot of boilerplate needs to be written and if you want to call another method from this method then again you need to manage it in the code.

Choosing between **Programmatic**and **Declarative Transaction Management**:

* Programmatic transaction management is good only if you have a small number of transactional operations. (Most of the times, this is not the case.)
* Transaction name can be explicitly set only using Programmatic transaction management.
* Programmatic transaction management should be used when you want explicit control over managing transactions.
* On the other hand, if your application has numerous transactional operations, declarative transaction management is worthwhile.
* Declarative Transaction management keeps transaction management out of business logic, and is not difficult to configure.

**II.Declarative Transaction (Usually used almost in all scenarios of any web application)**

**Steps1:**

* 1. Define a transaction manager in your spring application context xml file.

<bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager"/>

2.Turn on support for transaction annotations by adding below entry to your spring application context XML file.

<tx:annotation-driven transaction-manager="txManager"/>

Or use @EnableTransactionManager in configuration class

@Configuration

@EnableTransactionManager

Public Class ServiceClass{

}

\* Spring recommends that you only annotate concrete classes (and methods of concrete classes) with @Transactional annotation as compared to annotating interfaces.

The reason for this is if you put an annotation on the Interface Level and if you are using class-based proxies (proxy-target-class="true") or the weaving-based aspect (mode="aspectj"), then the transaction settings are not recognized by the proxying and weaving infrastructure .i.e Transactional behaviour will not be applied.

**Polymorphism:** Polymorphism is the ability of an object to take on many forms. The most common use of polymorphismin OOP occurs when a parent class reference is used to refer to a child class object. Any Java object that can pass more than one IS-A test is considered to be polymorphic.

**Encapsulation: Encapsulation** is one of the four fundamental OOP concepts. The other three are inheritance, polymorphism, and abstraction. **Encapsulation in Java** **is a mechanism of wrapping the data** (variables) and code acting on the data (methods) together as a single unit.

Eg: **Class** binding of data (attributes and methods) and the code in the form of a class and that's all.

Think objects, for each employee in an employee class, here employee object is the capsule that holds data and methods specific to each employee).

**Abstraction:Abstraction**is the process of hiding the implementation details from the user and providing a common interface for them

Abstract class and interface provide abstraction in java.

**Dependency Injection:**In Spring frameowork, Dependency Injection (DI) design pattern is used to define the object dependencies between each other.

**SetterInjection:**

<bean id="OutputHelper" class="com.mkyong.output.OutputHelper">

<property name="outputGenerator">

<ref bean="CsvOutputGenerator" />

</property>

</bean>

<bean id="CsvOutputGenerator" class="com.mkyong.output.impl.CsvOutputGenerator" />

<bean id="JsonOutputGenerator" class="com.mkyong.output.impl.JsonOutputGenerator" />

**ConstructorInjection:**

<bean id="OutputHelper" class="com.mkyong.output.OutputHelper">

<constructor-arg>

<bean class="com.mkyong.output.impl.CsvOutputGenerator" />

</constructor-arg>

</bean>

<bean id="CsvOutputGenerator" class="com.mkyong.output.impl.CsvOutputGenerator" />

<bean id="JsonOutputGenerator" class="com.mkyong.output.impl.JsonOutputGenerator" />

Method Overloading:

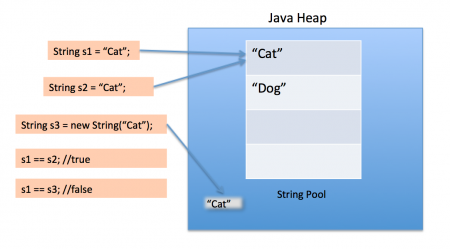
Method Overriding:

String Pool:

String pool (String intern pool) is a special storage area in Java heap.

The hashcode of string is frequently used in Java.

As the name suggests, **String Pool in java** is a pool of Strings stored in [**Java Heap Memory**](http://www.journaldev.com/4098/java-heap-space-vs-stack-memory). We know that String is special class in java and we can create String object using new operator as well as providing values in double quotes as literals

. 

[**String is immutable in Java**](http://www.journaldev.com/802/string-immutable-final-java)

s1 == s2 :true

s1 == s3 :false

Sometimes in java interview, you will be asked question around String pool. For example, how many string is getting created in below statement;

String str = new String("Cat");

In above statement, either 1 or 2 string will be created. If there is already a string literal “Cat” in the pool, then only one string “str” will be created in the pool. If there is no string literal “Cat” in the pool, then it will be first created in the pool and then in the heap space, so total 2 string objects will be created.

Why String is immutable?

**In a HashMap. Being immutable guarantees that hashcode will always the same, so that it can be cashed without worrying the changes.That means, there is no need to calculate hashcode every time it is used. This is more efficient.**

String is widely used as parameter for many java classes,  e.g. network connection, opening files, etc. Were String not immutable, a connection or file would be changed and lead to serious security threat.

hashCode():

The hashcode of a Java Object is simply a number, it is 32-bit signed int, that allows an object to be managed by a hash-based data structure. We know that hash code is an unique id number allocated to an object by JVM.

Hash code is not an unique number for an object.

If two objects are equals then these two objects should return same hash code.

So we have to implement hashcode() method of a class in such way that if two objects are equals, ie compared by equal() method of that class, then those two objects must return same hash code. If you are overriding hashCode() you need to override equals method also.

package com.java2novice.algos;

import java.util.HashMap;

public class MyHashcodeImpl {

    public static void main(String a[]){

        HashMap<Price, String> hm = new HashMap<Price, String>();

        hm.put(new Price("Banana", 20), "Banana");

        hm.put(new Price("Apple", 40), "Apple");

        hm.put(new Price("Orange", 30), "Orange");

        //creating new object to use as key to get value

        Price key = new Price("Banana", 20);

        System.out.println("Hashcode of the key: "+key.hashCode());

        System.out.println("Value from map: "+hm.get(key));

    }

}

class Price{

    private String item;

    private int price;

    public Price(String itm, int pr){

        this.item = itm;

        this.price = pr;

    }

    public int hashCode(){

        System.out.println("In hashcode");

        int hashcode = 0;

        hashcode = price\*20;

        hashcode += item.hashCode();

        return hashcode;

    }

    public boolean equals(Object obj){

        System.out.println("In equals");

        if (obj instanceof Price) {

            Price pp = (Price) obj;

            return (pp.item.equals(this.item) && pp.price == this.price);

        } else {

            return false;

        }

    }

    public String getItem() {

        return item;

    }

    public void setItem(String item) {

        this.item = item;

    }

    public int getPrice() {

        return price;

    }

    public void setPrice(int price) {

        this.price = price;

    }

    public String toString(){

        return "item: "+item+"  price: "+price;

    }

}

equals():

Collections:

Hash Table:

Hash Map:

\*Loadfactor of hashmap is if reach the size of 75% of it .It will double the size of the hashmap by recomputing the existing hashcode.

\* HashMaps are much faster for retreiving data than arrays and linked lists.

\*It will not follow insertion order.

\*

Hash Set:

Linked HashSet:

Linked HashMap:

List:

Map:

Pojo & beans:

Constructor Injection:

Setter Injection:

SpringAOP:

SpringDAO

SpringTransactionManagment:

Entity:

Hibernate:

**Session Factory:** A Session Factory is what we use to grab individual Sessions so that we can talk to our database. It uses the Factory design pattern, which essentially means it’s sole purpose is to dish our Sessions whenever you ask for them.

**Session:** A Session is what we use to create open our initial transaction, perform some kind of database operation, and then commit the transaction (or roll it back if there was a failure).To inte

**Spring-ORM:**spring-orm dependency for Spring ORM support, it’s required for hibernate integration in our spring project.

For Spring and Hibernate 3 integration, **Spring ORM** provides two classes – **org.springframework.orm.hibernate3.LocalSessionFactoryBean** when hibernate mappings are XML based and **org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean** for annotations based mapping.

**Hibenrate till 3.X we used to have this type sessionfactory instation**

SessionFactory sessionFactory = new Configuration().configure().buildSessionFactory();

Session session = sessionFactory.openSession();

**In Hibernate 4.X the Configuration class’ buildSessionFactory() method is deprecated and it recommends developers to use the buildSessionFactory(ServiceRegistry) instead. Here is the new recommended code snippet that builds the SessionFactory based on a ServiceRegistry and obtains the Session:**

Configuration configuration = new Configuration().configure();

ServiceRegistryBuilder registry = new ServiceRegistryBuilder();

registry.applySettings(configuration.getProperties());

ServiceRegistry serviceRegistry = registry.buildServiceRegistry();

SessionFactory sessionFactory = configuration.buildSessionFactory(serviceRegistry);

Session session = sessionFactory.openSession();

**Sample SessionFactory Class:**

package net.codejava.hibernate;

import org.hibernate.SessionFactory;

import org.hibernate.boot.registry.StandardServiceRegistryBuilder;

import org.hibernate.cfg.Configuration;

import org.hibernate.service.ServiceRegistry;

public class HibernateUtil {

    private static SessionFactory sessionFactory;

    public static SessionFactory getSessionFactory() {

        if (sessionFactory == null) {

            // loads configuration and mappings

            Configuration configuration = new Configuration().configure();

            ServiceRegistry serviceRegistry

                = new StandardServiceRegistryBuilder()

                    .applySettings(configuration.getProperties()).build();

            // builds a session factory from the service registry

            sessionFactory = configuration.buildSessionFactory(serviceRegistry);

        }

        return sessionFactory;

    }

}

**HashMap InternalImplementation:**

I believe that it's going to be more helpful if I explain HashMaps in English.

**What is a HashMap?**

A HashMap is a data structure that is able to map certain keys to certain values. The keys and values could be anything. For example, if I were making a game, I might link every username to a friends list, represented by a List of Strings.

**Why use a HashMap?**

HashMaps are much faster for retreiving data than arrays and linked lists. A sorted array could find a particular value in O(log n) with binary search. However, a HashMap can check if it contains a particular key in O(1). All keys must be unique.

**How do HashMaps work?**

HashMaps use an array in the background. Each element in the array is another data structure (usually a linked list or binary search tree). The HashMap uses a function on the key to determine where to place the key's value in the array. For example, if my HashMap accepts Strings...possible hash functions can be:

A. Return the ASCII value of the first letter.

B. Return the sum of the ASCII values of every character in the String.

C. Return the ASCII value of the last character in the String.

The value returned will determine the index in which the value goes into the array.

**But Wait! There's a Problem!**

It is possible that the returned value will be outside of the array's bounds. Therefore, we are supposed to mod the returned value by the arrays length.

return Math.abs(number%hashMapArray.length);

**Collisions:**

Isn't it possible that multiple keys will make the hash function generate the same index? Yes. For example, if we used the first hash function shown above in a hash map of strings...any two Strings that start with the same letter will be given the same array index.

This is called a collision.

How do we handle collisions?

One collision handling technique is called Chaining. Since every element in the array is a linked list (or similar data structure), multiple keys that have the same hash value will be placed in the same linked list or "bucket". Afterwards, the hash map is able to retrieve values by calculating the hash code with the hash function, and searching the particular linked list to see if it contains a value with the same key.

A good hash function must be written to avoid collisions.

**Advantages to Chaining:**

-Array cannot overflow

-Data can be easily removed

**Disadvantages to Chaining:**

-May suffer a performance hit if buckets contain very long linked lists.

The total number of entries to the number of buckets is called the load factor. If the load factor is too low, a lot of space is wasted. If the load factor is too high, the advantage of hashing is lost. A good compromise on load factor is .75

**BinarySearch:**

\*Every iteration eliminates half of the remaining possibilities. This makes binary searches very efficient - even for large collections.

\*Binary search requires a sorted collection. Also, binary searching can only be applied to a collection that allows random access (indexing).

**package** algorithams;

**public** **class** BinarySearch {

**public** **static** **int** binarySearch(**int**[] array,**int** key){

**int** start = 0;

**int** end=array.length-1;

**while**(start<=end){

**int** mid= (start+end)/2;

**if**(key==array[mid]){

System.***out***.println("Check"+mid);

**return** mid;

}

**if**(key<array[mid]){

end = mid-1;

}**else** {

start = mid+1;

}

}

System.***out***.println("not found");

**return** -1;

}

**public** **static** **void** main(String[] args) {

**int** [] array1 = {1,5,10,32,56};

System.***out***.println(*binarySearch*(array1,56));

}

}

**SpringAOP:**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.

AOP breaks the program logic into distinct parts (called concerns). It is used to increase modularity by cross-cutting concerns.

A cross-cutting concern is a concern that can affect the whole application and should be centralized in one location in code as possible, such as transaction management, authentication, logging, security etc.

@Before,@After,@ThrowsException,@Around

**ThreadLifeCycle:**

 we can create a [java thread](http://www.journaldev.com/1016/java-thread-example) class by implementing **Runnable interface** or by extending **Thread** class, but to start a java thread, we first have to create the Thread object and call it’s start() method to execute run() method as a thread.

**New**

When we create a new Thread object using *new* operator, thread state is New Thread. At this point, thread is not alive and it’s a state internal to Java programming.

**Runnable**

When we call start() function on Thread object, it’s state is changed to Runnable. The control is given to **Thread scheduler** to finish it’s execution. Whether to run this thread instantly or keep it in runnable thread pool before running, depends on the OS implementation of thread scheduler.

**Blocked/Waiting**

A thread can be waiting for other thread to finish using [thread join](http://www.journaldev.com/1024/java-thread-join-example) or it can be waiting for some resources to available. For example [producer consumer problem](http://www.journaldev.com/1034/java-blockingqueue-example) or [waiter notifier implementation](http://www.journaldev.com/1037/java-thread-wait-notify-and-notifyall-example) or IO resources, then it’s state is changed to Waiting. Once the thread wait state is over, it’s state is changed to Runnable and it’s moved back to runnable thread pool.

### Dead

Once the thread finished executing, it’s state is changed to Dead and it’s considered to be not alive.

**Multithreading in java** is a process of executing multiple threads simultaneously

**Thread is basically a lightweight sub-process, a smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.**

But we use multithreading than multiprocessing because **threads share a common memory area**. **They don't allocate separate memory area** so **saves memory**, and **context-switching between the threads** takes less time than process.

Advantages:

\* It **doesn't block the user** because threads are independent and you can perform multiple operations at same time

\* You **can perform many operations together so it saves time**

**\*** Threads are **independent** so it doesn't affect other threads if exception occur in a single thread.

**InterThreadCommunication:**

Java provide benefits of avoiding thread pooling using inter-thread communication.

The **wait()**, **notify()**, and **notifyAll()** methods of Object class are used for this purpose. These method are implemented as **final**methods in Object, so that all classes have them. All the three method can be called only from within a **synchronized** context.

* **wait()** tells calling thread to give up monitor and go to sleep until some other thread enters the same monitor and call notify.
* **notify()** wakes up a thread that called wait() on same object.
* **notifyAll()** wakes up all the thread that called wait() on same object.

**Wait and sleep:**

**\*Wait should be in synchronized method. No such requirement**

**\*monitoring of thread is released won’t release**

**\*Get’s awake when a notify or notifyall method calls won’t awake at notify() or notifyall()**

**\*Not a static method Static method**

**\*wait () is used generally in conditions. Simply to keep thread in sleep**

**ThreadPooling:**

Java **Thread pool** represents a group of worker **threads** that are waiting for the job and reuse many times.

**DeadLock:**

Deadlock is a situation of complete Lock, when no thread can complete its execution because lack of resources

**final:**

Final is used to apply restrictions on class, method and variable. Final class can't be inherited, final method can't be overridden and final variable value can't be changed.

**class** FinalExample{

**public** **static** **void** main(String[] args){

**final** **int** x=100;

x=200;//Compile Time Error

}}

**finally:** Finally is used to place important code, it will be executed whether exception is handled or not.

Class Test{

Public static void main (String args[]){

try{

int i=10;

}catch(Exception e){System.out.println(“Exception”+e.printStatckTrace())}

finally{system.out.println(“Manditory block to execute”}

}

}

**finalize:** Finalize is used to perform clean up processing just before object is garbage collected.

**class** FinalizeExample{

**public** **void** finalize(){System.out.println("finalize called");}

**public** **static** **void** main(String[] args){

FinalizeExample f1=**new** FinalizeExample();

FinalizeExample f2=**new** FinalizeExample();

f1=**null**;

f2=**null**;

System.gc();

}}