**Collections Framework** (introduced in Java 1.2 version)

* The collection in Java is a framework that **provides an architecture** to store and manipulate the group of objects.
* Java collections can **achieve all the operations that you perform** on a data such as searching, sorting, insertion, manipulation, and deletion.
* Java collection framework provides many interfaces (Set, List, Queue, Deque) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet)…/
* Hierarchy of collection Framework (The **java.util package** contains all the classes and interfaces for the Collection framework.)



**Iterable** Interface: This is the super Interface of Collection Interface.

The Iterable interface is the root interface for all the collection classes. The Collection interface extends the Iterable interface and therefore all the subclasses of Collection interface also implement the Iterable interface. It contains only one abstract method. i.e.,

Iterator<T> iterator()

\*<T> specifically stands for generic type. (T is for Type)

All Known Subinterfaces: BeanContext, BeanContextServices, BlockingQueue<E>, **Collection**<E>, List<E>, Queue<E>, Set<E>, SortedSet<E>

All Known Implementing Classes: AbstractCollection, AbstractList, AbstractQueue, AbstractSequentialList, AbstractSet, ArrayBlockingQueue, ArrayList, AttributeList, BeanContextServicesSupport, BeanContextSupport, ConcurrentLinkedQueue, CopyOnWriteArrayList, CopyOnWriteArraySet, DelayQueue, EnumSet, HashSet, JobStateReasons, LinkedBlockingQueue, LinkedHashSet, LinkedList, PriorityBlockingQueue, PriorityQueue, RoleList, RoleUnresolvedList, Stack, SynchronousQueue, TreeSet, Vector

**Collection** Interface: The Collection interface is the interface which is implemented by all the classes in the collection framework. It declares the methods that every collection will have. In other words, we can say that the **Collection Interface** builds the foundation on which the collection framework depends.

A collection represents a group of objects, known as its elements. Some collections allow duplicate elements and others do not. Some are ordered and others unordered. The JDK does not provide any direct implementations of this interface: it provides implementations of more specific sub interfaces like Set and List.

Methods: <https://www.cs.mun.ca/java-api-1.5/api/java/util/Collection.html>

**Before Start Learning: Good to Know->**

**3Cs**

1. c-collections – concept of collection framework
2. C-Collections—Class of java.util package
3. C-Collection –Interface of java.util package

**Implements & Extends**

**Implements** -> An Interface needs to be implemented->only Class can implement an Interface or multiple Interfaces (Java support multiple implementations)

**Extends** ->As per OOP concept Java allows a Class (child) to **extend** another Class (parent) AND an Interface **extends** another Interface (a class can’t extend multiple classes)

**List Interface**

->List interface is the child interface of Collection interface.

->It inhibits a list type data structure in which we can store the ordered collection of objects.

->It can have duplicate values.

->List interface is implemented by the classes ArrayList, LinkedList, Vector, and Stack.

To instantiate the List interface, we must use :

1. List <data-type> list1= new ArrayList();

2. List <data-type> list2 = new LinkedList();

3. List <data-type> list3 = new Vector();

4. List <data-type> list4 = new Stack();

There are various methods in List interface that can be used to insert, delete, and access the elements from the list.

The **classes that implement** the List interface are given below.

**ArrayList** Class

* implements the List interface
* uses a dynamic array
* duplicate element
* different data types (Heterogeneous elements are allowed)
* maintains the insertion order (insertion order will be preserved ->will print as you add)
* non-synchronized
* elements stored in the ArrayList class can be randomly accessed (by using get(index)
* stores values in the basis of index
* null insertion is possible (String)

**A simple example:**

public static void main(String args[]){

ArrayList<String> list=new ArrayList<String>();//Creating arraylist

list.add("Sohag");//Adding object in arraylist

list.add("Tufayel");

list.add("Sharif");

//Traversing list through Iterator

Iterator itr=list.iterator();

while(itr.hasNext()){

System.out.println(itr.next());

**Need to cover:**

->add(), size(), get()

->get all values by using for loop, iterator()

->generic and non generic

->user-defined class obj

->addAll()

->removeAll()

->retainAll()

->how array is fixed but ArrayList not

->why ArrayList is slow

\*in case of adding element and removing element🡪for shifting needs time

->ArrayList is better choice

\*for data retrieval

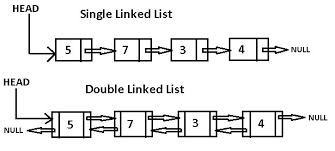
ar.get(0);----for example it will take 1 m.sec

ar.get(1000)-- -- it will take 1 m.sec too

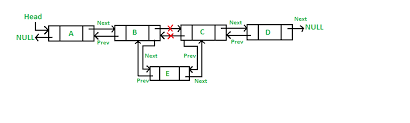
RandomAccess (marker Interface- no method available) Interface is implemented by ArrayList

**LinkedList**

* LinkedList implements the Collection interface.
* It uses a doubly linked list internally to store the elements.



* It can store the duplicate elements.
* It maintains the insertion order
* not synchronized.
* In LinkedList, the manipulation is fast because no shifting is required.



**Some Commonly used methods**

addFirst()

addLast()

get(0)🡪specific value

set(0,”Rahman”)—to set(replace) specific index value

removeFirst()

removeLast()

We can print by using: advance for loop, iterator, while loop

**Simple example:**

public static void main(String args[]){

LinkedList<String> al=new LinkedList<String>();

al.add("Sohag");

al.add("Orfat");

al.add("Sharif");

Iterator<String> itr=al.iterator();

while(itr.hasNext()){

System.out.println(itr.next());

**Vector:**

* implements the List interface
* uses a dynamic array
* different types of value
* synchronized and contains many methods that are not the part of Collection framework.

public static void main(String args[]){

Vector<String> v=new Vector<String>();

v.add("Sohag");//Adding object

v.add("Orfat");

Iterator<String> itr=v.iterator();

while(itr.hasNext()){

System.out.println(itr.next());

**Commonly used methods:**

1.boolean **add(Object obj)**: -> true if the specified element is added successfully into the Vector, otherwise it returns false.

2.void **add(int index, Object obj):**

3.boolean **addAll(Collection c)** This method appends all of the elements

in the specified Collection to the end of this Vector. –we can copy ArrlyList value

4. boolean **addAll(int index, Collection c)** This method inserts all of the elements in the specified Collection **into** this Vector at the specified.

5.void **clear()** This method removes all of the elements from this vector

6.Object **clone()** This method returns a clone of this vector

7. boolean **contains(Object o):** This method returns true if this vector contains the specified element.

**8.void ensureCapacity(int minCapacity):**This method increases the capacity of this vector, if necessary, to ensure that it can hold at least the number of components specified by the minimum capacity argument .

9. Object **get(int index):**This method returns the element at the specified position in this Vector

10. int **indexOf(Object o):** This method returns the index of the first occurrence of the specified element in this vector, or -1 if this vector does not contain the element.

11. boolean **isEmpty():** This method tests if this vector has no components

**# Stack**

* The stack is the **subclass of Vector.**
* models and implements Stack data structure
* It implements the last-in-first-out data structure, i.e., Stack.
* The stack contains **all of the methods of Vector class** and also provides its methods like boolean push(), boolean peek(), boolean push(object o), which defines its properties.

**Methods in Stack class**

1.Object **push(Object element)** :🡪**to Add element** (Pushes an element on the top of the stack.)

2.Object **pop()** :🡪**to delete element**( Removes and returns the **top element of the stack**. An ‘EmptyStackException’ exception is thrown if we call pop() when the invoking stack is empty.)

3.Object **peek()** **:-> to find Element** Returns the element on the top of the stack, but does not remove it.

4.boolean **empty()** : It returns true if nothing is on the top of the stack. Else, returns false.

5. int **search(Object element)** : It determines whether an object exists in the stack. If the element is found, it returns the position of the element from the top of the stack. Else, it returns -1.

public static void main(String args[]){

Stack<String> stack = new Stack<String>();

stack.push("Sohag");

stack.add("Sharif");

stack.push("Orfat");

//Print

Iterator<String> itr=stack.iterator();

while(itr.hasNext()){

System.out.println(itr.next());

}

//to check the position

System.out.println(stack.search("Sohag"));

//to find the peek element as per LIFO

System.out.println("Find the element using peek():"+stack.peek());

//poping out peek element and find the latest peek element

stack.pop();

System.out.println("Find the element after using peek() then peek():"+stack.peek());

//popping out all elements and check stack is empty or not

stack.pop();

stack.pop();

System.out.println("It will returns true because no element in our stack: "+stack.empty());

**#Queue Interface**

* Queue interface maintains the first-in-first-out order.
* It can be defined as an ordered list that is used to hold the elements which are about to be processed.
* There are various classes like PriorityQueue, Deque, and ArrayDeque which implements the Queue interface.

**Queue interface can be instantiated as:**

Queue<String> q1 = new PriorityQueue();

Queue<String> q2 = new ArrayDeque();

**PriorityQueue**

* PriorityQueue class implements the Queue interface
* holds the elements or objects which are to be processed by their priorities
* PriorityQueue doesn't allow null values to be stored in the queue

public static void main(String args[]){

PriorityQueue<String> queue=new PriorityQueue<String>();

queue.add("Sohag");

queue.add("Tufayel");

queue.add("Orfat");

System.out.println("head:"+queue.element());

System.out.println("head:"+queue.peek());

//iterating the queue elements:

Iterator itr=queue.iterator();

while(itr.hasNext()){

System.out.println(itr.next());

}

queue.remove();

queue.poll();

System.out.println("after removing two elements:");

Iterator<String> itr2=queue.iterator();

while(itr2.hasNext()){

System.out.println(itr2.next());

**Deque Interface**

Deque interface extends the Queue interface. In Deque, we can remove and add the elements from both the side. Deque stands for a double-ended queue which enables us to perform the operations at both the ends.

Deque can be instantiated as:

Deque d = **new** ArrayDeque();

public static void main(String[] args) {

//Creating Deque and adding elements

Deque<String> deque = new ArrayDeque<String>();

deque.add("Sohag");

deque.add("Orfat");

deque.add("Sharif");

//Traversing elements

for (String str : deque) {

System.out.println(str);

Methods:

Element()-🡪retrive peek element

Peek()🡪retrive peek element

Remove()->delete peek (as per fifo)

Poll()->delete last element

**Set Interface**

Set Interface in Java is present in java.util package. It extends the Collection interface. It represents the unordered set of elements which doesn’t allow us to store the duplicate items. We can store at most one null value in Set. Set is implemented by **HashSet**, **LinkedHashSet**, and **TreeSet**.

Set can be instantiated as:

Set<data-type> s1 = new HashSet<data-type>();

Set<data-type> s2 = new LinkedHashSet<data-type>();

Set<data-type> s3 = new TreeSet<data-type>();

**HashSet**

* HashSet class implements Set Interface.
* It represents the collection that uses a hash table for storage.
* Hashing is used to store the elements in the HashSet.
* It contains unique items.

public static void main(String args[]){

//Creating HashSet and adding elements

HashSet<String> set=new HashSet<String>();

set.add("Sohag");

set.add("Tufayel");

set.add("Orfat");

set.add("Sharif");

//Traversing elements

Iterator<String> itr=set.iterator();

while(itr.hasNext()){

System.out.println(itr.next());

**Commonly used Methods**

Add()

Clear()

Contains()—return true if contains

Remove()

Iterator()🡪Used to return an iterator over the element in the set.

isEmpty()->returns true if empty

size()

clone()-to copy

**LinkedHashSet**

* LinkedHashSet class represents the **LinkedList implementation of Set Interface**.
* It **extends the HashSet class** and **implements Set interface**.
* Like HashSet, It also contains unique elements.
* It maintains the insertion order and permits null elements.

public static void main(String args[]){

LinkedHashSet<String> set=new LinkedHashSet<String>();

set.add("a");

set.add("b");

set.add("c");

set.add("d");

set.add("c"); // no duplicate

Iterator<String> itr=set.iterator();

while(itr.hasNext()){

System.out.println(itr.next());

**Methods Used:**

Add()

Size()

Remove()

Contains()-returns true if contain

Clear()-clear all elements

**SortedSet Interface**

* SortedSet is the alternate of Set interface that provides a **total ordering on its elements**.
* The elements of the SortedSet are arranged in the increasing (ascending) order.
* The SortedSet provides the additional methods that inhibit the natural ordering of the elements.

The SortedSet can be instantiated as:

SortedSet<data-type> set = new TreeSet();

**TreeSet**

* Java TreeSet class implements the Set interface that uses a tree for storage.
* Like HashSet, TreeSet also contains unique elements.
* However, the access and retrieval time of TreeSet is quite fast.
* The elements in TreeSet stored in ascending order.

**Note: Need to study more onTreeSet**

public static void main(String args[]){

//Creating and adding elements

TreeSet<String> set=new TreeSet<String>();

set.add("a");

set.add("b");

set.add("c");

set.add("d");

//traversing elements

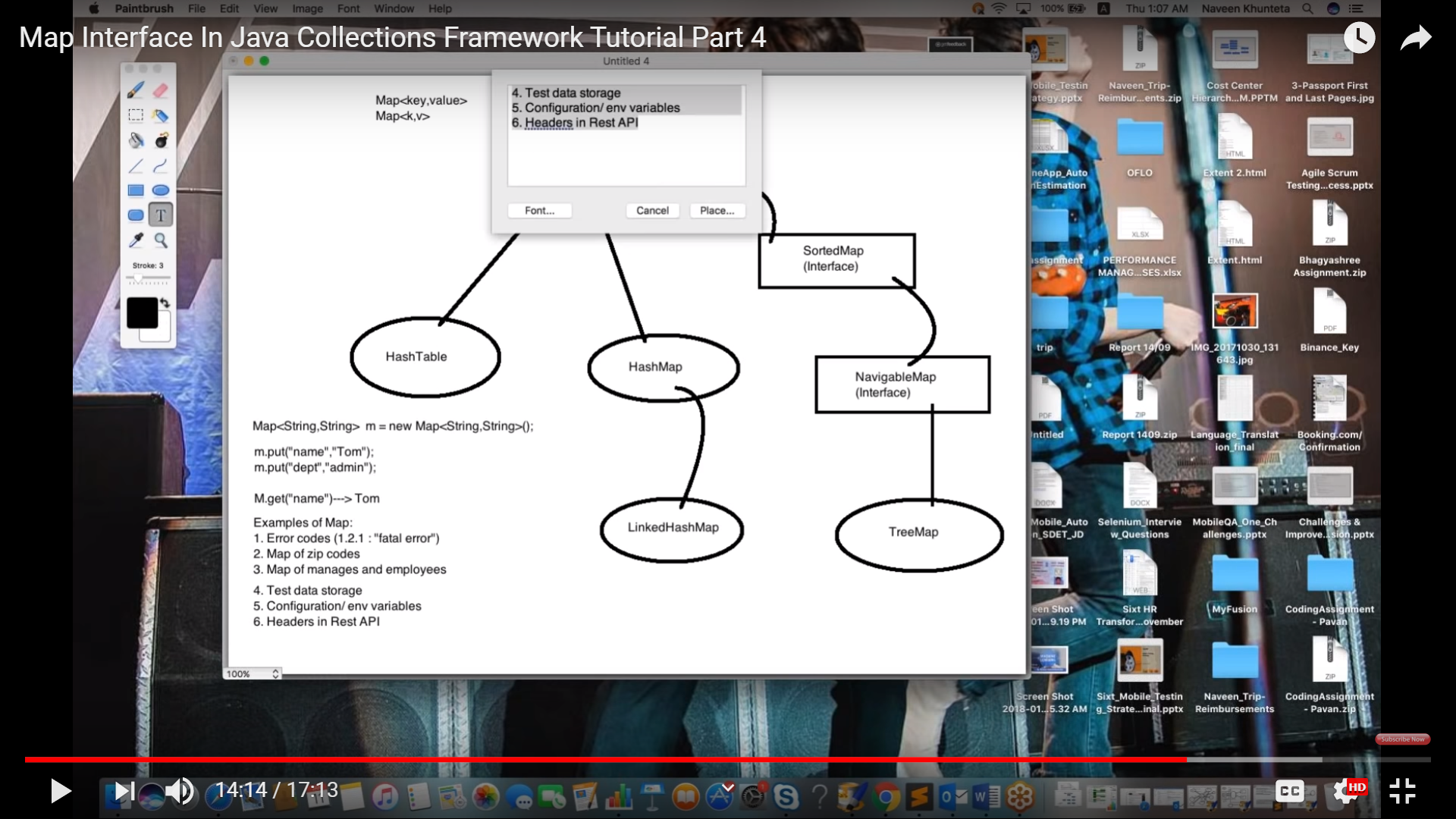
Iterator<String> itr=set.iterator();

while(itr.hasNext()){

System.out.println(itr.next());

**### Map Interface**

* **Part of collection framework series**
* **Implemented by different classes**



**# HashMap Class:**

**->implements Map Interface**

**->extends AbstractMap**

**->contains only unique element**

**->stores value in key value pair**

**->it may have one null key and multiple null values**

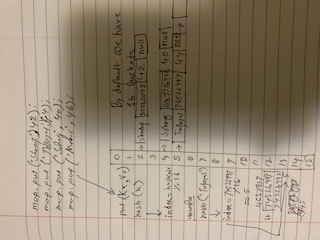
**->it maintains no order**

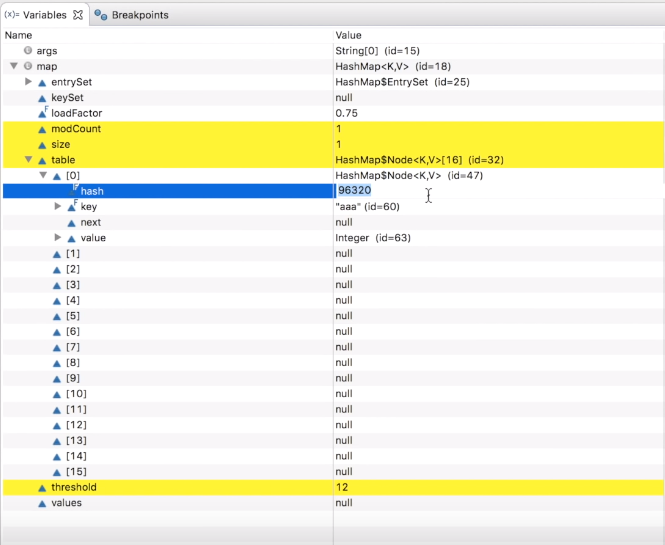
**->HashMap is non-synchronized (not thread safe)**

->internally HashMap follows hashing policy

A hash function is a mathematical function that converts an input value into a compressed numerical value – a hash or hash value. Basically, it's a processing unit that takes in data of arbitrary length and gives you the output of a fixed length – the hash value

->when we create any map it creates it creats a table to store key,hashcode,value and next link (it is called linkedlist or nodes)





->for putting elements it goes for indexing policy based on calculation of hash & (n-1) **hashcode%number of buckets** (initially map has 16 buckets/index to store

\*\*if the moduler number is 5 ->it goes for 5 index

hash = 10

hash % n = 10 % 8 = 2

(n - 1) & hash = 7 & 10 = 0 1 1 1 & 1 0 1 0 = 0 0 1 0 = 2

{binary of 7:

7-4=3 3-2=1 1-1=0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 32 | 16 | 8 | 4 | 2 | 1 |
|  |  | 0 | 1 | 1 | 1 |

Binary of 10:

10-8=2 2-2=0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 32 | 16 | 8 | 4 | 2 | 1 |
|  |  | 1 | 0 | 1 | 0 |

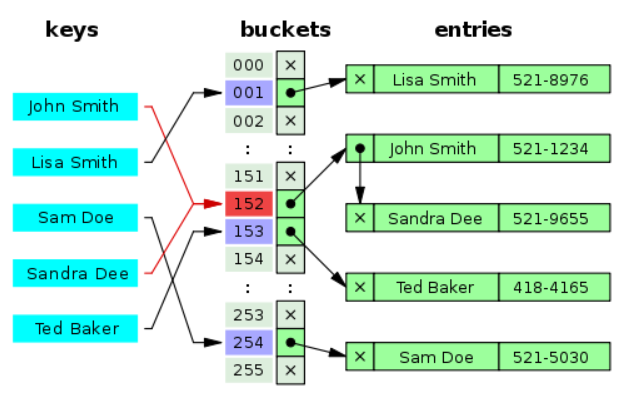
7&10: (How does it work:1&1=1 1&0=0 0&1=0 0&0=0)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 7----> | 0 | 1 | 1 | 1 |
| 10--> | 1 | 0 | 1 | 0 |
| 7&10 | 0 | 0 | 1 | 0 |
| Converting back |  |  | 2 |  |

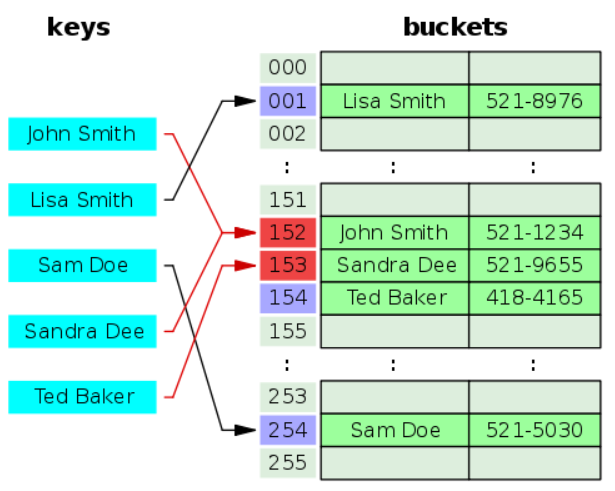
**->hash-collision:** A collision will occur when two different keys have the same hashCode, which can happen because two unequal objects in Java can have the same hashCode

->**hash-collision is handled** by additional data structure inside the bucket. -> HashMap creates linked list or tree and all keys with the same hash-code are added there.

Separate chaining



Open addressing



Coalesced hashing

Cuckoo hashing

Robin Hood hashing

2-choice hashing

Hopscotch hashing

**Dynamic resizing:**

Resizing by copying all entries

Incremental resizing

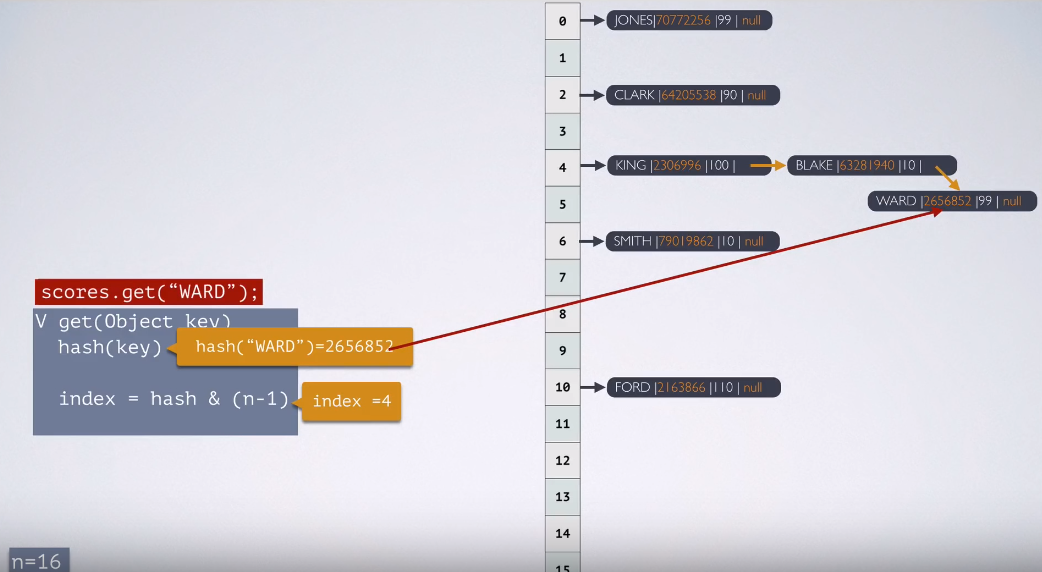
Monotonic keys

* How does get() works

Same way hascode is generated->index choosing->cross match with existing hashcode

\*(if matched)->equal() is used (key already in bucket and key is passing with get())->if returns true then it returns the value for the key

\*if not matched then goes for next linked of the same index and proceed in same manner



Additional:

Different Types of Iterator

<https://www.geeksforgeeks.org/iterators-in-java/>