INTERVIEWBIT LINK: https://www.interviewbit.com/java-collections-interview-questions/amp/

Collection

In [**Java**](https://www.interviewbit.com/online-java-compiler/), a collection is a framework that provides an architecture for storing and manipulating a collection of objects.

two bases interfaces of Collection

Collection

Map

Differences TreeSet and HashSet

HashSet :  For search, insert, and remove operations, it takes constant time on average. TreeSet is slower than HashSet

TreeSet: For search, insert, and delete, TreeSet takes O(Log n), which is higher than HashSet. some other methods are implemented floor(), ceiling() and others

ORDERS

HashSet: the elements are not ordered.

TreeSet: keeps the order in a sorted order defined by the Comparable or Comparator methods. By default it is ascending order.

Null values

HashSet: null objects are allowed in hashset.

TreeSet: does not allow null objects throws a nullpointerexception.

Comparison

HashSet: compares two objects in a Set and detects duplicates using the equals() method.

TreeSet: TreeSet employs the compareTo() method. If equals() and compareTo() are not consistent, that is, if equals() returns true for two equal objects but compareTo() returns zero, the contract of the Set interface will be broken, allowing duplicates in Set implementations like TreeSet.

Difference between Collection and collections in the context of Java

Collection: In the java.util.package, there is an interface called a collection. It's used to represent a collection of separate objects as a single entity. The collection framework's root interface is referred to as the collection. It has a number of classes and interfaces for representing a collection of individual objects as a single unit. The key sub-interfaces of the collection interface are List, Set, and Queue. Although the map interface is part of the Java collection framework, it does not inherit the interface's collection. The Collection interface's most significant functions are add(), remove(), clear(), size(), and contains().

Collections: The java.util.package has a utility class called Collections. It defines various utility methods for working with collections, such as sorting and searching. All of the methods are static. These techniques give developers much-needed convenience, allowing them to interact with Collection Framework more successfully. It provides methods like sort() to sort the collection elements in the normal sorting order, and min() and max() to get the minimum and maximum value in the collection elements, respectively.

Advantages of the Collection framework

**Consistent API**: The API has a core set of interfaces like Collection, Set, List, or Map, and all the classes (ArrayList, LinkedList, Vector, and so on) that implement these interfaces have some common set of methods.

**Improves program speed and quality** by offering high-performance implementations of useful data structures and algorithms, as the programmer does not have to worry about the optimum implementation of a certain data structure in this scenario. They can simply use the best implementation to improve the performance of their program significantly.

Differences between LinkedList and ArrayList

ArrayList:

* The elements of this class are stored in a dynamic array.
* The elements of this class are stored in a dynamic array.
* Because of the internal implementation, manipulating an ArrayList takes longer. Internally, the array is scanned and the memory bits are shifted whenever we remove an element.
* This class is more useful when the application requires data storage and access.

LinkedList:

* The elements of this class are stored in a doubly-linked list.
* The List and Deque interfaces are both implemented by this class.
* Because there is no concept of changing memory bits in a doubly-linked list, manipulating it takes less time than manipulating an ArrayList. The reference link is changed after traversing the list.
* This class is more useful when the application requires data manipulation.

Differences between List and Set

The List interface is used to keep track of an ordered collection. It is the Collection's child interface. It is an ordered collection of objects that allows for the storage of duplicate values. The insertion order is preserved in a list, which enables positional access and element insertion.

The set interface is part of java.util package and extends the Collection interface. It is an unordered collection of objects in which duplicate values cannot be stored. This interface inherits the Collection interface's methods and adds a feature that prevents duplicate elements from being inserted.

| **Set** | **List** |
| --- | --- |
| It is an unordered sequence. | It is an ordered sequence. |
| Duplicate elements are not permitted in Set. | Duplicate elements are allowed in the list |
| Access to items from a certain position is not permitted. | Elements can be accessed based on their position. |
| A null element can only be stored once. | It is possible to store several null elements. |

Differences between Iterator and ListIterator

| **Iterator** | **ListIterator** |
| --- | --- |
| Only has the ability to traverse components in a Collection in a forward direction. | In both forward and backward orientations, can traverse components in a Collection. |
| Iterators cannot be used to obtain indexes. | It offers methods to get element indexes at any time while traversing List, such as next Index() and previous Index(). |
| It aids in the traversal of Maps, Lists, and Sets. | Only List may be traversed, not the other two. |
| It throws a Concurrent Modification Exception since it can't add elements. | At any time, you can quickly add elements to a collection. |
| next(), remove(), and has Next are some of the Iterator's functions (). | next(), previous(), has Next(), has Previous(), and add() are some of the List Iterator's methods |

Differences between Array and Collection

Array and Collection are equivalent in terms of storing object references and manipulating data, but they differ in a number of ways.

| **Array** | **Collection** |
| --- | --- |
| Arrays have a set size, which means that once we build one, we can't change it to meet our needs. | Collection are naturally grow-able and can be customized to meet our needs. We can change its size as per our requirement. |
| When it comes to performance, Arrays are the preferred to Collection. | Considering performance, Collection are not preferred to Arrays. |
| Only homogeneous data type elements can be stored in arrays. | Both homogeneous and heterogeneous components can be stored in a collection. |
| Because arrays have no underlying data structure, there is no ready-made method support. | Any collection class is built on a standard data structure, and so there is ready-made method support for every demand as a performance. These methods can be used directly, and we are not responsible for their implementation. |
| Objects and primitives can both be stored in arrays. | Only object types can be stored in a collection. |
| When it comes to memory, Arrays are not preferred to Collection. | Considering memory, Collection are preferred to Arrays. |

Differences between Set and Map

| **Set** | **Map** |
| --- | --- |
| It cannot have values that are repeated. It is not possible to add the same elements to a set. Only the unique value is stored in each class that implements the Set interface. | It is possible for different keys to have the same value. The map has a unique key and values that are repeated. |
| Using the keyset() and entryset() methods, we can quickly iterate the Set items. | It is not possible to iterate across map elements. To iterate the elements, we must convert Map to Set. |
| The Set interface does not keep track of insertion order. Some of its classes, such as LinkedHashSet, however, keep the insertion order. | The Map does not keep track of the insertion sequence. Some Map classes, such as TreeMap and LinkedHashMap, do the same thing. |

Differences between HashSet and HashMap

**HashSet** is a Set Interface implementation that does not allow duplicate values. The essential point is that objects stored in HashSet must override equals() and hashCode() methods to ensure that no duplicate values are stored in our set.

**HashMap** is a Map Interface implementation that maps a key to a value. In a map, duplicate keys are not permitted.

| **HashSet** | **HashMap** |
| --- | --- |
| It implements the Set Interface. | It implements the Map Interface. |
| It does not allow duplicate values. | The key needs to be unique while two different keys can have the same value. |
| While adding an element it requires only one object as a parameter. | While adding an entry, it requires two object values, the **Key**and the **Value**as the parameter. |
| Internally, HashSet uses HashMap to add entries. The key K in a HashSet is the argument supplied in the add(Object) method. For each value supplied in the add(Object) method, Java assigns a dummy value. | There is no concept of duplicate values. |
| It is slower than HashMap. | It is faster than HashSet. |
| It uses the add() method for adding elements. | It uses the put() method for adding data elements. |

Differentiate between Array and ArrayList

ArrayList is not a fixed-size data structure, but Array is. When creating an ArrayList object, there is no need to provide its size.

Arrays can include both primitive data types and class objects, depending on the array's definition. ArrayList, on the other hand, only accepts object entries and not primitive data types.

MEDIUM LINK: <https://medium.com/@harendrakumarrajpoot5/top-50-java-collections-interview-questions-you-need-to-know-e55fcdc8dbfb>

1. collections API – minden benne van
2. utána elmondom mi van benne és miért van alatta list, queue, set
3. utána elmondani részletesen hogy mi van benne
4. mikor melyiket használjuk és miért
5. Map, implementációk mi mit tartalmaz és miért

Arraylist:

* dinamikus tömb
* gyors hozzáférést és indexelési műveleteket tesz lehetővé
* alkalmazása gyakori az olvasást igénylő műveleteknél
* gyenge a beillesztés és törlés során, különösen a lista közepén

LinkedList:

* állandó idejü beszúrásokat és törléseket tesz lehetővé, függetlenül a lista méretétől
* gyakori alkalmazása amikor módosítunk (data manipulation)
* viszont nagyobb a memóriaterhelése
* az elemek indexen keresztüli eléréséhez a lista bejárása szükséges, ami lassan véletlen hozzáférési időt eredményez.

ArrayList:

* default size is 10
* dynamic array
* fast access to find an element (constant time always doesnt matter from where)
* behind the scenes create a new array and put the element for the related index not great for data manipulation linkedlist better

Linkedlist:

* the elements has a reference to the next element and that element has a reference to the previous element, thats why we called doubly list
* follow the reference pointers to for example get the 400th element from a list (time consuming) slower than arraylist
* great for data manipulation

in Arraylist get an element is much faster than at LinkedList, inserting or removing is much better to use LinkedList (data manipulation)

QUEUE

egyik legfontosabb : FIFO

egy elemet csak a végére vehetek fel, és egy elemet pedig csak az elejéről törölhetek

SET

HashSet: egy osztály ami implementélja a set interface-t

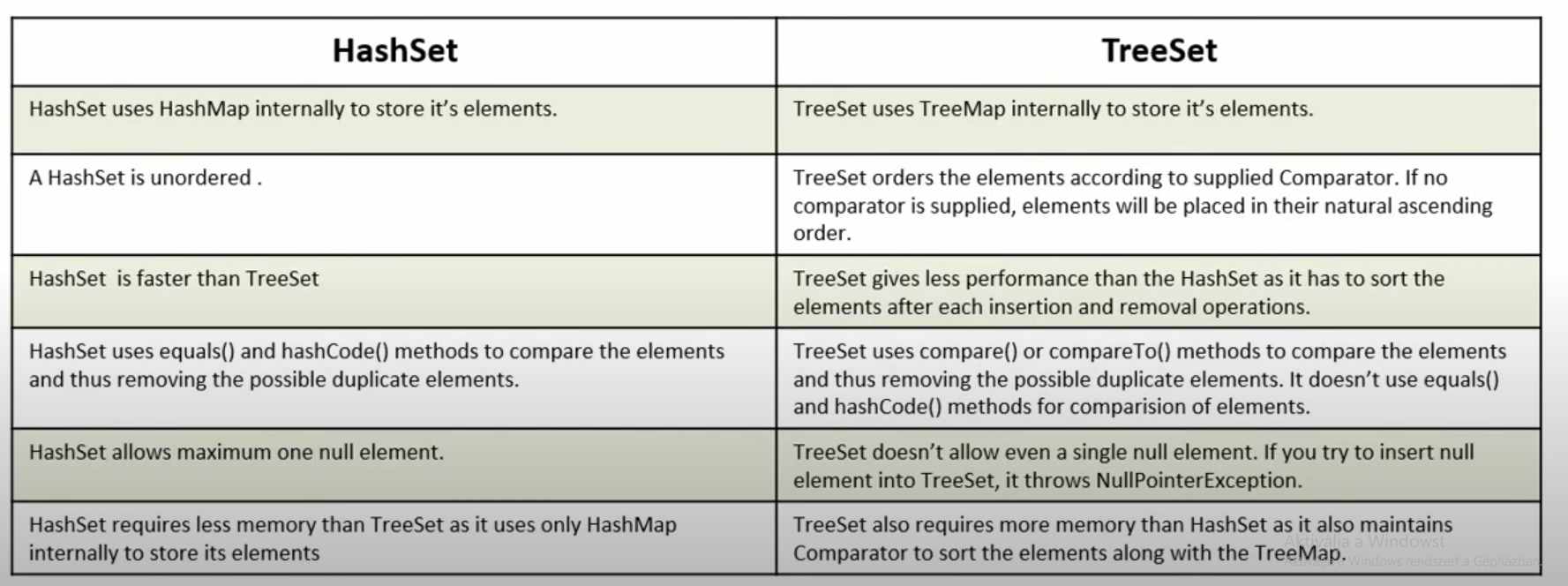
rendezetlen (unordered)

Hash alapú halmaz ami azt jelenti, hogy hashcode es az equals alapján dönti el hogy benne van e már a setben vagy sem

pl (Kati, 22 – 2x inicializáltuk és benne van 2x de a hashcode és equals override esetén csak 1x kerül bele)

TreeSet: SortedSet-nek egy konkrét implementációja, sorrendben van tehát rendezett

Comparator interface-t implementálni kell, ez segit az összehasonlitásban és a rendezésben



HashSet or TreeSet

1) First major difference between HashSet and TreeSet is performance. HashSet is faster than TreeSet and should be preferred choice if sorting of element is not required.

2) Second difference between HashSet and TreeSet is that HashSet allows null object but TreeSet doesn't allow null Object and throw NullPointerException, Why, because TreeSet uses compareTo() method to compare keys and compareTo() will throw java.lang.NullPointerException.

3) Another significant difference between HashSet and TreeSet is that , HashSet is backed by HashMap while TreeSet is backed by NavigableMap in Java.

4) One more difference between HashSet and TreeSet which is worth remembering is that HashSet uses equals() method to compare two object in Set and for detecting duplicates while TreeSet uses compareTo() method for same purpose. if equals() and compareTo() are not consistent, i.e. for two equal object equals should return true while compareTo() should return zero, than it will break contract of Set interface and will allow duplicates in Set implementations like TreeSet

5) Now most important difference between HashSet and TreeSet is ordering. HashSet doesn't guaranteed any order while TreeSet maintains objects in Sorted order defined by either Comparable or Comparator method in Java.

6) TreeSet does not allow to insert Heterogeneous objects. It will throw classCastException at Runtime if trying to add hetrogeneous objects, whereas HashSet allows hetrogeneous objects.

QUEUE

FIFO data structure. first in first out

head and tail (első utolsó, változik)

remove from head and add to tail

MAP

HashMap:

* Map interface implementációja
* null értéket felvehet
* unordered, nem garantálja az elemek sorrendjét
* nem szinkronizált, tehát nem szál-biztos (thread-safe)
* gyors keresést, törlést és beszúrást biztosít
* akkor érdemes használni ha a gyorsaság fontos és nem számit a sorrendű

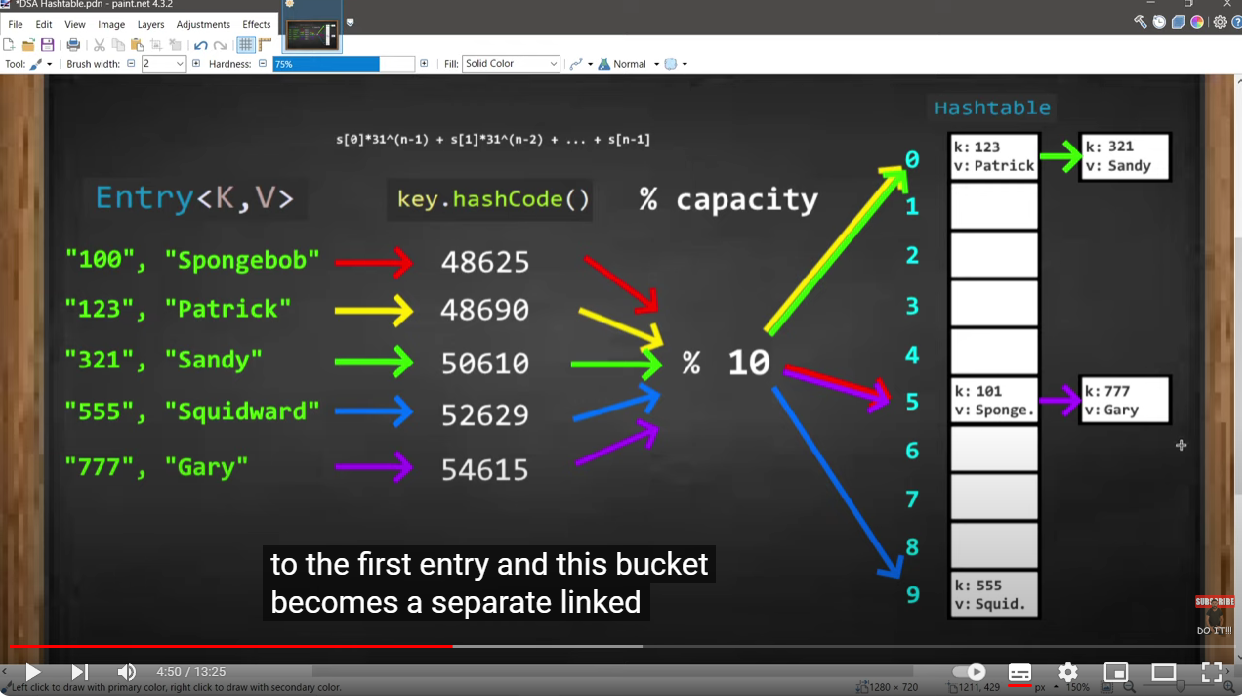
TreeMap:

* null kulcs nem lehet, de a value lehet null érték
* ordered, vagy természetes vagy a Comparator alapján rendezi az elemeket
* nem szálbiztos (thread-safe)
* akkor használjuk ha fontos a sorrendiség
* lassabb mint a hashmap

HashTable:

* null kulcs és null value nem lehet
* szálbiztos , thread-safe!!!
* elemek sorrendje nem garantált

müködése: capacity 10% a maradék alapján dönti el hogy hanyas indexre teszi be az elemet 0-9ig ha két kulcsnak is például 0a a vége akkor oda teszi be mindkettőt és készít egy linkedlistet ami mutat a következő elemre ha több van ott tárolva.



STACK

úgy képzeljük el hogy egymás felett vannak és ha pl kikérjük az elsőt a .pop()al akkor ki kerül a stackből az első elem.

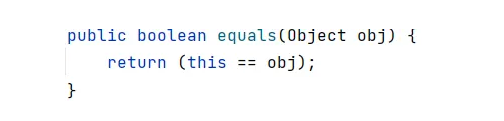
.peek() ugyanaz mint a pop csak nem tűnik el

amit utoljára teszek be az lesz az első elem

hashcode() and equals()

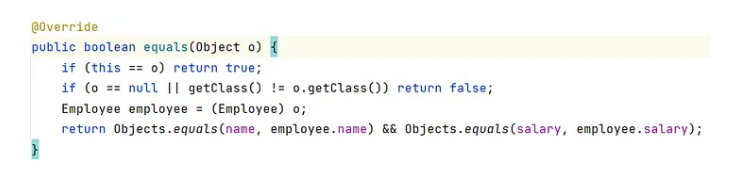
Object parent class of all classes provides two important method equals() and hashcode() for comparing objects.

equals() : The default implementation of the equals() method checks for referential equality or simply said checks if two objects’ references point to the same memory address.



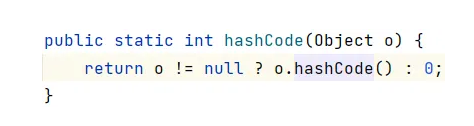
Overriding equals()

Let us override the **equals()**method so that it not only considers referential equality but also the value of the attributes for comparison. Now if we check the standard output we can see **true**being returned.



hashcode()

A hash code is an integer value of an object. This hash code is unique to an object and generated using a hashing function. The method definition for **hashCode()** is given below.



Overriding hashcode()

Let us now override the **hashCode()** method in our code and as shown below the overridden implementation, internally calls the APIs to generate a hash code unique to the object based on the number of attributes provided to create the hash value.



Conclusion

equals() and hashCode() must go hand in hand.

If two objects are equal, they must have the same hash code.

If two objects have the same hash code, it does not necessarily mean they are equal.

Overriding the equals() method alone would fail with hashing data structures like Map, Set, etc.

Overriding the hashCode() method alone would not help in comparing objects.

HashMap

Hashing: A hashing azt jelenti hogy egy objektumot integer formába konvertálunk a hashcode() metódussal.

HashMap: Az adatokat key es value párokban tárolja.

Amikor egy kulcsot és értéket helyezünk el a HashMap-ben, először az equals metódus ellenőrzi, hogy mindkét objektum egyenlő-e, majd a hashing segítségével meghatározza, melyik bucket-be kell helyezni a kulcsot és értéket. Ha mindkét objektum egyenlő, a már meglévő objektum felülíródik a bucket-ben.

 equals(): Ellenőrzi két objektum egyenlőségét. Összehasonlítja a kulcsokat, hogy egyenlőek-e vagy sem. Az Object osztály egyik metódusa, és felülírható. Ha felülírjuk az equals() metódust, kötelező felülírni a hashCode() metódust is.

 hashCode(): Az Object osztály metódusa, amely integer formában visszaadja az objektum memóriacímet. Az ebből a metódusból kapott érték a bucket száma lesz.