## 1) Class

Class is basically a blueprint from which the objects are created, they contain state and behaviour.

### Object

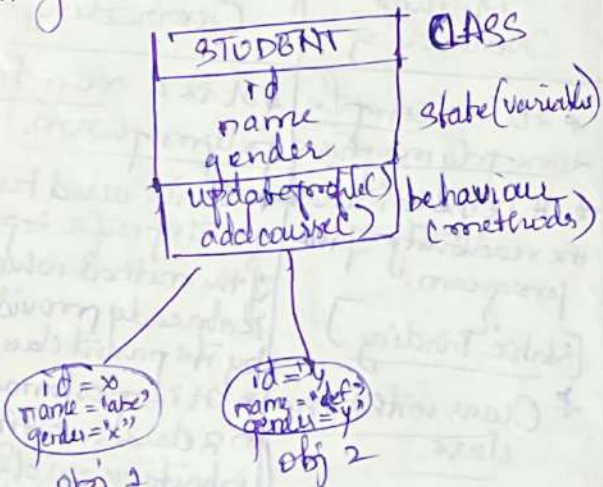
Object is a instance of a class that holds the state and behaviour of the class. The objects can be created by calling a constructor using "new" keyword

### Variables

Variables are basically the containers that holds the data of a particular datatype.
They are also called as state.

### Methods

Methods are set of codes that does a specific functionality on class calling it.
They are also called as behaviour




states are common with different data

but the methods/ behaviour is common to all obj.

### main Method

main()
→ The program starts from main() method.
→ must be declared as public static & void.
The program ends with main() method

## Explain Oops Concept

As we know, Oops stands for Object Oriented programming. As the name itself defines the value of object.

Here, we make use of objects in order to extract the behaviour of a class, for example, consider a building as an object and for that building to get build, we need a blueprint for the same, that blueprint is what we call as a class.

### Constructors [No return type]

Constructors are the object creaters. Usually when we do not create any constructors during the class design, the compiler itself will create one for us.
The main goal of constructors are is to initialize the state of an object, i.e., to initialize the instance variable of the class. [Std s = new Std();]

Backend → Create an object
→ invoke the const
→ store the memory address into ref variable

### Diff b/w default const & Parameterized Constructor

| Default | Param |
|---|---|
| * Zero argument constructors. | * will have atleast 1 argument |
| * Default constructs are the reasons there, when the default values are assigned to instance | * With this, we can assign specific values to the instance variable (user-specific) of different object |

| * Compiler adds the default constructor | * Programmer explicitly adds the constructors. |
|---|---|
| * Access modifiers are always public (or) default | * Access modifiers are always public (or) default. |

* Note :- Every default constructor will have zero arguments, but every zero argument constructor is not a default constructor.

Constructor Overloading.

A constructor having same name but different parameters.

Initialisers

Initialisers are basically the code blocks to initialise instance variables [In priority]

1) Static initialiser block
It is a class level initialiser that helps us define any static instance variables.

2) Member initialiser
This is a line of code, where we are forcefully declaring the value of an instance variable after initialising it using "=" operator

3) Initialiser block
It is a set of code to initialise the instance variable

4) Constructor

Before every constructor is called while building the obj, if a set of initialisat° to be performed 1-3) blocks can be used respectively Here in this stage, before the constructor is called, the above initialisers block will be executed based on the useage & requirement



Method Overloading vs Method Overriding.

Basically method overloading is nothing but having same method signature with different parameters.

Method overriding. ["@Override"]

In method overriding, we basically telling the compiler that I have a better implementation of the same method that is there in the superclass.

"If a subclass has the same method as declared in the parent class, it is known as method overriding in Java"

To be in detail, if a subclass (child class) provides the specific implement°/ a better implement° of a method that has been already declared by one of its parent class, it is known as method overriding.

| Method Overloading | Method Overriding |
|---|---|
| * It is a compile time polymorphism | * It is a run time polymorphism. |
| * It helps to increase the readability of the program. | * It is used to grant specific imp° of the method which is already provided by its parent class. |
| [Static Binding] | * It is performed in 2 classes with inheritance relationship [Dynamic binding] |
| * Occurs within the class | * Always need inheritance. |
| * May or may not require inheritance | * private & final methods cannot be overridden. |
| * private & final method can be overloaded | * Argument list should be same in method overriding. |
| * Argument list should be different while doing method overloading | |

## super() vs this()

| super() | this() |
|---|---|
| super() - refers to immediate parent class instance | this() refers to the current class instance. |
| * Can be used to invoke immediate parent class methods | * Can be use to invoke current class methods. |
| * super() acts as immediate parent class constructor and shd be in first line in child class constructor | * this() acts as a current class constructor and can be used in parameterised constr |
| * While invoking a superclass version of an overridden method, the super keyword is used | * while invoking a current version of an overridden method this keyword is used. |

## Abstract class vs Interface.

| Abstract class | Interface |
|---|---|
| * Abstract classes have abstract & non abstract methods | * Interface will have only abst methods & default methods & static methods |
| * Abstract class doesn't support multiple inheri -tance | * Supports multiple inheritance. |
| * It can have final, non-final, static, non static variables | * It can only have final & static variables. |
| * Abstract class can provide imple -mentation to interface. | * Interface cannot provide implemn to the abs class. |
| * Abs class can be extended using keyword "extends" | * Uses keyword "implements". |
| * Abs class methods are to be public & protected | * Members of interface are public by default |
| * Can extend abs class can | * Can extend only interfa |



## Exceptions

### throw vs throws

| throw | throws |
|---|---|
| * throw keyword is used to throw an exception explicitly in a code, inside a funct° or a block of code. | * throws keyword is used in the method signature to declare an exception might get thrown by the funct° during the time of execution. |
| * The type of except° throw Using throw keyword, we can only propagate unchecked except°, i.e, the checked except° cannot be propagated using throw only | * Using the throws keyword, we can declare both check & uncheck exceptions. However, the throws keyword can be used to propagate checked exceptions only |
| * The throw keyword is followed by an instance of exception to be thrown. | * The throws keyword is follow -ed by class names of exceptions to be thrown |
| * throw is used with in the method | * Used with the method signature |
| * We are allowed to throw only one exception at a time i.e., multiple except°s cannot be thrown | * We can declare multiple exceptions using throws key -word that can be thrown by the method |

### Rules of except° in overriding

* If the super class method does not declare an except°, then overriding methods in subclass cannot declare a checked except°, but can declare an unchek except°.
* If a superclass method declares a checked except°, then overriding method in the subclass can declare same checked except° or a subclass except° or no except° at all, but cannot declare parent exception.

* If the superclass method declares unchecked exception, then overriding method can declare any unchecked except or no except at all, but cannot declare a checked except.

## Final Keyword

It is basically an access modifier. The "final" keyword is applicable to classes, methods and variables.

### final variable

* When a final keyword is applied to a variable, it cannot be further modified
* A final variable must be initialized wherever where it is declared
* In coding standards, the final variables are declared in UPPERCASE
* They dynamically occupie memory at runtime, rather than on a per-instance basis.

Ex:- final int FILE_OPEN = 1;

### final method.

* When a method is declared as a final, then it cannot be overridden by its subclasses.
* Small methods that are declared as final can be made "inline" by the compiler which will reduce the overhead of function called and can increase performance enhancements.
* Usually, overridden methods are dynamically resolved, but when a method is declared as final, it cannot be overridden. Here, the function calling can be resolved at static time (compile time)

Ex:-
```
class A {
    final void do() {
        SOP("do");
    }
}
class B extends A {
    // cannot be done / cannot be overridden / inherited.
}
```



### final class

* When a class is declared as final, then it cannot be inherited by an subclasses
* Declaring a class as final will automatically declare all its methods as final.
* We cannot declare a class as both abstract and final.

Ex:-
```
final class A {
    // ...
}
class B extends A {
    // cannot be inherited.
}
```

## finally

* It is always associated with a try, catch block.
* finally block always executes after try-catch block.
* No matter, if the except' is thrown or not, finally block will always execute.
* finally block can also be used to clean up memory used by the resources in the try block.
* "finally" block is optional.

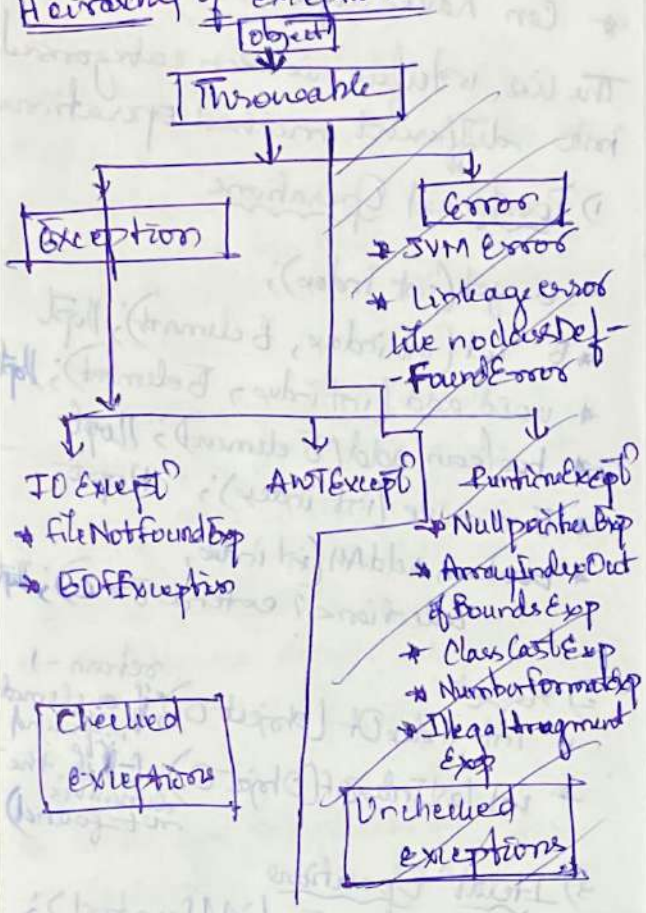
Ex:-
```
try {
    do something(); connectToDb();
} catch (Exception e)
{ e.printStackTrace();
} finally {
    closeDbConnection();
}
```

## finalize

* finalize is a method in a Object class.
* Usually, an object may be holding some non-java resources such as file-handle; then it must be freed before it is destroyed.
* This method is invoked by the GC before it destroys the object
* This method performs cleanup activities for Obj before it is being destroyed

* protected void finalize(){
  // code
}
* The finalize method is declared as protected so that it cannot be accessed from outside the class.
* This method is always called before the GC process is done.

Hairachy of Exceptions




Custom Exception

Main important conventions / items from effective java.

Item 58 : Use checked except° for recoverable conditions and runtime except° for programming errors

Item 65 : Don't ignore exceptions. [Do not have empty catch blocks]

When ever we decide create a new Exception, consider Item 58.

* According to that, we cannot extend "Errors", since there is a strong convention that only JVM produce errors, so we shouldn't be doing that (not a best practice)
* If it is something related to the programming error then, we shall go for "Runtime Exception".
* Usually, it is always better to make a checked exception rather than unchecked ("Runtime except°"). By this we can actually force the code to handle the exception by the compiler.
* Thus we have to extend only "Exception" or any of the sub-class of Exception because we cannot extend the "Throwable" class, as it is against the convention. [Just to avoid confusions].

Item 63 : Include failure-capture information in detail messages.

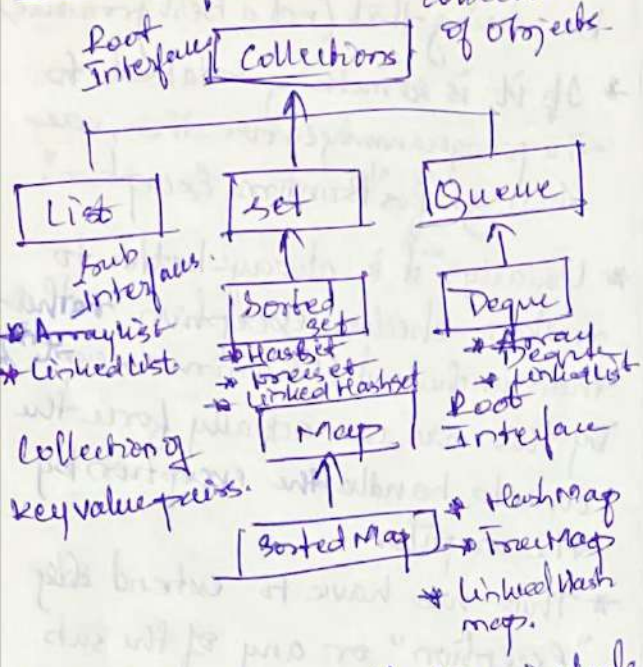
This item suggest that we as much as context possible to provide the info and encapsulate in the exception, to make it much more understandings clear for a debugger.

Collections (java.util)

why collections ~~[illegible]~~?

* No knowledge about size (not for us)
* Automatically extendible
* Fast random access
* Fast search / look up
* Ordered vs unordered
* Duplicates vs unique
* null vs non-null
* Automatic sorting
* <key, value> mappings.

## Core Interfaces




Collections on a collective include many fundamental methods and they are categorised as:-

* Basic Operation
* Bulk Operation
* Array Operation.

Basic Operat° methods <E> Generics

* boolean add (E element); // opt°
* boolean remove (Obj element); // opt°
* boolean contains (Obj element);
* int size();
* boolean isEmpty();
* Iterator <E> iterator();

Bulk Operation methods

* boolean addAll (Collect<? extends E> c); opt°s
* boolean removeAll (Collect<?> c); // opt°
* boolean retainAll (Collect<?> c); // opt
* boolean containsAll (Collect<?> c);
* void clear (); // opt

Array Operation methods

* Object [] toArray();
* <T> T[] toArray(T[] a);

Ex. - String[] a = c.toArray(new String[0]);



## List Interface

* It is useful when "Sequence/position-ing" matters. [preserving the order/retrieving the inserted [say storing the data accordingly one after the other from top to bottom]
* It models a resizeable linear array with indexed access
* Zero based.
* Can have duplicates.

The list interface is been categorised into different method operations

1) Positional Operations

* E get(int index);
* E set(int index, E element); // opt
* void add(int index, E element); // opt
* boolean add(E element); // opt
* E remove(int index); // opt
* boolean addAll(int index, Collection<? extends E> c); // opt

2) Search

* int indexOf(Object O); return -1 if no element is found
* int lastIndexOf(Object O); (-1 if the element is not found)

3) Iterat° Operations

* ListIterator<E> listIterator();

They basically return the instance of a Iterator / subclass i.e., ListIterat°

* ListIterator<E> listIterator(int index)

4) Range-view Operat°

* List<E> subList(int fromIndex, int toIndex);

## List Interface Implementation

### ArrayList

It is basically array implemented of a list interface i.e., Array list impl with resizeable feature.

* Basically the default capacity size is 10, if the capacity reaches limit, then a new array is created which is increased by 50% in size and are all copied to the new array.
* Say, if we know that the size (or) the capacity is high initially, then we shall actually set that with the help of a method; i.e.,

ArrayList(int initialcapacity)

* Say, if we wanna add large capacity data, then we can make sure to have a higher capacity of the Array size by a method

ensureCapacity(int);

If the size is not sufficient then it adds up the remaining capacity to ensure that the req size is available.

* It allowes storing duplicates and Nulls.

Typical Uses

* Used for simple Iteration of elements
* It got fast random access - O(1) since, we can make use of the index as an add on advantage.
* appending elements / deleting elements ~ O(1) ⇒ constant time. (To add elements, it would take some extra time when the array's maximum capacity is reached [if due to resizing].

add & remove methods.

* add(index, element)

when a new element is added, then all the other element shift right by 1 position

* Time complexity ⇒ O(n) ⇒ Linear



* remove(index)
  - following elements shifted left by one position.

Time complexity → O(n) ⇒ Linear Time.

Search - contains() & indexOf().

* Both the method will start searching for element from the begining of the arrayList hence the time complexity can be linear.
* Use equals() to compare element
* If frequent search is in need. then consider Set implementation (HashSet)

removeAll(Collection)

* collection.contains()
  Uses this method on every element if the element is present → remove it.
* Time complexity ⇒ O(n²) Can be worst than quadratic, i.e.,
  O(n) → for iterating all the element in the current list.
  O(n) → After iteration, we call contains() for each element if the input collect° obj is a list.
  O(n) → After removal, then shifting would cost time.

[Affect performance, if we are using large collections].

retainAll()

same as above. It has same performance issue.

Methods that run in constant time O(1)

* size()
* isEmpty()
* set() & get()
* iterator() & listIterator()

# Array Vs ArrayList

| Array | ArrayList |
|---|---|
| * It is basically a container that helps in holding the data of same datatype. | * ArrayList is a concrete implementation class of List which is a subinterface of Collection. |
| * Array is static in size | * ArrayList is dynamic in size |
| * Array is fixed length data-structure | * It is a variable-length datastructure |
| * While initializing, it is mandatory to provide the size of an array. | * We can create instance of ArrayList without specifying its size. JVM create the default size, defaultSize = 10 |
| * Can store both objects & primitive types | * Cannot store primitive types. It automatically converts primitive types to objects |
| * We can use for (or) for-each loop to iterate over an array | * We use an Iterator to iterate over ArrayList. |
| * We cannot use generics with Array because it is not a convertible type of Array | * ArrayList allows us to store only generic type, that's why it is type-safe |
| * Array can be multi-dimensional | * ArrayList is always single-dimensional. |

Similarities

* Both are used for storing elements
* Array & ArrayList, both can store null values
* They can have duplicates values
* They do not preserve the order of elements (?)



# Concurrent Modification Exception

Case-1

* Say we have created a list
  * list1 = [1, 2, 3, 4, 5, 6]
* through this list, We will create a sublist i.e.,
  List<Integer> list2 = list.sublist(2,5)
  so, list2 = [3, 4, 5]
* for sublist, we can add ext or do what ever we want but when do that for list1 in this case, it would throw an exception.
* i.e., After modifing the list1 say list1.add(0, 7)
  the list1 = [7, 2, 3, 4, 5, 6] (Error) X
  but after this, If we try to access the list2 (or) any of its method, it would throw concurrentModificat° Exception as there is a change in structure of list1.

[when we make an change in the backed array, then due to the structural change that has happened in the backed array the sub-array will throw an exception] [Sol$^n$ → Do not make any structural changes]

Case-2

when ever we try to use for-each loop for iterating and removing an element from a list, we come across the same exception, hence, we shouldn't be using for (or) for-each loops instead use Iterator() methods.

## Arrays Vs Collection

| Arrays | Collection |
|---|---|
| * Arrays are fixed in size i.e., it cannot be altered later. | * Collection and implementations has dynamic size allocation. It could be altered even after declaration |
| * Arrays due to fast execution, consumes more memory & has better performance | * Collection consumes less memory but also have low performance when compared to Array |
| * Can hold only same type of data i.e., only homogeneous data type elements are allowed | * Can hold both homogeneous & heterogeneous elements. |
| * Arrays can hold both objects & primitive type data. | * Collection holds only object type but while adding to the collection it converts primitives to objects |
| * Arrays due to its storage and internal implement, it has got better performance | * Less performance when compared to that of Array |

### Iterators

Iterator is basically an interface that is meant to iterate over the elements of a collection.

Iterator allows element removal during the iteration, which wasn't possible while doing it over a for-each loop. [ConcurrentModificationException].

To access the instance of a Iterator, 1st we need to invoke (or) extend the Iterable interface in those have to invoke Iterator() method which returns Iterator<T>



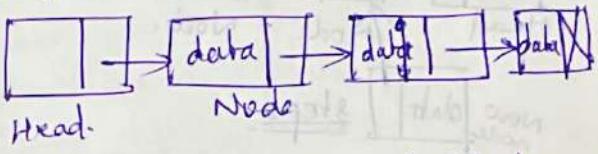
```
public interface Iterable<T> {
    Iterator<T> iterator();
    default void forEach(Consumer<? super T> action);
    default Spliterator<T> spliterator();
}
```
(partition & parallel operation)

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();
    default void forEachRemaining(Consumer<? super E> action);
}
```

### LinkedList [set of elements that are mutually linked]

Java is usually based on doubly linked list which has a variant called as singly linked list.

Singly linked list




Here are the set of elements that are linked, where each element is called as Node, and a node has 2 components where 1 is data which holds real data and 2nd is pointer to the next node.

```
class Node{
    int data;
    Node next;
}
```

The last node is has no pointer to the next node as there is no data hence, the last pointer (Node next) is set to null.

The 1st node is called Head node, (dummy node) that doesn't hold any data but points to 1st or last node based on the implement.

## Steps to link elements

Step1 = class LinkedList{

Node head = new Node();

void add (int data){

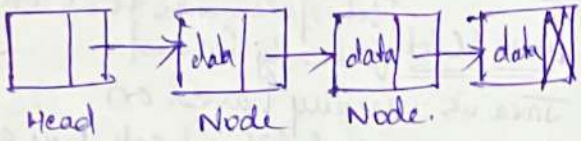
① Node newNode = new Node(data);

Here we have taking the int type data and we are creating a node accordingly in the line ①

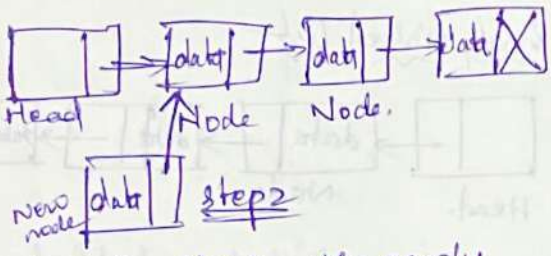
Step2 = void add (int data){

Node newNode = new Node(data);

② newNode.next = head.next()







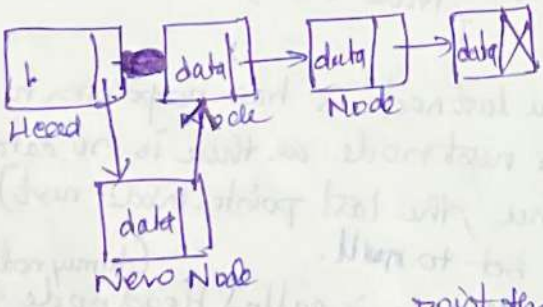

Now in step2, the newly added is also pointing to the same node, which head node is pointing to. [according to line ②]

Step3 = void add (int data){

Node newNode = new Node(data);

newNode.next = head.next();

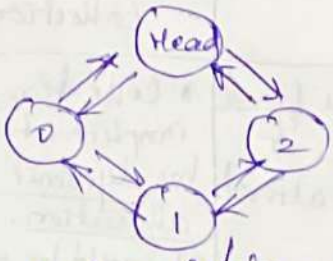
③ head.next = newNode;




Now, at last, we gonna point the head pointer to new data node and terminate the head pointer to previously added node automatically.



## Doubly linked list




Here, each node/element will have 2 pointers, one is towards its forward element & one is towards previous element.

Whenever a new element is added to the list, it will be right after the head. Here the added element is (2), which holds connectivity b/w (1) & (Head)

Basically, (Head) Holds the link or connectivity b/w 1st and last elements.

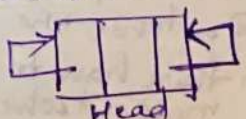
Advantage 1 - List can act as a Deque, which is subinterface of Queue

Deque basically have operations that frequently manipulate 1st and last element. (Head) will allow us to do that

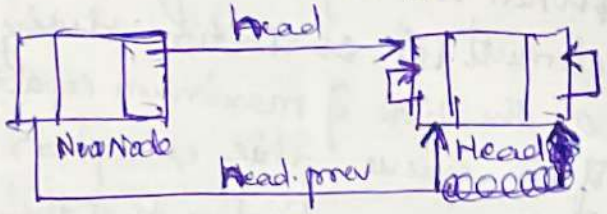
Advantage 2 - If we need to get or find any data based on the index, we just have to traverse through half of the list only. [traverse through the shortest distance].

### Steps to link elements doubly

Step1 = Initially we will have only head node, where the head node points next and previous to itself, because there are no other elements



Step2 = Create a new node and then set outgoing links.



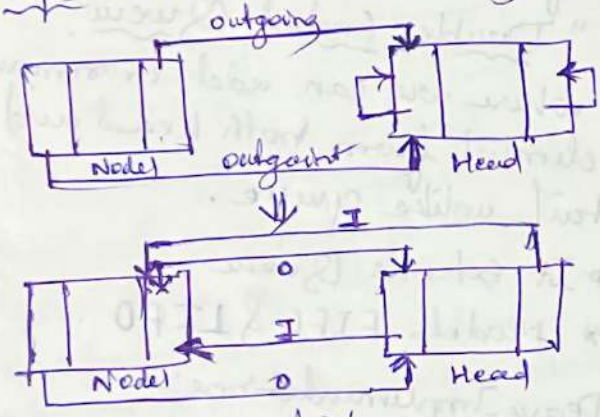

set outgoing links

① newNode.next = head

② NewNode.prev = head.prev

Since this node will be added right before head hence, the new node should be pointing to the head i.e., ①, and the previous pointer has to point to previously added node i.e., Head node.

Step3 = Set the incoming links.




Set incoming links.

① newNode.next.prev = newNode

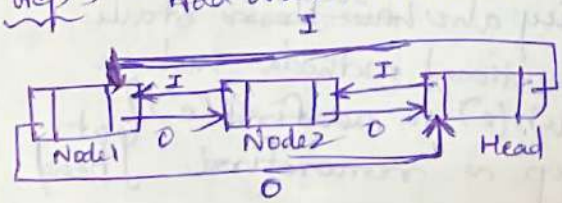
~~newNode.prev.prev~~

② newNode.prev.next = newNode.

Here newNode.next will be head and we are assigning its previous ref to the node.

newNode.prev will be previously added node and its next ref is been assigned to new node.

Step3 - Add one middle node.






Uses

* Frequent add/remove during iteration
  → Better for removeAll() & retainAll()

Some of the methods with O(n/2) time complexity

* get(i)
* add(i,e)
* remove(i)

implement to traverse over the list to have n/2 time complexity

```
if (index < (size >> 1)){
    for (int i=0; i<=index; i++)
        e = e.next;
} else{
    for (int i = size; i > index; i--)
        e = e.previous;
}.
```

Linked list models LIFO & FIFO operations in a constant time O(1).

Allows duplicates & nulls.

Arraylist vs linkedlist

| Arraylist | Linked list |
|---|---|
| * Arraylist initially uses a dynamic array to store the elements. | * It uses doubly linked list to store elements. |
| * Manipulat" is slow, since, it uses array internally and bit shifting slows down the performance. | * Manipulat" is faster since there is no bit shifting |
| * It can act only as a list because it implements list only. | * Acts as list as well as queue |
| * better for storing & accessing data | * better for manipulating data |
| * Memory for arraylist is assigned to elements are contiguous. | * the locat" for elemnts of a linked list is not contagious. |

* Generally, when an arraylist is initialis-ed, a default capacity of arraysize will be 10.

* Basically, an arraylist is a re-sizeable array

* There is no case of default capacity in a linkedlist
* An empty list is created when a linkedlist is initialised
* Linkedlist implements the doubly linked list of the list interface



## Queue Interface

* It is useful when manipulating head & tail. where the 1st element is termed to be head and last (or) recently added element can be the tail element.
* Any addition of elements → it happens at "tail" part
* Any removal (or) retaining process it happens at "head".
* It models FIFO
* It can have duplicates.
* Can have nulls, but generally not supported (It is not a good practise)
* No indexed access.

We have set of methods that inherits from collection interface and along with that, we get some special methods that returns special values.

| | Throws except° | Return special value |
|---|---|---|
| Insert | add(e) | offer(e) → false |
| Remove | remove() | poll() → null if queue is empty |
| Inspect | element() | peek() → null if queue is empty |

## Illegal StateExcept°



* When we try adding an element ~~into the at the time~~ during at the time of maximum capacity of the queue, this exception is thrown, indicating that the max size has been reached.

while on the other side,

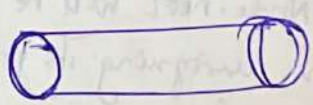
* offer(e) returns "false" for the same case [It is preferable to use offer method for capacity restricted queues].

## Deque

It is basically sub interface of Queue which stands for "Double Ended Queue" where we can add or remove element from both head and tail unlike queue.

* It extends Queue
* Models FIFO & LIFO

Deque Implementations:-
→ ArrayDeque
→ LinkedList
→ ConcurrentLinkedDeque
→ LinkedBlockingDeque.




add(e)/offer(e)
addlast/offerlast(e)
removelast/pollast
getlist/peeklast

remove/poll
element/peek
removefirst/pollfirst
getfirst/peekfirst
addfirst(e)/offerfirst(e)

They also support ~~have~~ stack operational methods such as
push(e) ~ addfirst(e) } at head
pop ~ removefirst

They also include few special methods

* removeFirstOccurance(Object) where the element is taken as an input and checks the queue, where ever the element has the 1st occurance, it removes that element.

* removeLastOccurance(Object). same as the above functionality but instead of 1st occurance of an element, it removes the last occurance of the input element.

We have something called Blocking Queues


* add() waits on full queue, to see if any of the elements vacancy can happen to add an element

* remove() waits on empty element i.e, if queue is empty it waits for an element to come in and then, that element will be removed.

This concept helps in Concurrence. (Inter thread communication)

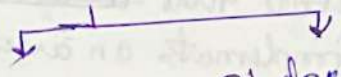
ArrayDeque

It is basically a resizeable array implementation of a Deque interface.

* It was added in Java 6.
* Models both FIFO & LIFO.
* It has got multiple intake of inputs such as ArrayDeque(), ArrayDeque(int) or ArrayDeque(Collection).



* Unlike LinkedList.



Nulls are prohibited

It does not implement List.

Adv of ArrayDeque Over LinkedList

* It is proved that ArrayDeque is 3x faster than the LinkedList for larger queues, it is because, in LinkedList, every time we add an element, we have to create a new node for it and would cost more memory. Similarly, while removing an element, we have to clear that node and this process is done by GC which would lead to more cost of GC which would also take more time and affect the performance.

* Unlike in ArrayList, we don't have elements shift, here instead the Array is called as a circular array which allows no elements shifts and takes advantage over ArrayList with constant time complexity O(1).

* There are some methods that also holds linear time complexity in true O(n).
  - → remove(Object)
  - → removeFirstOccurance
  - → removeLastOccurance
  - → contains.

Dis-adv

We cannot add/remove an element that is in between the queue, have to either remove all (or) make a new one.

## Hash Table

Hash Table is a fundamental datastructure that ~~is used for~~ implements an associative array which associates key with values. Basically, its a key value pairs.

Ex:- <Key, value> → <name, phone>

Each keyvalue pair is refered to as "mapping"

while hashtable is refered to "dictionary"

Opera Key Operations [O(1)] constant time complexity

* Insert <key, value>
  Insert a ~~key~~ value with a key - "map"
* Search by key,
  here basically, we can search out value by key
* Remove by key
  We can remove any value by using key of that specific key.

Characteristics

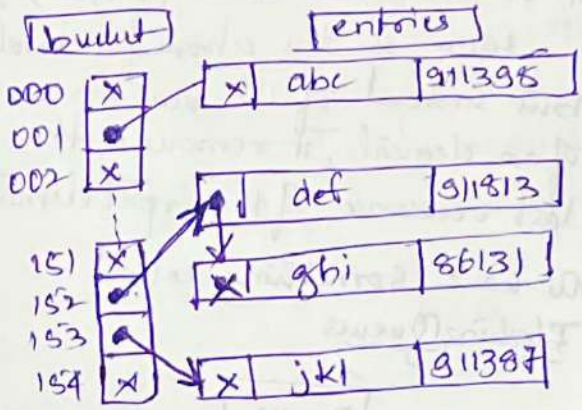
* No duplicate keys allowed
* Duplicate values are fine.
* 1 key can hold atmost 1 value.
* In few implementation, we can have 1 null key, and null values are fine into.
* In few other implementations No nulls at all for both keys & values.



## Implement

The HashTable is basically the functionality of array and linked list.

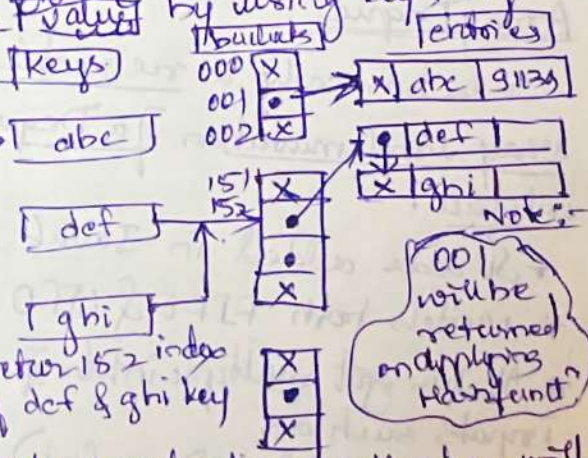
The array index, we call it as "buckets" and the linked lists are called as "entries"




in the implementation, we basically use array to store the linked-list coldness. So, basically, the key value pairs are stored in the linked list but not in array. We also can have multiple mapping as well, just as shown in the above image.

Whenever we wanna search or store elements, we have to 1st search or a proper bucket (or) the right array index and that operation is done by a funct called "Hash function"

Say for example., if we need a value by using key, say "abc"




Apply hashfunction to the key will return as 001 index.

Hash function (Dependent on Array size)

Hash function is a function of key and array-size.

bucket# = f(key, array-size)

Ex. key % array-size.

↓

% transforms large space to smaller one.

i.e., 315 % 25 = 15

target bucket

In java, it is basically 2 step hashing i.e,

2-step Hashing

hash = hash(key.hashCode())

bucket# = hash & (length - 1)

Here we use bitwise & instead of %.

key.hashCode() basically returns the Integer, which is basically the memory address of the object that is been converted. That int is taken as an input and hash(int) is invoked.

where the source code of hash() is

```
h^ = (h>>>20)^(h>>>12);
return h^(h>>>7)^(h>>>4)
```

Here the main goal of hashTable is to operate on constant time O(1) for that, the hashfunction must be very efficient and must have below mentioned properties.

* Quickly locate bucket

[Useage of bitshift and bitwise operators, the performance can be improved]



* hashfunction should be able to uniformly disperse the elements to the appropriate buckets

Hash collision.

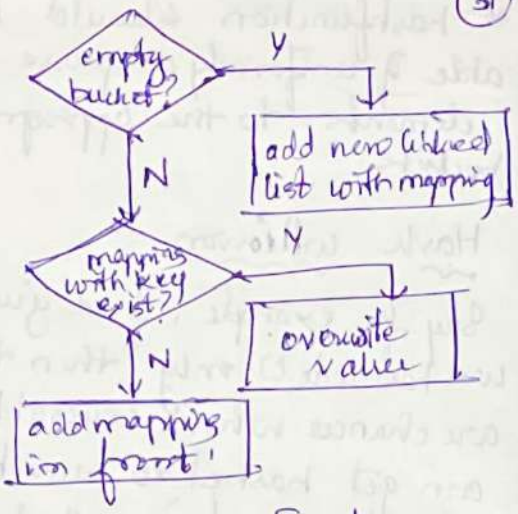
Say for example, if we just use hashcode() only, then there are chances where several keys can get hashed to same buckets and this concept is called as hash collision. If large number of mappings get stored into same bucket then the time complexity will differ. (or) gets slower. [Inside bucket, it would cause linear search to search for a linked list].

Insertion ~ Separate Chaining

1st, we will locate a bucket of a given key and will check if the bucket is empty or not. If the bucket is empty, we create a new LinkedList at that index and add the mapping into that linked list, but, if the bucket is not empty, i.e., there is a hash collision, then we will check if the LinkedList having another mapping with the same key. If no then add Mapping in front. If yes, then the value of key will be overridden by the new value for the same key.

This functionality is called as Separate Chaining.

The flowchart is been displayed in the next page.




## Other performance factors

There are other factors besides the good hash function() i.e.,

→ Capacity = # buckets.

Capacity is the number of buckets in the HashTable

→ LoadFactor = decides on resizing

Loadfactor is about how full the hashTable can get before its capacity is automatically increased.

For Hashset, default capacity is 16 and loadfactor is 0.75 i.e., the table will not be resized until it is 3 quarters full (3/4)

Resizing involves expensive rehasing i.e., because when the resizing of an array happens, then the whole new array is created and old array is copied to new, at that time the hash() has to be applied again newly as the capacity or array size is changed.

Hash() is dependent of Array size that which is rehashing.



## Applications

* Database Indexing.

In relation DB, indexes help in fast retrival of data and the indexes are based on HashTables.

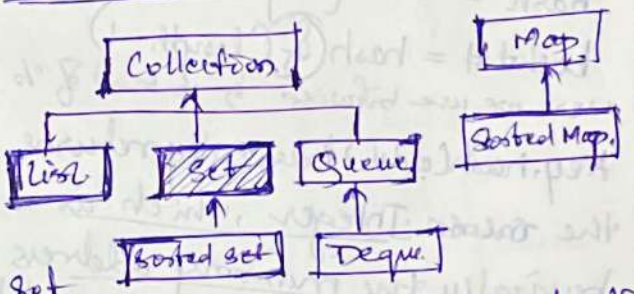
* NoSQL DB.

Even these DBs are based on HashTables as they are essentially key value stores.

* Switch Statement

Java, internally uses HashTable concept to quickly locate matching case blocks.

## Set & Hash set




## Set

Set interface extends collection interface. Hence Hashset is basically an implementation of Set interface.

* Set models mathematical set i.e., no duplicates.
* Useful when uniqueness & fast lookup matters. (HashTable example)
* Sorted set is a sub-interface of a set that provides sorting features.

## HashSet

It is basically a hashTable based implementation of a Set Interface.

## HashSet

* Internally uses HashMap
* key = element, value = new Object()

Since it uses hash map, the data is been stored in key value pairs. Moreover, since the hashset stores only individual Objects, these objects will be stored as keys while an empty object will be stored as value which is a instance of Object class.

* It also allows 1 null value.

### Use Cases.

* Used when rapid lookup, insertion and deletion is the main moto. - O(1) constant time.
* Insertion order is not important unlike list.
* Better for removeAll() & retainAll(). (Better than arraylist as the time complexity is $O(n^2)$.

### Equals() vs HashCode()

equals() and hashCode() methods are 2 important methods that are provided by Object class for comparing Objects.

Since Object class is a parent class for all java Objects, hence all objects inherit the default implementation of these 2 methods.

### Java equals()

* Java equals() is a method of lang.Object class, and is used to compare two Objects.
* To compare 2 objects that whether they are same, it compares the values of both the object's attributes.
* By default, two objects will be the same only if stored in the same memory allocation.

Eg - public boolean equals(Object o)

o: it takes the reference object as the parameter, with which we need to make the comparison.

=> It returns True if both the objects are the same, else returns false.

### General contract of equals() method.

equals() method must be:

* reflexive :- An object x must be equal to itself which means for object x, equals(x) should return true. x.equals(x);
* symmetric :- for 2 given objects x & y, x.equals(y) must return true if and only if y.equals(x) returns true.
* transitive :- for any objects x, y, z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.
* consistent :- for any objects x and y, the value x.equals(y) should change, only if the property in equals() changes.

### Java HashCode()

* A hashcode is an integer value associated with every object in Java, facilitating hashing in hash-tables.

* To get this hashcode value for an object, we can use the hashCode() method. Here through this method, i.e., hashCode() method returns the integer hashcode value of the given object.
* Since this method is defined in the Object class, hence it is inherited by user-defined classes also.
* The hashCode() method returns the same hash value when called on 2 objects, which are equal according to the equals() method stored at the same memory location. And if the objects are unequal, it usually returns different hash values.

## LinkedHashSet

* It is basically the hashtable of LinkedList implementation of Set interface.
* preserves insertion order which is not a feature of set before but due to doubly linked list it is made possible.
* It extends HashSet - nearly as fast too.
→ Used in rapid lookup, insertion and deletion ~ O(1);
* permits only 1 null element.
* Internally uses LinkedHashMap
→ Used when insertion order is important.
→ Better for removeAll() & retainAll().



Although many methods of LinkedHashSet might be slower but iterating of LinkedHashSet is faster than in HashSet.

Iteration(LinkedHashSet) faster than iteration(HashSet).

This because, the iteration of LinkedHashSet is dependent on the size of the set due to the use of doubly linked lists i.e., # elements.

Where as in HashSet, iteration is dependent on the capacity. i.e., no. of buckets in the HashTable i.e., Array size.

## SortedSet & NavigableSet

SortedSet which is basically the sub interface of Set interface. Where HashSet is known for its uniqueness and fast lookup while SortedSet extending set adds on sorting capabilities.

Along with that, we also have NavigableSet which is a sub-interface of SortedSet and it is implemented by "TreeSet"

### Methods of SortedSet

public interface SortedSet<E> extends Set<E> { // Range view

SortedSet<E> subSet(E fromElement, E toElement);

SortedSet<E> headSet(E toElement);

SortedSet<E> tailSet(E fromElement)

// End points.

E first();
E last();

// Comparator access.

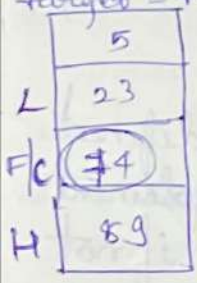
Comparator <? super E> comparator();
default Spliterator<E> spliterator();

public interface NavigableSet<E> extends SortedSet<E> {

// Closest matches.

E lower(E); // greatest element < E
E floor(E); // greatest element <= E
E ceiling(E); // least element >= E
E higher(E); // least element > E

target = 74




returns the value that is strictly lesser than the input element say here our input element is (74).

Now hence it returns "23" as the value.

// Iterators

Iterator<E> iterator() & descendingIterator();

iterator() method is the basic iteration of a ~~set~~ sorted set where as the descendingIterator() is the iteration in the reverse order

NavigableSet<E> descendingSet();
Identical to descendingIterator();

// Endpoints

E pollFirst() & pollLast();
polls/removes the first element of the set, similarly pollLast() removes the last element.



// Range view

NavigableSet<E> headSet(E toElement, boolean inclusive)

inclusive param is to just tell if they wanna include the input element or not. If yes then the param shd be true, if not then false

## Tree Set

Basically, tree set uses red-black tree - based implementation of NavigableSet interface.

→ It internally uses TreeMap
→ Key = element, value = new Object()
since set will have only elements, hence at the space of value, a new (ps) an empty object is filled, which is a reference of a Obj class.
→ Here the elements are unique & sorted.
→ ~~Fast~~ lookup too - O(log(n)) for add / removal / contains, although slightly slower than Hash set.

## Comparative vs Comparable

Usually in a TreeSet implementation we have to tell treeset that which type of sorting it has to make use of with having elements. For that our elements should have to be in some order.

Basically, there are 2 types of ordering that can happen,

① Natural Ordering -
Uses Comparable interface which has only 1 method i.e.,

compareTo(Object o)

It takes only 1 object and it has a return type as int

⇒ public int compareTo(T o)

So, here, we get to return integer value which can either be +ve, 0, -ve.

where say for example if there are set of elements say {10, 11, 13, 21}

Now, say we wanna insert

1) 10 ⇒ here the return type would be int & value will be "zero" as 10 already exists.

2) 12 ⇒ say I wanna insert 12, now 1st it will compare with 10 and returns "+ve value" as 12 is greater than 10.
now, it goes for 2nd element i.e., 11, compares with it and returns "+ve value" as 12 is greater than 11.

3) goes with 3rd element i.e., 13, checks with it. now 13 > 12 and hence returns "-ve value", now the compiler will know to put the value 12 b/w 11 & 13

Hence, the new set will be {10, 11, 12, 13, 21}.

3) Similar with "-ve" value as explained example.



Comparative

This is also an interface which has only 1 method i.e.,

⇒ public int compare(T o1, T o2)

Here we take 2 objects instead 1, we compare and return the int value, again here as well we have 3 types of values that can be returned i.e., +ve, 0, -ve.

This is the (2)nd type of ordering i.e., Comparitive interfaces methods.

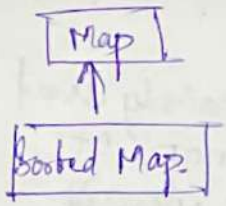
Note1:- for any user defined class, we have to implement Comparable interface if not it would throw ClassCast Exception.

for java defined class such as non-primitive types ie:) String, Integer etc, it is extended in prior.

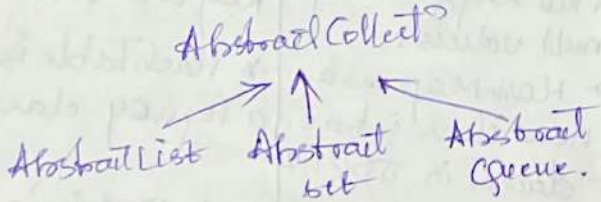
Note2:-

1) Comparable package
java.lang.comparable
2) Comparative package
java.util.Comparator.

## Map Interface




* As, SORT, the map interface is ~~one~~ one of the heirarchy of collection framework and they mainly use key-value pairs and values are found by keys.
* when fast look up by matters then, can go with Map Interface.
  Ex - data cache.
* Also called as associative array
* No duplicate keys. duplicate values are fine.
* 1 key → 1 value.
* null keys, max to max 1, based on implement°
* Can have null values again based on implementation.
* They also have Abstract Map which is basically a skeletal ~~representation~~ ~~of Map to Int~~ implement°. Basically, the implement° class like HashMap extend this particular class.




Declare°

```
public interface Map<K, V> {
```

Basic Operat°

Bulk Operation

Collection View Operations

```
}.
```



## Basic Operation

```
public interface Map<K, V> {
→ V put(K key, V value);
```

return type - returns the old value if there was a ~~key~~ old same key present, and the value will be overridden by a new value. The old value will be returned.

```
→ V get(Object key);
→ V remove(Object key);
→ boolean containsKey(Object key)
→ boolean containsValue(Object value)
→ int size(); // returns the size of the map.
  boolean isEmpty();
}.
```

## Bulk Operation

```
public interface Map<K, V> {
→ void putAll(Map<? extends K,
  ? extends V> m);
// basically adds all the elements of the input map along with keys to the existing map.
→ void clear();
// clears the map keys and values.
```

Collection view Operations — specially used for iterating view

```
public interface Map<K, V> {
→ Set<K> keySet()
```

// This method helps return all the keys of the map in the form of set. [set methods can be used]

```
→ Collection<V> values();
```

// It will return the collection view of all the values of a map. [Collect° methods can be used].

→ Set<Map.Entry<K,V>> entrySet();

(13)

// Mainly used for iterating the map. It returns the set view of all the mappings in the map, and each mapping is an instance of Entry interface which is a nested interface within the map interface

```
public interface Entry {
    K getKey();
    V getValue();
    V setValue(V value);
}
```

## HashMap

* O(1) for put, get & remove.
* Insertion order is not preserved as it internally uses hash() method.
* Permits null values and one null key.
* It is not synchronized unlike Hashtable, if it needs to be synchronized, it should be handled externally.

## Iterating Over Map

for iterating, we have set of methods mainly 2 methods.
i.e., 1) keySet().
Here by using keySet, we will get all the keys of the map and then, we make use of the same key to get the respective values by using for: each loop and then print them.

2) Using entrySet().

(14)

where Set<Map.Entry<K,V> entrySet();

This method is mainly used to iterating the map. It returns the set view of all mappings where each mapping is a instance of Entry interface [It is a nested interface within map interface] with this interface, we can iterate over for-each loop and get the respective key and value of the respective map.

## HashMap vs HashTable

| HashMap | HashTable |
|---|---|
| * HashMap is non-synchronised it is not thread safe and can't be shared b/w many threads without proper synchronized code. | * HashTable is synchronized. It is threadsafe & can be shared with many threads. |
| * It allows 1 null key & multiple null values. | * HashTable doesn't allow any null key & value. |
| * HashMap is a new class introduced in JDK 1.2 | * HashTable is a legacy class. |
| * HashMap is fast. | * HashTable is slow. |
| * We can make HashMap synchronized by calling this code. Map m = Collections.synchronizedMap(HashMap) | * It is internally synchronized and can't be unsynchronized. |

| | |
|---|---|
| * HashMap is traversed by Iterator | * Hashtable is traversed by Enumerator and Iterator. |
| * Iterator in HashMap is fail fast | * Enumerator in Hashtable is not fail-fast. |
| * HashMap inherits Abstract-Map class | * Hashtable inherits Dictionary class |

## Fail-fast and Fail-Safe Iterator.

Iterators in Java are part of the Java Collection framework. They are used to retrive elements one by one.

The Java collection supports 2 types of iterators i.e., fail fast and fail-safe. which is also called as non-fail-fast. These iterators are very useful in exception handling.

The fail-fast Iterators will abort the operation as soon as it exposes failure and stops the entire operation.

Comparatively, fail-safe iterator doesn't abort the operation in case of failure. Instead it tries to avoid failure as much as possible.

For understanding this concept it is important understand a concept called concurrent modification.



## Concurrent Modification.

Concurrent modification is a process to modify an object concurrently while another task is running over it. Basically, it is a changing the structure of the data collection either by removing, adding or updating the value of elements in collection. Usually, if anything goes wrong with the modification, the compiler will throw an exception called concurrent modification exception.

### Failfast Iterator.

Fail-fast system is a system that shuts down immediately after an error is reported. All the operations will be aborted instantly in it. Usually they through concurrent modification exception in case of structural modification of collection.

Eg - ArrayList, HashMap.

### How it works?

The failfast iterator uses an internal flag called modcount to know the status of the collection, whether the collection is structurally modified or not. So, the modcount flag is updated each time a collection is modified; it checks the value, if there is any modification done, if yes then it

would throw concurrentModifica-tion exception. (47)

## Fail-Safe Iterator:

The fail-safe iterator is a process that continues to operate even after an error or fail have occured. These systems do not abort the operations instantly; instead, they will try to hide the errors and will try to avoid failures as much as possible.

A fail-safe iterator does not throw any exceptions unless it can handle if the collection is modified during the iteration process.

This can be done because they operate on the copy of collection object instead of original object. The structural changes performed on the original collection are ignored by them and affect the copied collection, not the original collection. So, the original collection will be kept structurally unchanged.

| fail-fast Iterator | fail-safe-Iterator |
|---|---|
| * It throws concurrentModification except" in modifying the object during the iterat" process. | * It does not throw Except" unless it can't be handled. |
| * No clone object is created during the iterat" process | * A copy of clone object is created during the iteration process. |
| * It requires less memory during the process | * It requires more memory during the process. (48) |
| * It does not allow any modification during iterat" | * It allows modification during the iteration process. |
| * performance is fast | * slightly slower than fail-fast. |
| Ex - HashMap ArrayList HashSet | Ex - CopyOnWrite ArrayList ConcurrentHashMap |

## Internal implementation of HashMap.

There are mainly 2 methods that we have to know to understand the internal implement" of HashMap.

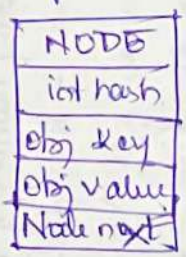
① map.put(key, object) and
② map.get(key, object).

So, basically, hashmap is a key, value pairs. datastructure Hashmap internally maintains a table that contains key value pairs.

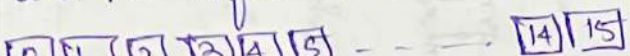
When we call put method, first it will check if there is any key of same type, if not found then will add the key and its value. But just in case if the key is found, the old value will be updated by a new value.

| Key1 | Value 1 |
|---|---|
| Key2 | Value 2 |
| Key 3 | Value 3 |
| ⋮ | ⋮ |

Hashmap store the key pair value in the "Node" object from. Basically Node object contains hashcode, obj key, obj val and link of next node.




WKT, hashmap internally uses hashtable to store the nodes and the default table size is 16.




Say for example, we have to put a data i.e., (key) (value)

① map.put("alice", 98123456");

Here the hash value is will calculated by using hashCode() method, where the hash code will return store int value

② "alice".hashCode() ⇒ 92903

Similarly, post that index will be calculated using that hashCode

i.e. ③ index = 92903 & (n-1)

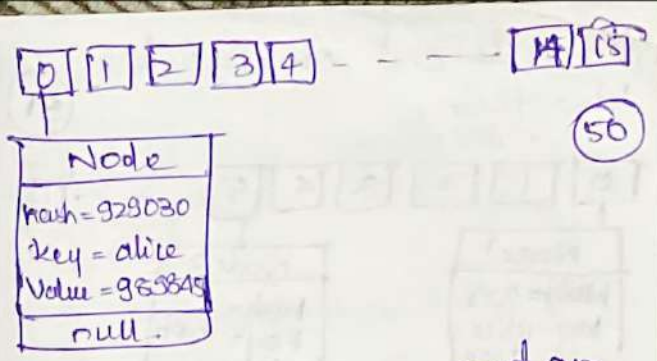
where n is the size of hashtable and the index number will be calculated using logical AND operation.

Now, say our index value is zero.

④ index = 0.

Hence the Node obj will be stored at index 0.

And since there is no next node link hence it will be set to null






Say, now we wanna put another data to hashMap.

⑤ map.put("bob", 9858548);

Now, it is same as usual,

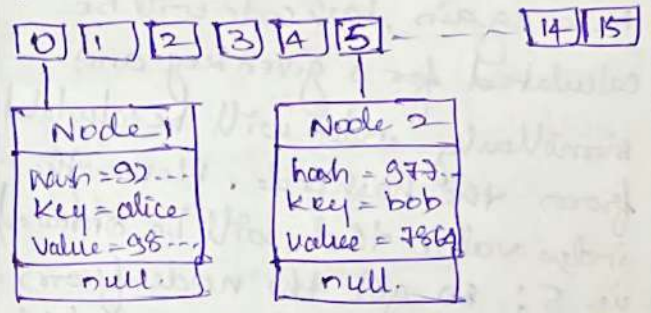
→ hashcode will be calculated for a given key

→ with that key, we gonna calculate the index as well

→ and then assign it to that index

Say the index value will be 5.

Now,




Now, we shall go adel 1 more key value pair,

⑥ map.put("claj", 675649);

Again same process,

→ calculate hashcode, index and then assign it to the resulted index.

Say, Now, we have got index value as "0".

Usually, different object will get same index number after calculat°. This process is called hash-collision.

Now, since we got index as 0 it will check if the same key exist. If not it will stack it after the

## List

* List interface allows duplicate elements.
* The list maintains insertion order.
* We can add any number of null values.
* List impl classes - ArrayList, LinkedList
* The list provide get() method to get the element of a specified index
* If you need to access the elements frequently by using index, then we can use list
* To traverse the list elements we use ListIterator.

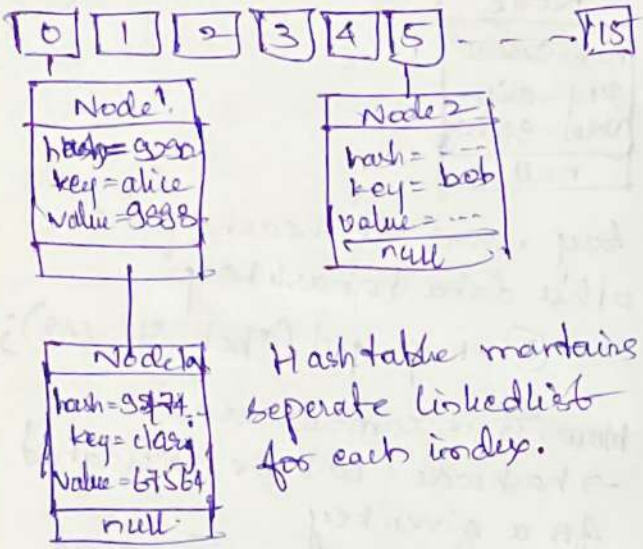
## Set

* Set does not allow duplicate elements
* Do not maintain any insertion order.
* Almost only one null value
* Set impl classes - HashSet, TreeSet.
* Set does not provide get method to get the elements of a specified index
* If you need to create a collection of unique elements then we can use set
* Iterator can be used to traverse the set elements.

## Map

* Map does not allow duplicate elements. (Keys)
* Does not maintain any insertion order.
* Almost single null key and many number of null values.
* Map impl classes - HashMap, TreeMap, Concurrent HashTable, LinkedHashMap
* Map does not provide get method to get the elements.
* If you want to store data in form of key, value pairs then we use map.
* Traversing is done through keyset(), value and entryset.

Old node and create a link b/w them. (51)




Hashtable maintains seperate linkedlist for each index.

Now, very we have to get the value by the key and the process table here.

(9) map.get("bob");

Now again, hashcode will be calculated for a given key and simultaneously index will be calculated from the hashcode. Hence the index value that will be obtained is 5; so get the node from index 5 and compare the hashcode using equals() method. If both are same, then, it checks the node key with the given key. If both are same then it just returns node's value.

Here we get out value.

i.e., value = 7655645;

Differences b/w List Set & Map

## LinkedHashMap

The LinkedHashMap is basically the combination of hashTable and LinkedList of a Map Interface.

* It preserves insertion order where as this wasn't possible in HashMap alone.
* The linked list mechanism used overhere is "Doubly linked list".
* Extends HashMap - nearly as fast too!
* Rapid lookup, insertion, deletion are having constant time complexity ≈ O(1) just like HashMap.
* It permits null values & null key
* Not synchronized.

Iterat"

Iteration(LinkedHashMap) is faster than Iteration(HashMap).

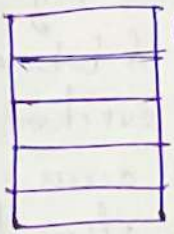
Reason :- In LinkedHashMap, since we use linked list, the iterat" speed is dependent on the size of the doubly linked list. Hence holds upper hand in iterating the elements, where as the HashMap is dependent on the capacity of the map which is a drag in the performance slightly when compared to LinkedHashMap.

Iteration(LinkedHashMap) → #mappings
Iteration(HashMap) → capacity.

LinkedHashMap has a special feature ~~called~~ i.e., it can be used as LRU cache i.e., Least Recently Used Cache



## LRU Cache



least recently used (full-cache) → remove (evict)

most recently used. ≈ insert / most recently accessed.

1) Here in this mechanism, say, if we add one entry to the cache, it will be considered as most recent used item and will be the highest position.
2) Say, if we are in stage of getting an entry if an existed item, then that item would be moved to highest position and the old item will be shifted back by 1 position.
3) Similarly, when the element/item is at the last position(or) if it is least used then that item will be evicted.
4) If the item in need (or) in search in the cache is found, it is called as "cache hit", If by if the item is not found, then it is called as "cache miss", at this stage, compiler will search the data in DB, if found then puts it to cache, so that it can be used multiple times without any lag.

Constructor that helps in enabling or creating use of LRU cache (or) LinkedHashMap as LRU cache.

* LinkedHashMap(int initialCapacity, float loadfactor, boolean accessOrder)

boolean accessOrder = "true"

Usually, when we use this constructor for LRU cache by enabling it, although it behaves as LRU cache, but the eviction process will not gonna occur when we insert a new element as the size of LRU cache is unlimited.

But, just in case if we need to enable the evition process when we have to create a new class and that new class has to be extended by LinkedHashMap, in there, we have to set the size limit, post which eviction can occur by using this method i.e., overriding it.

removeEldestEntry (Map.Entry<K,V> eldest)

Usually, this method will be invoked when we use put(or) putAll method.

→ This method while eviction checks if the size of cache is full or not, if it is full then it returns true and the eviction of eldest value will be done, while if it is not full then, it would return false. and no element will be removed

## SortedMap & NavigableMap

### SortedMap

- Map capabilities + data sorted by key, the sorting of key is either done by natural ordering or by comparator (refer HashMap TreeSet)



+ Natural Ordering or Comparator ~ by keys.

```
public interface SortedMap<K,V>
  extends Map<K,V> {
  //Range-view
SortedMap<K,V> subMap(K fromKey,
                      K toKey);
SortedMap<K,V> headMap(K toKey);
SortedMap<K,V> tailMap(K fromKey);
  // Endpoints
K firstKey();
K lastKey();
  // comparator access
Comparator<? super K> comparator();
  // Collection view Operations
Set<K> keySet();
Collection<V> values();
Set<Map.Entry<K,V>> entrySet();
}
```

---

```
NavigableMap<K,V> extends
  SortedMap<K,V>{
  //closest Matches
K lowerKey/floorKey/ceilingKey/
higherKey(K key);
Map.Entry<K,V> lowerEntry/
floorEntry/ceilingEntry/higherEntry
(K key);
NavigableSet<K> descendingKeySet();
NavigableMap<K,V> descendingMap();
  // Endpoints
Map.Entry<K,V> pollFirstEntry();
Map.Entry<K,V> pollLastEntry();
```

// range view

NavigableMap<K,V> headMap
(K toKey, boolean inclusive);

};

## STRING methods

length() ⇒ returns int

isEmpty() ⇒ returns boolean

equals() ⇒ returns boolean

equalsIgnoreCase() ⇒ returns bool
Ignore Case sensitivity.

compareTo() ⇒ returns int
⇒ It compares 2 strings lexicographically.

* The comparision is based on the unicode value of each character in the Strings.
* The character sequence is lexicographically compared i.e., String Object is compared to character seq. represented by the argument string.
* result is +ve, if the string obj lexicographically follows the argument string.
* result is -ve, if the String Object lexicographically precede the argument string.
* result is 0, if the strings are equal. [It would return 0 when equals() method returns true]

contains() ⇒ returns boolean

startsWith() ⇒ returns boolean

endsWith() ⇒ returns boolean

indexOf() ⇒ returns int (index value)
⇒ Index number

lastIndexOf() ⇒ returns int (else -1)



substring(from index)
⇒ returns string from index to last

substring(from index, to index)
⇒ returns from index to toindex - 1

toUpperCase() ⇒ converts the string to upper case.

toLowerCase()

a copy is returned because string is immutable

trim() ⇒
// returns a copy of a string after trimming any leading & trailing white spaces

replaceAll("x", "y")
from by
// returns a updated copy.

split()
⇒ returns the split string as a form of arrays.

Ex = s = "hello World!"

```
String[] sa = s.split("o");
for(String temp : sa){
    SOP(temp);
}
```

o/p = hell
w
rld!

[split a doc into words (or) split a line of text by tab or comma or white space]

valueOf() ⇒ static method
⇒ String.valueOf()
⇒ Returns the string rep of the double argument.

Ex ⇒ String.valueOf(1.3);
o/p = "1.3" as a string.

## Other String Utility Classes

⇒ Apache Commons Lang ~ StringUtils
⇒ Guava's String Utility Classes.

## String Immutability

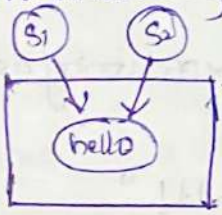
* Why strings are immutable.

⇒ Once a string is created its values can never be changed.

```
String str1 = new String("abcd");
str1 = new String("1234");
// above obj is abandoned.
```

① String Interning.

If strings were mutable, then it wouldn't be possible to share the memory [string pool].




say s1 and s2 are pointing to same memory and say s1 has modified literal, now s2 will be expecting to have its content as hello, but it would no more be same. This would cause serious trouble.

② Concurrency


Usually when 1 thread accss the content and modifies, and the thread 2 that is been still referenced to the same would actually get affected. This would cause more bugs.

③ Security

There are many networking related classes in java that take string as i/p Using these classes with mutable strings can cause lot of security vulnerabilities. Ex, to read a file in java there is class called "FileInputStream(Str name = file path for authe. | name)



## String Pool.

Here before string pool, 1st we'll see how the strings objects are stored internally by JVM: Usually String creat occurs in 2 types:-

1) With String literals;
2) By using new keyword

### String literals

* Usually, the string literals are stored in a "string pool" which is apart of heap memory where the objects reside.
* In the "string pool", the objects with the same content share common storage (shared)

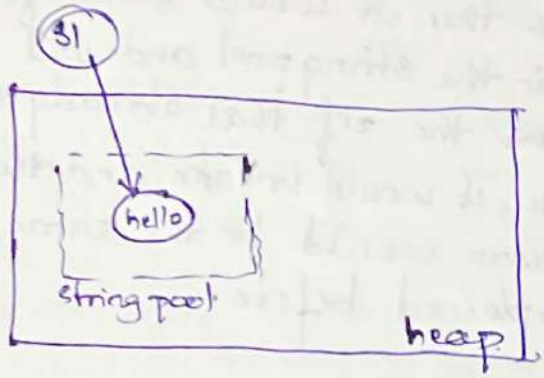
### new

* It is same as that of other objects which just resides at heap memory but not in "string pool".
* Here, even if the string objects having same content, the storage is not shared/common.

### Mechanism

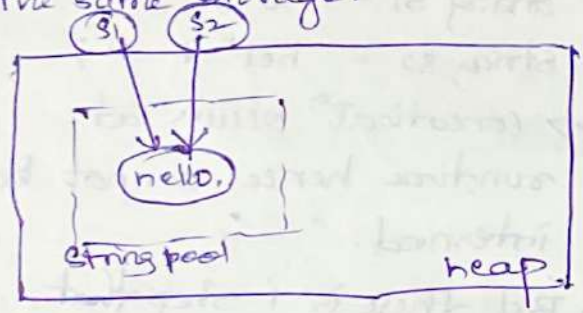
1) String s1 = "hello";

Now the jvm will check for this string literal if it exist in the string pool or not, as of now, there is no literal with this word in the the stringpool, hence, an obj is created and is stored in the string pool and that storage refere is pointed to s1 as shown in the image



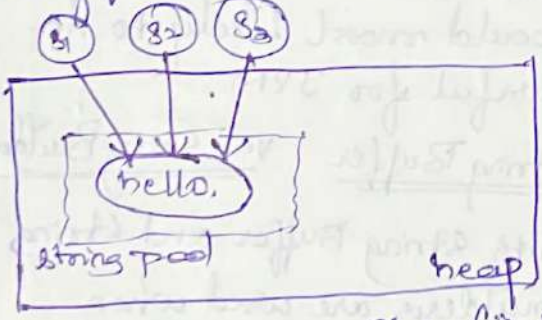

2) String s2 = "hello";

Now, since jvm knows that there is already a literal with the same content, hence that s2 will also be refferenced to the same storage.




3) String s3 = s1;

Now, say s3 = s1; again it means that the content is same and hence, again the s3 will be referenced to the same obj in the string pool.



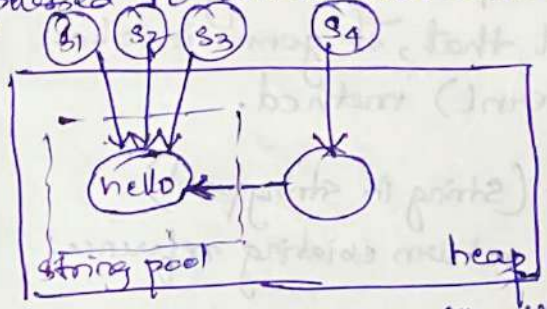

4) String s4 = new String("hello");

Now, since the new keyword is used, the object is created as usual but this time it is outside the string pool, but jvm somehow tries to refer to string pool.

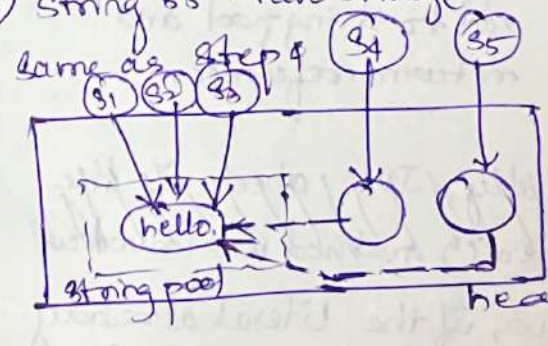
Now, JVM checks if there is a content / literal of the same kind, and wht it exist. (62)

Hence, now s4 will be referencing to the newly created object where that obj will be refern - ing to the string pool storage and that refference address is been passed to the constructor.

Say, if there was no same content, even at that sitcuat' a new obj ~~was~~ will be created in the string pool and also outside the string pool and then the ref of string pool obj will be passed to constructor as usual.




5) String s5 = new String("hello");




Now, in total. On obj comparison

$s_1 == s_2$ ? ⇒ true

where as $s_4 == s_5$ ⇒ false

Hence, the main advantage of using string literal is, it would save us memory.

* String pool stores single copy of each string literal as string object
* Only one string pool.
* It is also called as string table.

" The whole process of building up string pool is called as string Interning" and " Each literal in the string pool is called is intern".

String interning by JVM.

So, basically, whenever JVM encounters a string literal for the 1st time, it would first create an object with the given literal. Post that, it gonna invoke intern() method.

```
if (String in stringpool)
    return existing reference;
else
    add to stringpool and
    return reference.
```

Usually, JVM check if the intern() method is invoked

Now, if the literal already exist, then the created obj is abandoned & garbage collected. else, a new obj is created in the string pool and the reference is passed to the constructor.

Usually, JVM whenever, it en-counters a literal, it will 1st check if the intern() method is



invoked for that literal, if yes then it would search for it in the string pool and will pass the ref that already exist else, it would invoke and the process would be the same as explained before.

Examples

* String s1 = "hel" + "lo";
  ⇒ since concatinat" occurs at compile time, it is interned.
* String s1 = "lo";
  String s2 = "hel" + s1;
  ⇒ concatinat" occurs at runtime hence cannot be interned.
* But there is 1 step that can be done. i.e.,
  s2 = s2.intern();
  This is called explicit interning but most likely it is not that useful. It is better to be meant with the JVM. And this process would most likely to be useful for JVM.

String Buffer vs String Builder

Both String Buffer and String builders are used when mutability is required. With this we can actually, append, delete, insert, reverse and replace the string literals or objects, where while after each append() is invoked, the ref is returned to the variable.

| StringBuffer | StringBuilder |
|---|---|
| * Every method present in StrBuf is synchronised | * No methods are synchronised |
| * At a time only one thread is allowed to operate hence it is Thread safe. | * At a time multiple thread are allowed to operate hence non thread safe |
| * It increases waiting time of the threads hence relatively performance is low | * It has got no waiting periods as all threads can operate at 1 shot resulting good performance relatively |
| * Introduced in V1.0 | * Introduced in java V1.5 |

## Functional Programming (1)

* They basically support creating Immutable objects.
* More concise & readable codes
* Using functions/methods as first-class citizens. - passing as a method as param.

Ex:-

```
Function<String, String>
addSomeString =
(name) -> name.toUpperCase().
          concat("default");
```

can be passed as a param to the method arguments.

* Writing code using Declarative way rather than Imperative approach.

### Imperative vs Declarative Programming

| Imperative | Declarative |
|---|---|
| * focus on how to perform operations. | * Focuses on what is the result you want rather than how you are doing it. |
| * Embraces obj mutability i.e., changing the obj state to achieve an objective. | * Embraces Obj Immutability |
| * This typed programming lists the step-by-step of instructions on how to achieve an objective | * Analogus to SQL - where we don't know how we retrive data from table at back. we just write query |
| * We write code on what needs to be done. | * Usese the funct* that are already part of lib to achive objective. (2) |
| * Imperative style is used with classic Object Oriented Programming | * functional style uses the concept of declarative programming |

## Lambda

What is Lambda Expression?

* Lambda is a equivalent to a function (method) without a name.
* Lambdas are also refered as Anonymous functions which will have
  - → method parameters.
  - → method body
  - → return type
* Lambdas are not tied to any classes like regular method
* lambda can also be assigned to a variable and can be passed around as a parameter.

Syntax

```
( ) ----> { }
```

( ) ↑ lambda i/p parameter; ----> ↑ Arrow; { } ↑ Lambda Body.

Usages

* Lambdas are mainly used to implement functional Interfaces (or) SAM (Single Abstract Method Interfaces).
* functional Interfaces are annotated by @FunctionalInterface annotation.

Example:-

```
@FunctionalInterface
public interface Comparator<T>{
    int compare(T o1, T o2);
}

@FunctionalInterface.
public interface Runnable {
    public abstract void run();
}
```

Lambda in Practise

() → Single Statement (or) Single Expression.

// curly braces are not needed

() → {<Multiple Statements>};

// curly braces are needed for multiple statements.

Functional Interface

* They exist since Java 1.0
* A functional Interface (SAM) is an interface that has exactly 1 abstract method.
* @FunctionalInterface is the annotation that was introduced as a part of JDK 1.8
* It is an optional annotation to signify an interface as a Functional Interface.

Basically to maintain a contract i.e., to have only 1 abstract method.



New Functional Interfaces in Java8

There are 4 primary functional interfaces that were added i.e.,

1) Consumer
2) Predicate
3) Function
4) Supplier.

Each and every interface have their own extensions.

1) Consumer - BiConsumer
2) Predicate - BiPredicate
3) Function - BiFunction, UnaryOperator, BinaryOperator

Consumer Interface

```
@FunctionalInterface.
public interface Consumer<T>{
    void accept(T t);
    default Consumer<T> andThen
    (Consumer<? super T> after) {
        Objects.requireNonNull(after);
        return (T t) -> {accept(t);
                         after.accept(t);};
    }
}
```

where t => @param i.e., input argument.

* Consumer interface accepts only 1 Input.

BiConsumer Interface

* It is an extension of Consumer Interface.

* We can actually accept 2 parameter / 2 inputs in the BiConsumer Interface.

Predicate Consumer

```
public interface BiConsumer
  <T, O> {
    void accept(T t, O o)
}
```

Predicate Consumer

```
public interface Predicate<T>
{  boolean test (T t);
   // other default methods
}.
```

BiPredicate Consumer

```
public interface BiPredicate
  <T, U> {
    boolean test(T t, U u);
    // other default methods;
}.
```

Function Interface

```
public interface Function<T, R>{
  R apply (T t);
  // other static default methods.
}.
```

(1 i/p, 1 o/p)

where T is the i/p
R is the output.

Bifunction Interface (2 i/p, 1 o/p)

```
public interface BiFunction<T,U,R>{
  R apply (T t, U u);
  // other default methods : }
```



UnaryOperator Interface

```
public interface UnaryOperator<T>
  extends Function<T, T> {
  static <T> UnaryOperator<T>
  identity() { return t -> t; }
```

Here Unary operator interface extends functional Interface. WKT Functional Interface, we have only abs method where it takes 2 params, 1 is i/p and other is o/p, i.e., the types are different, say i/p is List<Students> and o/p is Map<String, Double>.

Where as, say when both i/p and o/p type is of same then, we can use Unary Operator Interface.

Binary Operator Interface

```
public interface BinaryOperator<T>
extends BiFunction<T, T, T> {
public static <T> BinaryOperator
<T> minBy (Comparator<? super
T> comparator) {
   Objects.requireNonNull(comparator);
   return (a, b) -> comparator
   .compare(a, b) <= 0 ? a : b;
} // minBy() - static method.
```

When the i/ps and the o/ps are of the same type, then you can make use Binary Operator that extends BiFunction.

## Supplier - Functional Interface (7)

```
public interface Supplier<T> {
    T get(); // returns something to the caller
}
```

basically returns a type to the caller.

* Exactly opposite to consumer Functional Interface.

Consumer - takes i/p
returns none

Supplier - takes no i/p
returns o/p.

## Method Reference

* Introduced as part of Java 8, and its purpose is to simplify the implementation Functional Interface.
* Shortcut for writing the lambda Expressions
* Refer a method in a class.

Syntax

```
ClassName :: instance-methodName
ClassName :: static-methodName
Instance :: methodName.
```

Where can we use it?

* Lambda expressions refering to method directly.

Example - Using (λ)

```
Function<String, String>
toUpperCase Lambda = (s) -> s.toUpperCase();
```

## Using method reference (8)

```
Function<String, String>
toUpperCaseMethodRef =
    String::toUpperCase;
```

Where we cannot use it?

The place we are actually implementing our own logic we cannot use it.

Example

```
Predicate<Student>
predicateUsingLambda =
(s) -> s.getGradeLevel()>=3;
```

## Constructor Reference

* Introduced as a part of Java 1.8.

Syntax :-

```
ClassName::new.
```

Example :- Creates a brand new object when this supplier get() is called.

```
Supplier<Student> stdSupplier
    = Student::new;
```

Invalid :-

```
Student student =
    Student::new // compilation error.
```

when Student::new is used Student class must have a ~~default~~ empty constructor else will have a compilation error.

## Lambdas & local variables

Here, it is all about the local variable and the restrictions that local variable has with the lambda functions.

## Local Variable

* Any variable that is declared inside a method is called as a local variable.
* Lambda have some restrictions on using local variables :-
  → Not allowed to use the same local variable name as lambda parameters or inside lambda body.
* → Not allowed re-assign a value to a local variable. in the lambda expression.
* No restrictions on instance variable in the lambda expression

[NOTE :- Variable used in lambda expression should be final or effectively final.]

Effectively final

* Lambda's are allowed to use local variables but not allowed to modify it even though they are not declared final. This concept is called as Effectively final.
* Prior to Java 8, any variable that is used inside the anonymous class should be declared "final" - saves keystroke

Advantages

* Easy to perform concurrency operations
* Promotes Functional Style programming over Imperative style programming.



## Streams

Intro :-

* Introduced as a part of Java 8.
* Main purpose is to perform some operations on Collections
* Parallel operations are easy to perform with Streams API without having a spawn a multiple threads.
* Streams API can be also used with arrays or any kind of I/O.

What is a Stream ?

Stream is basically a sequence of elements which can be created out of collections such as List (or) Arrays or any kind of I/O resource and etc.

```
List<String> name = Arrays.asList(
"Adam", "Dan", "Jenny");
name.stream()
// creates a stream.
```

[Series stream] →

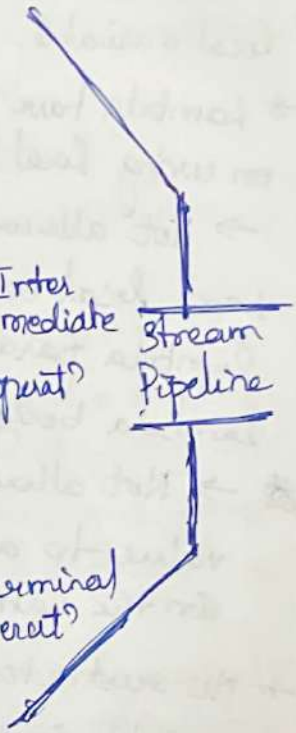
```
List<String> name =
Arrays.asList("Adam", "Dan",
"Jenny");
name.parallelStream();
```

[Parallel stream] →

How stream API works ?

P. T. O

| Stream API Operations | What's Happening? | Output |
|---|---|---|
| StudentDataBase.getAllStudent().stream(); | sequence: Stud-ent \| Std \| Std \| Std \| Std \| Std | ① Stream<Student> |
| filter(Student -> student.getGradeLevel() >= 3); | (One by One through filter) stream of students: Std \| Std \| Std \| Std \| Std | ② Stream<Student> |
| filter(Student -> student.getGPA() >= 3.9); | (One by One through filter) filtered stream of students: Std \| Std \| Std \| Std | ③ Stream<Student> |
| collect(Collectors.toMap(Student::getName(), Student::getActivities)); | Student \| Student | ④ Stream<Student> |




**Intermediate Operat°** :- The operation that is been happening b/no the .stream() and .collect(), which includes many methods such as .filters() and many.

**Terminal Operat°** :- Terminal Operator is the one that is executed at last, it includes many methods like .collect().

**collect()** :- collect is the method that collects all the intermediate processes stream and converts it to the final desired o/p which we are looking for, such as methods, .toMap().

collect() method is the one who starts the pipeline because if there was no collect method, then, the processed stream would not be able to get into any desired form of output i.e., there will not be any start of the pipeline.

**NOTE** ⇒ NO INTERMEDIATE OPERATIONS ARE INVOKED UNTIL THE TERMINAL OPERATION IS INVOKED. Hence, we tell that Stream() is a lazy one.

## Collection Vs Streams



| Collections | Streams |
|---|---|
| * Can add or modify elements at any point of time. Ex:- list → list. add(element) | * Cannot add or modify elements in the stream. It is a fixed data set. |
| * Elements in the collection can be accessed in any order. Use appropriate methods based on collection. Ex:- List → list. get(4); | * Elements in the stream can be accessed only in a sequence one by one. |
| * Collections are eagerly constructed | * Streams are lazily constructed. |
| * Collections can be traversed "n" number of times. | * Streams can be traversed only once. |
| * Performs External Iteration to iterate through the elements | * Performs Internal Iterat° to iterate through the elements |

## Stream API : map()

* map() = converts (transform) one type to another.
* Don't get confused this with Map Collection.

### flatMap()

* It converts one type to another as like map() method.
* It can transform one form to another form of the same type.
* But, they are used in the context of Stream where each element in the stream represents multiple elements.



Example:-

Each stream element represent multiple elements.

* Stream <List>
* Stream < Arrays > .etc.,

distinct(), count(), sorted()

distinct():- Returns a stream with unique elements

count():- Returns a long with total no of elements in the stream.

sorted():- Sort the elements in the stream.

### filter()

* They are used to filter the elements in the stream.
* Input to the filter is a Predicate function Interface.

### reduce()

* This is the terminal operation. Used to reduce the contents of a stream to a single value.
* It takes two parameters as an input.
  - → 1st param – default (or) initial value
  - → 2nd param – BinaryOperator <T>

```
pub static Integer reduceStream
(List<Integer> intList){
    return intList.stream()
        .reduce(1,(a,b)->a*b);
}
pub stat void main(str[] a){
    List<Int> = Array.
        .asList{1,3,5,7}
```

Operat°

.reduce (1, (a, b) → a * b);

where 1 is a identity value. (or) a starting value (or) a default value.

a ⇒ Resultant int

b ⇒ value from stream.

say i/p is {1, 3, 5, 7}

and identity value is taken "a" and stream 1st element is taken by "b".

(identity) (from stream)

a = 1 , b = 1 ⇒ a * b = 1

a = 1 , b = 3 ⇒ 3

a = 3 , b = 5 ⇒ 15

a = 15 , b = 7 ⇒ 105

final o/p = 105

i.e., 1 * 3 * 5 * 7 = 105.

we can actually use without Identity to avoid few problems by using as a return type.

Optional<Integer> instead of List<Integer>



## limit() skip()

* These three functions helps to create sub-streams.
* limit(n) → limits the "n" numbers of elements to be processed in the stream.
* skip(n) → skips the "n" number of elements from the stream.

## anyMatch(), allMatch(), noneMatch()

* All these functions takes in a predicate as an input and returns a Boolean as an output
* anyMatch() - Returns "true" if any one of the element matche the predicate, otherwise "false".
* allMatch() - Returns "true". if all the elements in the stream matches the predicate otherwise, "false".
* noneMatch() - O/p to allMatch() Returns "true", if non of the elements in the stream matches the predicate, otherwise "false".

## findAny() and findFirst()

* Both are used to find the element in the stream.
* The diff b/w anyMatch(), allMatch() & nonMatch() is that, all these methods return boolean value, where as findAny() & findFirst()

returns the actual element. (16)

* Both functions returns the result type "Optional<T>"
* findFirst() – Returns first element in the stream based on the condition.
* findAny() – Returns first element encountered in the stream.

## Stream API – Short Circuiting

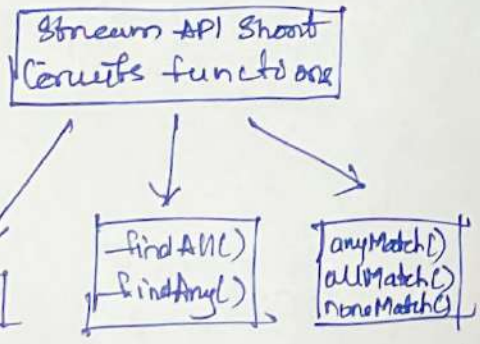
What is short circuiting?

Ex :-

```
if (boolean1 && boolean2){
    // body (AND)
}
```

If the 1st expression evaluates to false then the second exp won't even execute.

Same goes with (OR) as well and this concept is called as short circuit.




All these functions does not have to iterate the whole stream to evaluate the result.

## Stream API - Factory Methods (17)

of(), generate(), iterate().

* of() :- creates a stream of certain values passed to this method.

Example :-

```
Stream<String> stringStream
    = Stream.of("Adam", "Dan",
                "Julie");
```

* iterate(), generate()

They are used to create infinite streams.

Example :-

```
Stream.iterate(1, x -> x*2);
// creates infinite loop.
```

Example :-

```
Stream.generate(<Supplier>)
```

## NUMERIC STREAM

Numeric Streams are used to represent the primitive values in a stream.

They are of 3 types

* IntStream
* LongStream
* DoubleStream.

## Spring framework ①

It is a light, loosely coupled framework that is build on [illegible]

Specail features are:-

1) IOC [Inversion Of Control]
2) DI [Dependency Injection]

IoC ⇒ Creating and Managing the object.

DI ⇒ Creating an object and autowiring it to the required class and the created object is maintained in the Spring container [Object factory]

There are 2 main types of dep inj i.e.,

1) Constructor Injection.
define const in the class mark it @ component, come to xml and then define in the xml file as well.

2) Setter Injection
same process, but the xml tag used is different i.e,
< property =" ", value =" ">

## Bean Scope

Here in Spring framework, bean means object.
By default, the Scope of a bean is Singleton [returns single instance of a bean]

* prototype - returns new instance of a bean each time when required
* request - returns single instance for every HTTP request call.
* session :- Returns a single instance for every HTTP session. ②

Bean scope refers to the lifecycle of Bean that means when the bean will be instantiated, how long does the object live, and how many object will be created for that bean throughout.
Basically it controls the instance creation of the bean and it is managed by the spring container.

## Annotations

@ Component.
Spring component is an annotation used to denote a class as a component. It means the spring framework will autodetect these classes for dependency injection when annotation-based configuration and classpath scanning is used.

@Autowired
This annotation is used for automatic dependency injection. Spring framework is built on dependency injection and we inject the class dependency through the Spring bean configuration file.
To use @Autowired, it mandatory to use @Component of a specified class.

## @Bean ③

This annotation is applied on a method to specify that it returns a bean to be managed by Spring context.

## @Configuration

Basically in Spring, we have 3 types of configuration i.e.,
- → XML based
- → Annotation - based
- → Java based

We use @Configuration in order to do java-based configuration. We can do that taking any of the Java Class, Annotate it and then can declare beans in there methods and may be processed by the Spring container to generate bean definitions & service requests for those beans at runtime.

## @ComponentScan

With Spring, we use @ComponentScan annotation along with the @Configuration annotation to specify the package that we want to be scanned.

@ComponentScan without arguments tells Spring to scan the package & all of its sub-packages

## @EnableAutoConfiguration ④

This annotation tells Spring Boot to "guess" how you want to configure Spring, based on the jar dependencies that you have added.

Say, the Spring-boot-starter-web dependency is added to classpath leads to configuring Tomcat & Spring MVC, the auto-configuration assumes that you are developing a web application and sets up Spring accordingly.

## @SpringBootApplication

It is a combination of, 3 main annotations i.e.,
- @SpringBootConfiguration
- @EnableAutoConfiguration
- @ComponentScan.

## @Controller

The controller annotation indicates that a particular class serves the role of a controller in a WebApp development. It can be applied to classes only. It's used to mark a class as a web request handler.

## @RequestBody

By using request body annotation you will get your values mapped with the model you created in your system for handling any specific API call.

→ Demanding for a Body during call.

## @ResponseBody

@ResponseBody tells a controller that the object returned is automatically serialized into JSON and passed back into the HttpResponse object

| @Controller | @RestController |
|---|---|
| ⇒ returns complete HTML page i.e., model & view | ⇒ returns JSON format |
| → Introduce in Sp 2.5 | → Introduced in Sp v4. |

@RestController ⇒ @Controller + @ResponseBody

## @RequestParam

When we need any response wrt to a specific data, then we have to pass the param either along with request.

## Lazy fetching in Hibernate

It means that fetching and loading the data, only when it's needed, from a persistent storage like DB.

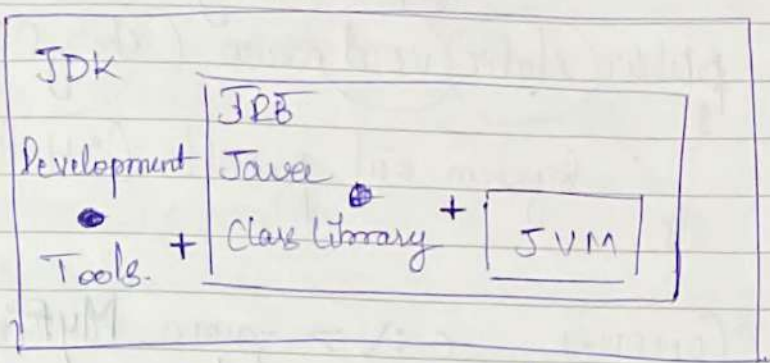
→ It improves the performance of data fetching and significantly reduces the memory footprint.

i.e., they do not load as soon as the code is executed. They are fetched only when it is queried.

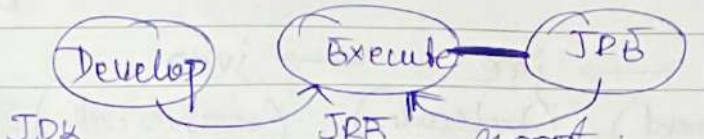
## Eager fetching in Hibernate

It fetches all the data and the related data as soon as they are queried.

# CORE JAVA




JDK ⇒ Java Development kit ⟶ Debug / compile
(library classes) JRE ⇒ Java Runtime Environment ⟶ Execute.
JVM ⇒ Java Virtual Machine.




first.java ⟶ javac ⟶ First.class.
↓ compile (development tool)
java first (runtime envi)

1. Set a path
2. make directory (md filename)
3. c:\ dir
4. c:\ cd filename

Notepad code ⟶ classcode (bytecode)
filename.java — done by javac (a seperate file)

Compile ⇒ javac Myfile.java
Run ⇒ java Myfile

to see the code ⇒ type Myfile.java
to see compiler code ⇒ type Myfile.class

import java.lang.*; package

class MyFirstProg { must be same as MyFirstProg.java file name

public static void main (String[] args) return nothing, command line argument

{

System.out.println ("Hello world");

}

}.

COMPILE C:\> javac MyFirstProg.

MyFirstProg.class

RUN C:\> java MyFirstProg

jdk (development) → jre (bytecode) → jvm (machine code)

STATIC → can access the method without any object just by using class name

## Reading from keyboard

Scanner sc = new Scanner (System.in) — a special method used to initialize an object; Constructor

Class. reference/object.

int read = sc.nextInt() (reads a number)

hasNextInt() / hasNextFloat() → To check the type of data entered

o/p → Boolean T/F

Light, loosly coupled framework that is

Spring ← JEE

Inversion Of Control (IoC)
Outsourcing the creating and managing the Object.

Dependancy Injection (DI)
Outsource the ~~creation~~ construct & injection of obj ~~and inject the obj~~ ~~required~~ side class
Outsource the construction & injection of the object to the external entity.

Spring
[Obj factory]

Spring container (Primary fun)
* Create & manage Objects (IoC)
* Inject Object's dependencies (DI)

There are many types of dep inject, but mainly there are 2 :-
1) Constructor Injection
2) Setter Inject

Example

1) <bean : id = "myfortuneService"
class = " ~~Class path~~ : fullyqualified ClassName >
</bean>
} applicationcontext.xml

⇒ Inter Springframework.
HappyfortuneService ● myfortuneService =
new HappyfortuneService();

2) < bean id = " mycoach "
class = " classpath " >
< constructor-args ref = " myfortuneService " >
</bean>

⇒ BasketballCoach ~~my~~coach = new
BasketballCoach (myfortuneService);

an id = "myCricketCoach"

class = "fullyQualifiedClassName"

< property name = "fortuneService" ref = "myFortuneService" />

Refer Defined in the constructor

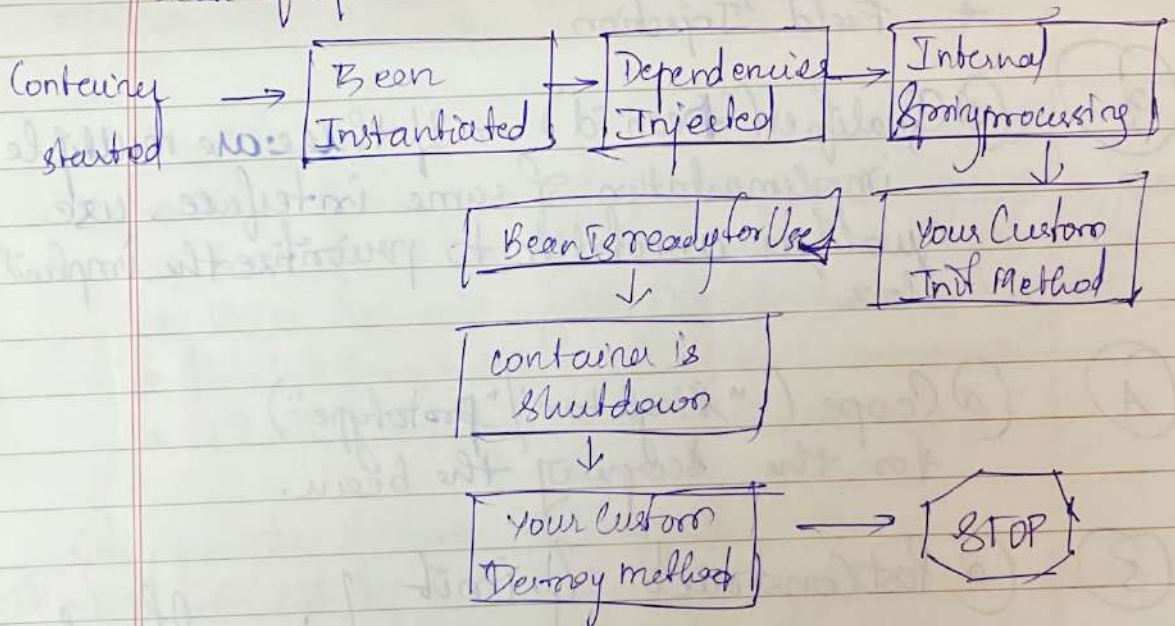
public void setFortuneService(...)

→ CricketCoach myCricketCoach = new CricketCoach();
myCricketCoach.fortuneService(myFortuneService)

**Bean Scope** [Default Scope: Singleton]

- Scope refers to the lifecycle of a bean
- How long does the bean live?
- How many instances are created?
- How is the bean shared?

Bean lifecycle.



## Java Annotations

- Special labels / markers added to Java Class
- Provide meta-data about the class
- Processed at compile time / run time for special processing.

1. @Component ⇒ creates a bean and manages it for a class.

2. @Autowired ⇒ Spring will look for a class that matches the property
   Say :- matches by type: class / Interface.
   If matched then it will inject the dependancy

   There are 3 types of Autowiring Inject types.
   - Constructor Injection
   - Setter Injection
   - Field Injection

3. @Qualifier("bean id") ⇒ If there are multiple implementation of same interface, use Qualifier annotation to prioritize the implement class.

4. @Scope ("singleton" / "prototype")
   for the scope of the bean.

5. @PostConstruct & init
   @PreDestroy & destroy
   } bean lifecycle.

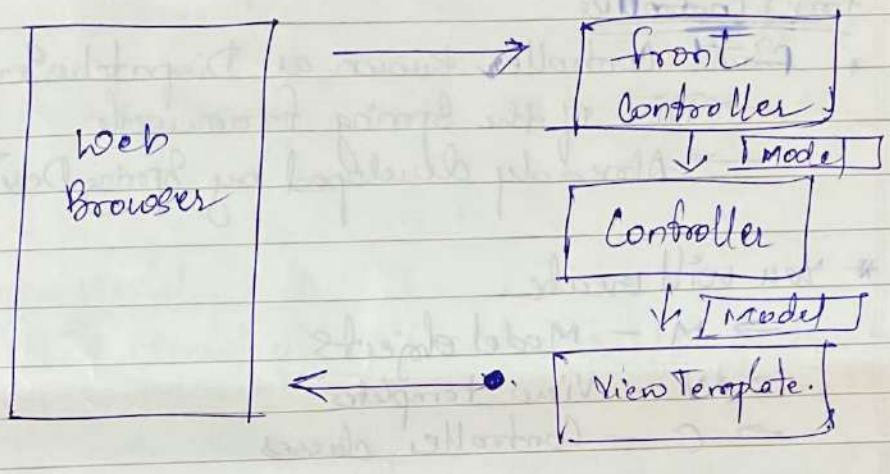
6. @Configuration (Spring container) generates the bean by itself and request at runtime.

7. @Component bean -

# Spring MVC

* Framework for building web applications in Java
* It is based on Model-View-Controller design pattern.
* Leverages features of the core Spring framework (IoC, DI).

## MVC

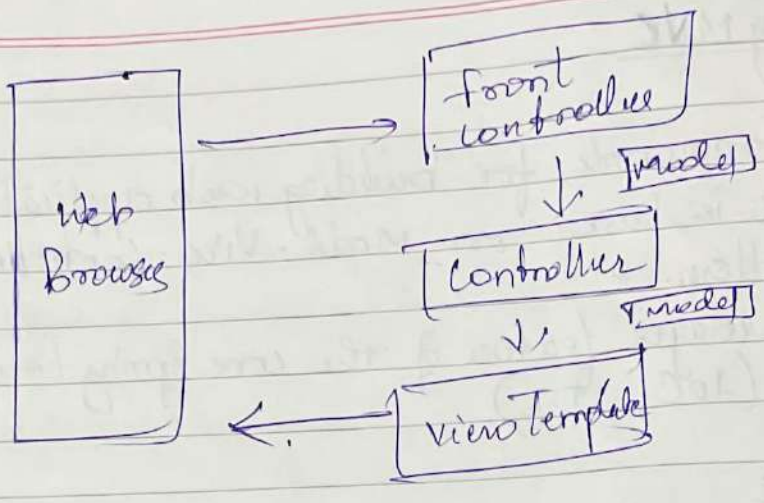



## MVC Benefits

* The Spring way of building web app UIs in Java
* Leverage a set of reusable UI components
* Help manage application state for web request
* Process form data: validation, conversion etc.
* Flexible configuration for the view layer.

## Components of Spring MVC Application

* A set of web pages to layout UI components
* A collection of Spring beans (controllers, services, etc...)
* Spring configuration (XML, Annotations / Java)




Front Controllers

* Front Controller known as DispatcherServelet
  → Part of the Spring framework.
  → Already developed by Spring Dev Team.

* You will create.
  → M – Model objects
  → V – View templates
  → C – Controller classes

Controllers (Business logic)

When the frontController has a request, it delegates the req to the controller
  → Contains your business logic
  → Handle the request (get, post, del, et
  → Store/retrive data (db, web Services)
  → Place data in model
  → Send data to appropriate view template

Model passes to View.

* This is the place where you will have your data
* Store/retrive data via backend systems
  → like database, web services etc
  → Use Spring bean if needed.
* Place your data in the model (Shipment)
  → Data can be any Java Obj/collection.

View

* Spring MVC is flexible → supports many view templates (Thymeleaf, Groovy etc.)
* Most commonly is JSP & JSTL
* Developer creates a page to display data.

(8) @ Controller.
⇒ inherits from @Component .. supports scanning.

(9) @RequestMapping("/") ⇒ mapping URL

Spring Model

* The Model is a container for your application data.
* In your controller
  → You can put anything in the model
  → String, Object, info from database, etc.
* Your View Page (JSP) can access from the model.

(10) @RequestParam ("studentName") String theName
Automatically binds to the attribute in the HTML page - No need of using getparameter

Spring MVC Form Tags

* form:form — main form container
* form:input — text field
* form:textarea — multi-line text field
* form:checkbox — check box
* form:radiobutton — radio buttons

<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>

(11) @ ModelAttribute → to retrive all the datas of the form using a bean.

Ex:- pub String abc(@ModelAttribute("student") Student theStudent)

(12) Validation Annotations

Part of Hibernate validator.

@NotNull - checks that the annotated value is not null.
@Min - Must be a number >= value
@Max - must be a number <= value
@Size - Size must match the given size
@Pattern - Must match a regular expression pattern
@Future/@Past - Date must be in future/past of given date.

etc...

(13) @Valid → To validate a bean and store the result in BindingResults

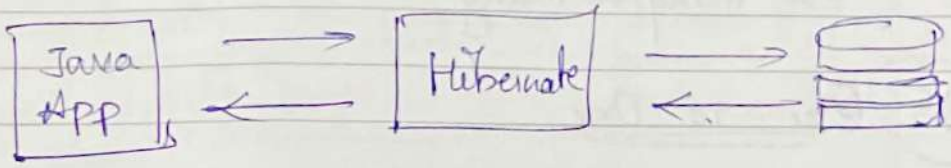
(14) @InitBinder

Advanced

* Annotation works as a pre processor
* It will pre-process each web request to our controller
* Method Annotated with @InitBinder is executed 1st.

# Hibernate

Hibernate is a framework that helps in persisting (or) saving the Java objects in the database (data)




Benefits

* Hibernate handles all of the low level SQL
* Minimizes amount of JDBC code you have to develop.
* Hibernate provides the Obj-to-Relational Mapping (ORM)

Main Roles

| | Class | Description |
|---|---|---|
| 1) | Session factory | Reads the hibernate config file<br>Creates Session Objects<br>Heavy-weight object<br>Only create once in your app |
| 2) | Session. | Wraps a JDBC Connect<br>Main Obj used to save/retrive objects.<br>Short lived objects<br>Retrived from Session factory. |

## ORM

Obj-Relational mapping provides an obj-oriented layer b/w relational databases and obj-oriented programming languages without having to write SQL queries. Helps reducing boilerplate & speeding development time.

## Hibernate Advance mapping

There are 3 types
- → One - to - One
- → One - to Many / Many to One
- → Many to Many.

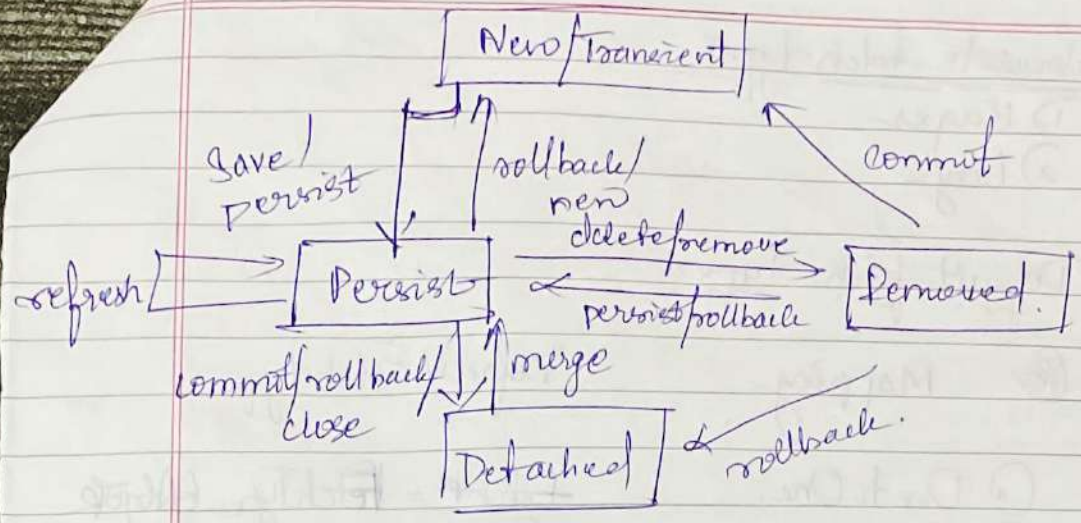
15 ## One - to - One

@ One to One
@ Join Column (name = "Primary key_id")
private InstructorDetail instructorDetail.
(FK Table Name)

Column Name

## Entity Lifecycle

| Operations | Description |
|---|---|
| Detach. | If entity is detached, it's not associated with Hibernate session. |
| Merge | If instance is detached from session, then merge will reattach to session. |
| Persist | Transitions new instances to managed state. Next flush / commit will save in db |
| Remove | Transitions managed entity to be removed. Next flush / commit will delete from db |
| Refresh | Reload / synch object with data from db Prevents stale data. |




@One-to-One Cascade types.

PERSIST → If entity is persisted/saved, related entity will also be persisted.

REMOVE → If entity is removed/deleted, related entity will also be deleted.

REFRESH → If entity is refreshed, related entity will also be refreshed.

DETACH → If entity is detached, (not associated with session) then related entity will also be detached

MERGE → If entity is merged, then all related entity will also be merged

ALL → All of above Cascade types together.

16

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
@Column(name="id")
private int id;
```

## Hibernate fetch types

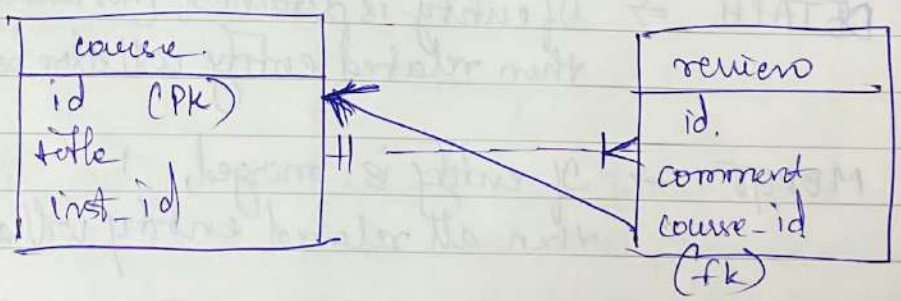
1) Eager
2) Lazy

## Default fetch types

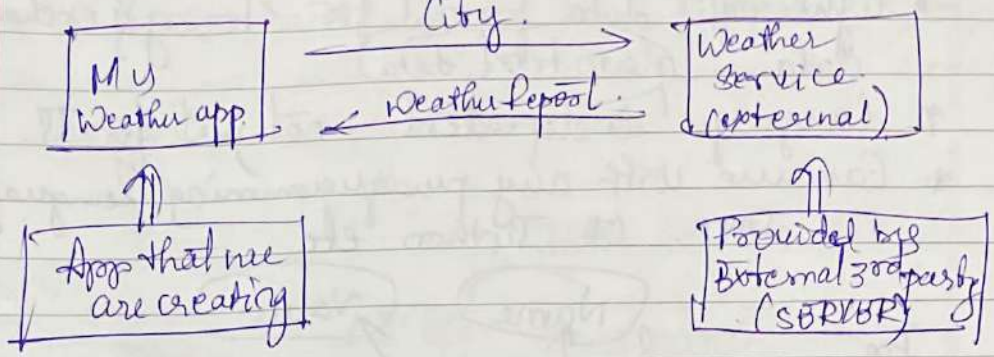
| Mapping | Default Fetch type |
|---|---|
| @ One to One | ~~fetch~~ = ~~Fetch~~Type.~~EAGER~~ |
| @ One To Many | FetchType.LAZY |
| @ Many to One | FetchType.~~EAGER~~ |
| @ Many to Many | FetchType.LAZY |

17 @JoinColumn(name = "course_id")

Here the @JoinColumn tells Hibernate to
- Look at the course_id column in the "review" table
- Use this info to help find associated reviews for a course.



# Spring REST




(1) How will we connect to the Weather Service?

Ans
* We can make REST API calls over HTTP.
* REST - Representational State Transfer.
* It is just a lightweight approach for communicating b/w application.

(2) What Programming language do you use?

Ans
* REST is language independent
* The client applicat° can use any programming lang. Eg:- Java, C++, Python, Ruby etc.
* The server applicat° can also use any programming language.

(3) What is the data format?

Ans
* REST Applicat° can use any dataformat.
* Commonly use XML and JSON./HTML
* JSON is most popular & modern.
* JavaScript Object Notation.
* eXtensible Markup Language.

JSON - JavaScript Object Notation

* Lightweight data format for storing & exchanging data -- plain text data.
* Language independent -- not just for JS.
* Can use with any programming language:- Java, C#, Python etc.

Ex

```
{ "id" = 14,
  "firstName" : "Mario",
  "lastName" : "Rossi",
  "active" : true
}
```

(Name → "id", Value → 14)

JSON Name

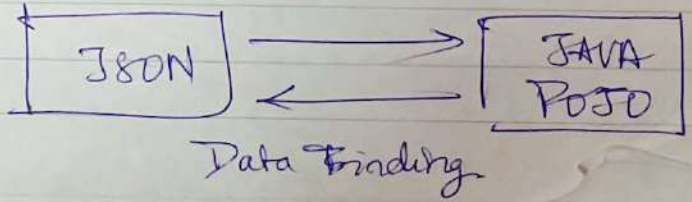
* Curly braces define Objects in JSON
* Object members are name/value pairs → Delimited by colons.
* Name is always in double-quotes

JSON Value

* Numbers : No quotes
* String : In double quotes.
* Boolean : true/false.
* Nested JSON objects
* Array
* null.

Java JSON Data Binding

* Data binding is the process of converting JSON data into a Java POJO.




Data Binding

- Spring uses the Jackson Project behind the scenes
- Jackson handles data binding b/w JSON & Java POJO
- Jackson will call appropriate getter/setter method during data binding.

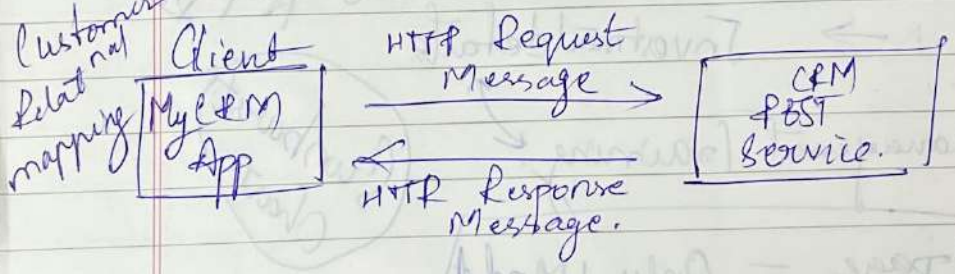
JSON —setter→ JAVA POJO

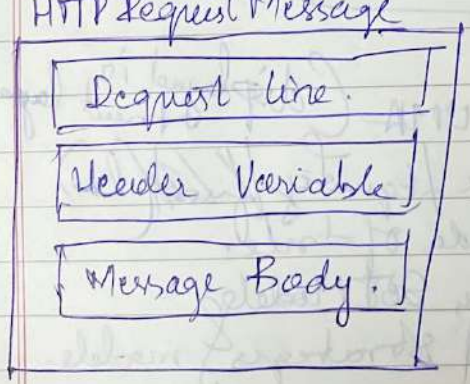
JSON ←getter— JAVA POJO.

## REST Over HTTP

- Most common use of REST is over HTTP.
- Leverage HTTP methods for CRUD Operation

| HTTP Method | CRUD Operation |
|---|---|
| POST | Create a new entity |
| GET | Read a list of entities / single entity |
| PUT | Update an existing entity |
| DELETE | Delete an existing entity. |




Request line → the HTTP Command GET, POST etc

Header Variable → request metadata (Additional data about the req)

Message body → contents of the message.