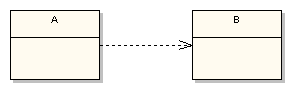
**Understanding UML Class Relationships**

* <https://vaughnvernon.co/?page_id=31>
* https://stackoverflow.com/questions/11881552/implementation-difference-between-aggregation-and-composition-in-java

## Dependency



In Java, the following is the proper interpretation of the constrained dependency relationship:

import B;

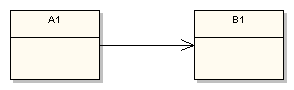
public class A { public void method1(B b) { // . . . }

OR

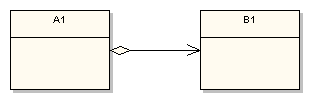
import B;

public void method2() { B tempB = new B() …. }

## Association



1. Aggregation



final class Car {

private Engine engine;

void setEngine(Engine engine) {

this.engine = engine;

}

void move() {

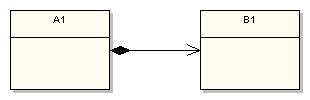
if (engine != null)

engine.work();

}

}

1. Composition



final class Car {

private final Engine engine;

Car(EngineSpecs specs) {

engine = new Engine(specs);

}

void move() {

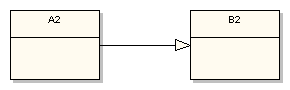
engine.work();

}

}

In the case of composition, the Engine is completely encapsulated by the Car. There is no way for the outside world to get a reference to the Engine. The Engine lives and dies with the car. With aggregation, the Car also performs its functions through an Engine, but the Engine is not always an internal part of the Car. Engines may be swapped, or even completely removed. Not only that, but the outside world can still have a reference to the Engine.

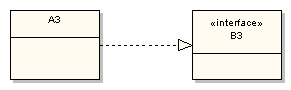
## Generalization



import B2;

public class A2 extends B2 { // . . . }

## Realization



import B3;

public class A3 implements B3 {// . . .}

**Why RSA**: to manage the functional, and non-functional requirements and to manage and monitor the traceability matrix between features and requirements in addition to that you can build the Use case diagram, ERD diagram, and class diagram.

**Dynamic programming**

<https://www.youtube.com/watch?v=jaNZ83Q3QGc>

http://bigocheatsheet.com/

In [computer science](https://en.wikipedia.org/wiki/Computer_science), [mathematics](https://en.wikipedia.org/wiki/Mathematics), [management science](https://en.wikipedia.org/wiki/Management_science), [economics](https://en.wikipedia.org/wiki/Economics) and [bioinformatics](https://en.wikipedia.org/wiki/Bioinformatics), **dynamic programming** (also known as **dynamic optimization**) is a method for solving a complex problem by breaking it down into a collection of simpler subproblems, solving each of those subproblems just once, and storing their solutions. The next time the same subproblem occurs, instead of recomputing its solution, one simply looks up the previously computed solution, thereby saving computation time at the expense of a (hopefully) modest expenditure in storage space. (Each of the subproblem solutions is indexed in some way, typically based on the values of its input parameters, so as to facilitate its lookup.) The technique of storing solutions to subproblems instead of recomputing them is called "[memoization](https://en.wikipedia.org/wiki/Memoization" \o "Memoization)".

Note:

1. A recursive function is tail recursive when recursive call is the last thing executed by the function

**public** **static** **int** **factorial**(**int** n) {

//return (n==1)?n:n \* factorial(n-1);

**if**(n==1) **return** 1;

**return** n \* ***factorial***(n-1);

}

1. A ternary search tree is a type of prefix tree where nodes are arranged as a binary search tree (BST). Like other prefix trees, a ternary search tree can be used as an associative map structure with the ability for **incremental string search**. So the most usage of Ternary search tree when you asked to implement an "auto-complete" user interface field.

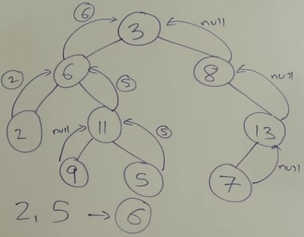
*Write a program which implements "auto-complete". Given a large list of names (forenames and surnames), create a UI that the user can start typing a name into, and for each letter typed, the names for which the text is a prefix appear below the text box.*

*Answer*

*It was specified that the UI here was not important. I wrote a very basic Java/swing application. I implemented a Trie to perform the prefix lookups. Since I had the weekend, I made this Trie implementation implement a prefix finder interface, and wrote simple HashSet and Ternary Tree implementations too and compared the performance.*

*Questions*

* Given 2 nodes in a tree, find the lowest common parent/ancestor (LCA) in a **Binary Tree**, must be efficient? See the following video <https://www.youtube.com/watch?v=13m9ZCB8gjw>

**

**class** Node {

Node left;

Node right;

Object value;

}

/\*parent is started by root node\*/

**public** **static** Node getLCA(Node parent, Node n1, Node n2) {

**if** (parent == **null**)

**return** **null**;

**if** (parent == n1 || parent == n2)

**return** parent;

Node left = *getLCA*(parent.left, n1, n2);

Node right = *getLCA*(parent.right, n1, n2);

**if** (right != **null** && left != **null**)

**return** parent; // the real ancestor such as “6”

**if** (right == **null** && left == **null**) // leaf node such as “9”

**return** **null**;

**return** left != **null** ? left : right;

}

* How to get LCA in **Binary Search Tree**

<https://www.youtube.com/watch?v=TIoCCStdiFo>

* How to get Largest BST in binary tree?

<https://www.youtube.com/watch?v=4fiDs7CCxkc>

Note:

A Tree is BST if following is true for every node x.

1. The largest value in left subtree (of x) is smaller than value of x.
2. The smallest value in right subtree (of x) is greater than value of x.

**Question**

Write a program which implements "auto-complete". Given a large list of names (forenames and surnames), create a UI that the user can start typing a name into, and for each letter typed, the names for which the text is a prefix appear below the text box.

Answer

It was specified that the UI here was not important. I wrote a very basic Java/swing application.

I implemented a Trie to perform the prefix lookups. Since I had the weekend, I made this Trie implementation implement a prefix finder interface, and wrote simple HashSet and Ternary Tree implementations too and compared the performance.

**Design Patterns**

* <https://www.programcreek.com/java-design-patterns-in-stories/>
* <https://stackoverflow.com/questions/1658192/what-is-the-difference-between-strategy-design-pattern-and-state-design-pattern>
* https://www.journaldev.com/1827/java-design-patterns-example-tutorial

**Creational**

* [Singleton - Only one president in America](https://www.programcreek.com/2011/07/java-design-pattern-singleton/)
* [Factory - A factory that produces human](https://www.programcreek.com/2013/02/java-design-patterns-factory/)
* [Abstract Factory - An abstract factory to produce CPUs](https://www.programcreek.com/2013/02/java-design-patterns-abstract-factory/)
* [Builder - Build a drink](https://www.programcreek.com/2013/02/java-design-pattern-builder/)
* [Prototype - Create a lot of similar objects](https://www.programcreek.com/2013/02/java-design-pattern-prototype/) – The Clone feature in Java :

**class** A **implements** Cloneable

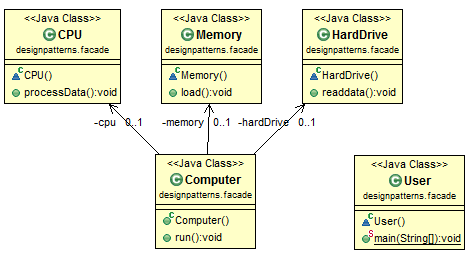
**Structural**

* [Adapter - Adapt an orange to an apple](https://www.programcreek.com/2011/09/java-design-pattern-adapter/)
* [Bridge - Bridge your remote control to a TV](https://www.programcreek.com/2011/10/java-design-pattern-bridge/)
* [Composite -Build a tree](https://www.programcreek.com/2013/02/java-design-patterns-composite/)
* [Decorator - Decorate your girlfriend](https://www.programcreek.com/2012/05/java-design-pattern-decorator-decorate-your-girlfriend/)
  + It’s typically more than one classes inherits a one class.

BufferedReader input = **new** BufferedReader(**new** InputStreamReader(System.in));

All inherit from java.io.Reader

* [Facade - Perform a complex task using simple interface](https://www.programcreek.com/2013/02/java-design-patterns-facade/)
  + It’s typically like a one class compose more than one classes to let you do a one function like run the computer and internally he use the all composed classes’ methods to do the complex work behind the seen.

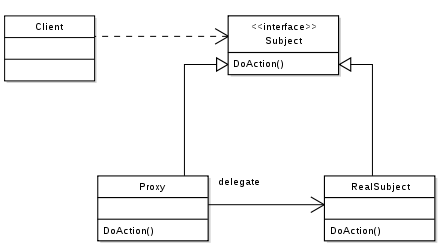


* [Flyweight - Create less and serve more](https://www.programcreek.com/2013/02/java-design-pattern-flyweight/)

It’s all about using any kind of data structures in java such as HashMap to put the heavyweight created objects in order to avoid recreating the same object with the same features twice.

Personally I used this pattern in the ECM to keep the initiated classes’ hierarchy that represents each workflow. This hierarchy has each workflow settings is loaded from the xml string that is saved in the database using JAXB.

* [Proxy - Use proxy to get do bad things](https://www.programcreek.com/2009/10/proxy-design-pattern-in-a-funny-story/)



In Client class -> Subject s = new Proxy();

In Proxy class -> DoAction() {RealSubject.DoAction();}

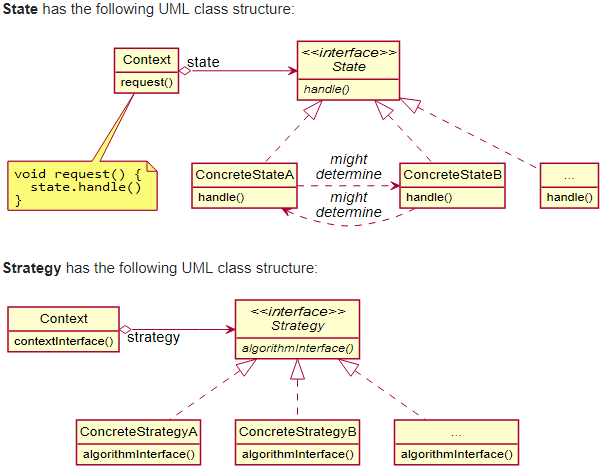
* [MVC - Module, View and Controller in Strut 2](https://www.programcreek.com/2011/08/struts-2-tutorials-mvc-design-pattern/)

**Behavioral**

* [Observer - Look for a job or observe a job?](https://www.programcreek.com/2011/01/an-java-example-of-observer-pattern/)

[Observer - A simple Swing GUI example](https://www.programcreek.com/2009/01/the-steps-involved-in-building-a-swing-gui-application/)

* [State - Work hard when life is hard](https://www.programcreek.com/2011/07/java-design-pattern-state/)
* [Strategy - Will you get a ticket if speeding](https://www.programcreek.com/2011/01/a-java-example-of-strategy-design-pattern/)

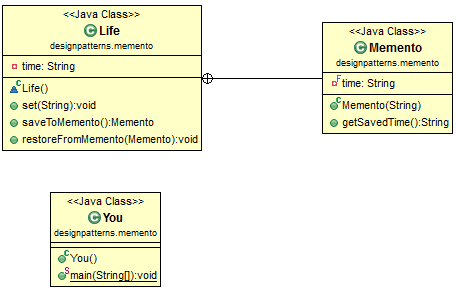


There can be a major difference, however, if ConcreteStates decide themselves the state transitions (see the "*might determine*" associations in the diagram above), so this is the state pattern in this case.

* [Template - Test a vehicle](https://www.programcreek.com/2012/08/java-design-pattern-template-method/)
* **abstract** **public** **class** Vehicle {
* *//set to protected so that subclass can access*
* **protected** **boolean** status;
* **abstract** **void** start();
* **abstract** **void** run();
* **abstract** **void** stop();
* **public** **void** testYourVehicle(){
* start();
* **if**(**this**.status){
* run();
* stop();
* }
* }
* }

This pattern is used in Spring framework's Data Access Object(DAO). org.springframework.jdbc.core.JdbcTemplate class has all the common repeated code blocks related with JDBC workflow, such as update, query, execute, etc.

* [Visitor - Visit New York City](https://www.programcreek.com/2011/05/visitor-design-pattern-example/)
* [Chain of responsibility - The responsibility chain](https://www.programcreek.com/2013/02/java-design-pattern-chain-of-responsibility/)
* [Command - Use different command to control computer](https://www.programcreek.com/2013/02/java-design-pattern-command/)
* [Interpreter - Interpret some context](https://www.programcreek.com/2013/02/java-design-pattern-interprete/)
* [Iterator - Iterate a collection of objects](https://www.programcreek.com/2013/02/java-design-pattern-iterator/)
* [Mediator - Mediate two colleagues](https://www.programcreek.com/2013/02/java-design-pattern-mediator/)
* [Memento - Use memento to time travel](https://www.programcreek.com/2013/02/java-design-pattern-memento/)



In future, time travel will be invented. Memento is the key to time travel. Basically, what it does is to allow an object to go back to a state.

Memento is an Inner Class in Life outer class

**Java Core Review by Question**

<https://www.javatpoint.com/corejava-interview-questions>

<http://www.geeksforgeeks.org/commonly-asked-java-programming-interview-questions-set-1/>

<http://www.geeksforgeeks.org/checked-vs-unchecked-exceptions-in-java/>

https://www.javatpoint.com/java-string

**Java Enum**

http://crunchify.com/why-and-for-what-should-i-use-enum-java-enum-examples/

* Why we cannot extends java Enums?

All enums implicitly extend java.lang.Enum. Since Java does not support multiple inheritance, an enum cannot extend anything else.

* What do you mean by Enum in Java are type-safe?

Enum has their own name-space. It means your enum will have a type for example “Company” in below example and **you cannot assign any value other than specified in Enum Constants**.

* You can specify values of enum constants at the creation time.

MyEnum.values() returns an array of MyEnum’s values.

public class CrunchifyEnumExample {

public enum Company {

EBAY(30), PAYPAL(10), GOOGLE(15), YAHOO(20), ATT(25);

private int value;

private Company(int value) {

this.value = value;

}

}

public static void main(String[] args) {

for (Company cName : Company.values()) {

System.out.println("Company Value: " + cName.value +

" - Company Name: " + cName);

}

}

}

So in this case the enum constants can send arguments to the enum constructor

# Regex in Java

https://docs.oracle.com/javase/tutorial/essential/regex/quant.html

# Collections in Java

# <https://www.javatpoint.com/collections-in-java>

# <http://careerdrill.com/blog/coding-interview/choosing-the-right-data-structure-to-solve-problems/>

# <http://www.columbia.edu/~jxz2101/#83>

# <http://www.geeksforgeeks.org/binary-tree-set-1-introduction/>

# http://www.javapractices.com/topic/TopicAction.do?Id=65

# Map Notes:

# <https://www.linkedin.com/pulse/10-things-java-developer-should-know-hashmap-chinmay-parekh>

# <https://stackoverflow.com/questions/11822552/chaining-in-hashmap> <LAST Comment>

Java Collection framework provides many interfaces (Set, List, Queue, Deque Map, etc.) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet etc).

hierarchy of collection framework

One of the Collection interface methods that is rarely used but still important is:

public boolean **retainAll**(Collection c)

This method is used to retain only the elements in this list that are contained in the specified collection. In other words, removes from this list all of its elements that are not contained in the specified collection.

There are **two** ways to traverse **collection** interface elements:

ArrayList<String> list=**new** ArrayList<String>();

1. By Iterator interface. -> Iterator itr=list.iterator(); … **while**(itr.hasNext()){…}
2. By for-each loop. -> **for**(String obj:al) {…}

There are **three** ways to traverse **list** interface elements:

1. By Iterator interface.
2. By for-each loop.
3. List interface. -> ListIterator<String> itr=al.listIterator(); … **while**(itr.hasNext()){…}

So what is the difference between Iterator interface and ListIterator interface?

* 1. **Iterator** is used for traversing List and Set both. We can use **ListIterator** to traverse List only, we cannot traverse Set using **ListIterator.**
  2. We can traverse in only forward direction using **Iterator**. Using **ListIterator**, we can traverse a List in both the directions (forward and backward).

# ArrayList class -> uses **dynamic array** to store the elements

The important points about Java ArrayList class are:

* Java ArrayList class can contain duplicate elements.
* Java ArrayList class maintains insertion order.
* Java ArrayList class is non-synchronized.
* Java ArrayList allows random access because array works at the index basis.
* In Java ArrayList class, manipulation (remove or insert in a certain position) is slow because a lot of shifting needs to be occurred if any element is removed from the array list.

# LinkedList class -> uses doubly linked list to store the elements (that means each node contains two fields, called links that are references to the previous and to the next node in the sequence of nodes.)

The important points about Java LinkedList are:

* Java LinkedList class can contain duplicate elements.
* Java LinkedList class maintains insertion order.
* Java LinkedList class is non-synchronized.
* In Java LinkedList class, manipulation is fast because no shifting needs to be occurred.
* Java LinkedList class can be used as list, stack or queue.

|  |  |
| --- | --- |
| ArrayList is **better for storing and accessing** data.  Random access for data | LinkedList is **better for manipulating** data.  Manipulation means (remove or insert in a certain position) |

# So in general

# LinkedList supports the random access for the data but slower than the ArrayList:

# LinkedList supports get() method but it’s not efficient.

Because if you use get(n) it skips past the first n-1 elements in the list. If index = n/2 then the search starts from the end of the list.

Random Access to a memory location means that getting location 1 is no quicker than location 8374587364587368569284365938475693875. As you have been told, getting location 2 in a linked list requires passing locations 0 and 1 first, so it is not random access.

# HashSet class -> is used to create a collection that uses a hash table for storage

The important points about Java HashSet class are:

* HashSet stores the elements by using a mechanism called **hashing.**
* HashSet contains unique elements only.
* It maintains no order.

# LinkedHashSet class -> it is a hash table and linked list implementation

The important points about Java LinkedHashSet class are:

* Contains unique elements only like HashSet.
* Provides all optional set operations, and allows null elements.
* Maintains insertion order.

# TreeSet class -> it implements the Set interface that uses a tree for storage

The important points about Java TreeSet class are:

* Contains unique elements only like HashSet.
* Access and retrieval times are quiet fast.
* Maintains ascending order.

Queue & Deque Interfaces

# Queue Interface: orders the element in FIFO (First In First Out).

# Deque Interface: linear collection that supports element insertion and removal at both ends. Deque is an acronym for **"**double ended queue**".**

## PriorityQueue class -> **it provides the facility of using queue. But it does not orders the elements in FIFO manner.**

Difference between PriortiyQueue and Linkedlist

PriortiyQueue as well as Linkedlist implement the Queue Interface and perform operations same the way Queue Operated (FIFO). The difference between PriorityQueue and Linkedlist is at the time of insertion PriorityQueue will be sorted as well as Ordered as per the natural Order but we can add a Commparator

## ArrayDeque class -> **it provides the facility of using deque and resizable-array.**

The important points about ArrayDeque class are:

* Unlike Queue, we can add or remove elements from both sides.
* Null elements are not allowed in the ArrayDeque.
* ArrayDeque is not thread safe, in the absence of external synchronization.
* ArrayDeque has no capacity restrictions.
* ArrayDeque is faster than LinkedList and Stack.

## **Map Interface & Map.Entry Interface**

# Map Contains values on the basis of key and value pair.

# Map contains only unique keys.

Map<Integer,String> map=**new** HashMap<Integer,String>();

.

.

.

**for**(Map.Entry m:map.entrySet()){

System.out.println(m.getKey()+" "+m.getValue());

}

OR

1. //Traversing Map
2. Set set=map.entrySet();//Converting to Set so that we can traverse
3. Iterator itr=set.iterator();
4. **while**(itr.hasNext()){
5. //Converting to Map.Entry so that we can get key and value separately
6. Map.Entry entry=(Map.Entry)itr.next();
7. System.out.println(entry.getKey()+" "+entry.getValue());
8. }

# HashMap class -> implements the map interface by using a hashtable

The important points about Java HashMap class are:

* A HashMap contains values based on the key.
* It contains only unique keys.
* It may have one null key and multiple null values.
* It maintains no order.

Note: A class that we plan to use as a “key” in hashMap needs to be immutable class.

* + What is chaining case in Map and when it takes place?

Further notes on when **Chaining** actually takes place when a HashMap is used:

The Java implementation for HashMap will either **override** a key or **chain** an object to it depending on the following:

1. You put an object *foo* as key, with hash code X into the map
2. You put another object *bar* (as key..) that has the same hash code X into the map
3. Since the hashes are the same, the algorithm would need to put the object *bar* on the same index where *foo* is already stored. It would then consult the *equals* method of *foo*, to determine whether it should **chain** *bar* to *foo* (i.e *foo.next()* will become *bar*) or **override** foo with bar:

3.1.If equals returns true, *foo* & *bar* are either the same object, or they are semantically the same, and overriding will take place rather than chaining.

3.2. If equals returns false, *foo* & *bar* are treated as two distinct entities and chaining will take place. If you then print your HashMap, you'll be seeing both *foo* and *bar*.

# LinkedHashMap class -> is Hash table and Linked list implementation of the Map interface, with predictable iteration order.

The important points about Java LinkedHashMap class are:

* A LinkedHashMap contains values based on the key.
* It contains only unique elements.
* It may have one null key and multiple null values.
* It is same as HashMap instead maintains insertion order.

# TreeMap class -> implements the Map interface by using a tree. It provides an efficient means of storing key/value pairs in sorted order.

The important points about Java TreeMap class are:

* A TreeMap contains values based on the key. It implements the NavigableMap interface and extends AbstractMap class.
* It contains only unique keys.
* It cannot have null key but can have multiple null values.
* It is same as HashMap instead maintains ascending order.

To iterate over the map you can do the following:

Map<K, V> map = **new** HashMap<>();

**for** (Map.Entry<K, V> entry : map.entrySet())

System.***out***.println(entry.getKey() + ", " + entry.getValue());

By Java 8 you can do the same by

map.forEach((User t, Integer u) -> System.***out***.println(t.name + ", " + u));

Playing around and nice features in Java Collections

List<String> list = **new** ArrayList<>();

list.add("Jessy");

list.add("Justin");

Set<String> set = **new** HashSet<>(list);

set.forEach(System.***out***::println);

## Java Comparator

The Player class is provided for you in your editor. It has fields: a String and an integer.

Given an array of Player objects, write a comparator that sorts them in order of decreasing score; if 2 or more players have the same score, sort those players alphabetically by name.

**class** Player {

String name;

**int** score;

**public** Player(String name, **int** score) {

**this**.name = name;

**this**.score = score;

}

@Override

**public** String toString() {

**return** "Player [name=" + name + ", score=" + score + "]";

}

}

**class** Checker **implements** Comparator<Player> {

@Override

**public** **int** compare(Player o1, Player o2) {

**int** score1 = o1.score;

**int** score2 = o2.score;

String name1 = o1.name;

String name2 = o2.name;

**if**(score2==score1) **return** name1.compareTo(name2);

// decreasing score

**return** **new** Integer(score2).compareTo(**new** Integer(score1));

}

}

.

.

Arrays.sort(players, **new** Checker());

OR using Comparable

**public** **class** Player **implements** Comparable<Player> {

String name;

**int** score;

**public** Player(String name, **int** score) {

**super**();

**this**.name = name;

**this**.score = score;

}

@Override

**public** String toString() {

**return** "Player [name=" + name + ", score=" + score + "]";

}

@Override

**public** **int** compareTo(Player o2) {

**int** score1 = **this**.score;

**int** score2 = o2.score;

String name1 = **this**.name;

String name2 = o2.name;

**if**(score2==score1) **return** name1.compareTo(name2);

// decreasing score

**return** **new** Integer(score2).compareTo(**new** Integer(score1));

}

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

Player[] players = **new** Player[]

{**new** Player("p1", 10), **new** Player("p2", 10), **new** Player("p3", 20)};

Arrays.*sort*(players);

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

}

}

Otherwise a ClassCastException will be thrown

My Conclusion on data collections

List => Insertion order

Map & Set => Unique

---------------------------------------------------------

Tree => Ascending order

Table => No Null is allowed

List {[ArrayList](https://www.geeksforgeeks.org/array-vs-arraylist-in-java/) (dynamic array based), LinkedList (linked list based)}

Set {HashSet, LinkedHashSet, TreeSet}

Map{HashMap, LinkedHashMap, Hashtable, TreeMap}

Vector -> Stack (Legacy)

Queue => PriorityQueue (Ascending order and no null allowed)

Deque => ArrayDeque (Add & remove from both sides)

LinkedList can act as Stack LIFO or queue FIFO

Checked Exception Handling with Method Overriding

**UNCHECKED EXCEPTIONS ARE ALLOWED IN ALL CASES.**

**1) Rule: If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception.**

**2) Rule: If the superclass method declares an exception, subclass overridden method can declare same subclass exception, more specific exception, or no exception.**

**Hibernate**

<https://www.youtube.com/watch?v=Yv2xctJxE-w&list=PL4AFF701184976B25&index=1>

* + The importance of mappedBy and cascade annotations

https://www.youtube.com/watch?v=fyryYbkiuok&list=PLAXbOMavY3k1VJGqqhfqAn0tUSC14Qsvq&index=47

Hibernate => JDBC, JTA, JNDI

* + Interceptors is used like AOP
  + Do you remember @Transient, @Temporal annotations
  + What if I add the attributes before getter instead of the fields?

Embed a Value Object => so it will be created as a separate class but it will be in the same table of the Entity that will embed it, but if we create a collection of Value Object then only in this case it will be created in a separate table.

@Embedable =equivilant to= @Entity

Then in entity @Embeded

@AttributeOverride,

@AttributeOverrides (

AttributeOverride, @AttributeOverride, @AttributeOverride})

For a primary key with multiple fields/(Compound key) use

@EbmededId

@ElementCollection => for handling a collection of Embeddable class, So it will create a new table by default.

@JoinTable => to override the default hibernate generated table @JoinTable(name=???, joinColumns=@JoinColumn(???))

>>>> Not JPA annotation

@Genericgenerator

@Genericgenerator(name="hilo-gen", type="hilo")

By default in case of collection of Value Object there is no Key for the new table that represents this collection, so to add a primary key for that table, you have to use a non JPA annotation called @CollectionId.

@CollectionId

@CollectionId(columns={@Column="ADDRESS\_ID"}, generator="hilo-gen", type=@Type(type="long"))

Notice ADDRESS\_ID not exist in the Embedded Class

Proxy Objects and Eager Fetch

Note: Hibernate always creating a dynamic proxies at run time for entities that have a collections to do the get list methods by fetching data first from database.

And this process done Lazily by default so that if you closed the session and try to get the list again it gives error.

To override that use @ElementCollection(fetch=fetchType.EAGER)

But you have to remember that this not the case in the first level properties/records like user.getUserName() this will still working even after closing the session because it is fetched eagerly by default.

Relation Mapping

@OneToOne, @OneToMany same as @ElementCollection

again if you add just @OneToMany the hibernate will create a new table

OR

@OneToMany(name="USER\_VEHICLE", joineColumns=@JoinColumn(name=USER\_ID)),inversJoineColumns=@JoinColumn(name=VEHICLE\_ID)) }

Then in Vehicle

Add @ManyToOne @JoinColumn(name="USER\_ID")

Thus no new table will be created for representing the one-to-many relation. Only a table for user and a table for vehicle as before.

Same case for Many-To-Many but it will create 2 tables if you gonna left as default without any settings by annotations.

So if you add in Vehilcle class @ManyToMany(mabedBy="vehiclList"), now it will create only one table to maintain the relation many-to-many.

In relation One-To-Many if you have a legacy data and some vehicles doesn't associated with a user, you can use @NotFound annotation which is not JPA standard.

To avoid to say session.save(vehilce1); session.save(vehilce2)...., you can add @OneTOMany(cascade=CascadeType.PERSIST).Now instead to say sassion.save(user) we gonna say session.persist(user).

Inheritance

Inheritance in hibernate strategies => Single Table strategy (Default behavior for hibernate), Table per class strategy, and Joined strategy.

* + Discriminator annotations to setup/discriminate the inheritance in hibernate, just in case of Single Table strategy

@DiscriminatorColumn

@DiscriminatorValue

@Inheritance => used to selecting the strategy.

In Table per class strategy, it will generate a table per class without any foreign key between parent and child.

Although in this strategy it still take care of auto generated values for the primary key. So if you have a field in parent class marked as @GeneratedValue, and try to insert a record in each table the parent and child so the value of this column in the first row in the parent table will be "1" and in the child table will be "2".

The Single Table strategy is the least normalize strategy => very bad design approach, but the Table per class strategy is slightly more normalize because the columns in the parent class are still repeated in the child classes.

PERSIST

Persist => by "session.save(user);" it will be tracked by hibernate so even if you change a property in the user object such as "user.setName("bla");" after the persisting the object the hibernate still tracking the object and save the updated data to database by running insert then update statements.

So hibernate tracked the persisted object by automatically triggering the one update statement to do the all changed in the properties, that done after the save object to session.

So hibernate is always intelligently do the least number of updates to reflecting the whole changes that were done on the properties.

So in general, persisting objects gives the authority to hibernate to reflecting the changes that made on the persisted object(s) into the database by watching those objects and keep tracking the values in objects to match the state of object with database.

Detaching => Once you close the session, hibernate doesn’t tracking the objects anymore and all persisted classes become in the detach state.

Persisting the detached objects by using => session.update(user)

### When and Why to Use Denormalization

<http://www.vertabelo.com/blog/technical-articles/denormalization-when-why-and-how>

It’s a step come after the normalization step in database design in order to

* **Maintaining history**
* **Improving query performance**
* **Speeding up reporting**
* **Computing commonly-needed values up front**

The idea behind it is to add redundant data where we think it will help us the most. We can use extra attributes in an existing table, add new tables, or even create instances of existing tables.

Database Tuning

<https://en.wikipedia.org/wiki/Database_tuning>

<https://www.toptal.com/sql/sql-database-tuning-for-developers>

https://dzone.com/articles/sql-query-optimization-and

Describes a group of activities used to optimize and homogenize the performance of a [database](https://en.wikipedia.org/wiki/Database). It usually overlaps with [query](https://en.wikipedia.org/wiki/Query_language) tuning, but refers to design of the database files, selection of the [database management system](https://en.wikipedia.org/wiki/Database_management_system) (DBMS) application, and configuration of the database's environment ([operating system](https://en.wikipedia.org/wiki/Operating_system), [CPU](https://en.wikipedia.org/wiki/CPU), etc.).

Database tuning aims to maximize use of system resources to perform work as efficiently and rapidly as possible. Most systems are designed to manage their use of system resources, but there is still much room to improve their efficiency by customizing their settings and configuration for the database and the DBMS.

Daemon thread

A **daemon thread** is a **thread** that does not prevent the JVM from exiting when the program finishes but the **thread** is still running. An example for a **daemon thread** is the garbage collection. You can use the setDaemon(boolean) method to change the**Thread daemon** properties before the **thread** starts.

Marker interface

Marker interface in Java interfaces with no field or methods. In **simple** words,**empty interface** in java is called marker interface. Example of marker interface is**Serializable**, **Cloneable** and **Remote interface**. These are used to indicate something to compiler or JVM.

**XSD** (**XML Schema Definition**) is a World Wide Web Consortium (W3C) recommendation that specifies how to formally describe the elements in an Extensible Markup Language (XML) document.

<https://stackoverflow.com/questions/2333998/what-is-the-difference-between-xml-and-xsd>

**JAXB >** <https://www.mkyong.com/java/jaxb-hello-world-example/>

**XStream**

Both are to convert Java object to / from XML file

**Spring**

<https://www.youtube.com/watch?v=GB8k2-Egfv0&list=PLGibysfsUS7NAbefiaj1V4LbX0glTftDI>

Spring is a container of beans or factory of beans.

Such as Tomcat is one of the servlet container that’s mean it creates the servlets that needs by the application based on the XML file.

So when we say Spring is the container of the objects it means: it handles the creation/instantiation of objects, handle the whole life cycle of objects/beans, and the destruction of those objects.

So Spring acts as or can be considered as Object factory and that happens using Spring Bean Factory that reads from Spring XML file to create the Spring Beans.

This way you isolating the instantiation of the objects from its usage, so that if the instantiation is a particular object is complex and need a lot of settings and work to do each time, you need an instance from that object, you can do these whole settings once in the XML and let the Spring to do that work each time you instantiate that class from Spring.

So the XML file is acting as a blueprint of the objects so the more complex the object pick up, the more detail the blueprint (XML) pickups and the more advantage you see in the method of creating objects through the spring.

Dependency injection as it's generally understood (and implemented by Spring) means that the dependencies that a class has such as a JDBC Datasource are not fetched by the class itself, but "injected" by a container when the instance is created at the startup time of the framework (Spring framework) itself.

Dependency Injection (DI)

The technology that Spring is most identified with is the Dependency Injection (DI) flavor of Inversion of Control.

The Inversion of Control (IoC) is a general concept, and it can be expressed in many different ways.

Dependency Injection is merely one concrete example of Inversion

of Control.

When writing a complex Java application, application classes should be as independent as possible of other Java classes to increase the possibility to reuse these classes and to test them independently of other classes while performing the unit testing.

Dependency Injection helps in gluing these classes together and at the same time keeping them independent.

What is dependency injection exactly? Let's look at these two words separately.

Here the dependency part translates into an association between two classes.

For example, class A is dependent of class B. Now, let's look at the second part,

Injection. All this means is, class B will get injected into class A by the IoC.

Dependency injection can happen in the way of passing parameters to the constructor or by post-construction using setter methods.

Aspect Oriented Programming (AOP)

One of the key components of Spring is the Aspect Oriented Programming (AOP) framework.

The functions that span multiple points of an application are called cross-cutting concerns and these cross-cutting concerns are conceptually separate from the application's business logic.

There are various common good examples of aspects including logging, declarative transactions, security, caching, etc.

The key unit of modularity in OOP is the class, whereas in AOP the unit of modularity is the aspect.

DI helps you decouple your application objects from each other, while AOP helps you decouple cross-cutting concerns from the objects that they affect.

* + 1. Spring lets your connect components, label them, control their lifecycles/caching, and alter behavior based on configuration.
    2. Testing an application written with Spring is simple because environment-dependent code is moved into this framework. Furthermore, by using JavaBeanstyle POJOs, it becomes easier to use dependency injection for injecting test data.

Scalable Apps

<https://www.romexsoft.com/blog/scalable-website/>

continuous-integration

https://www.thoughtworks.com/continuous-integration