NPTEL

noc24_cs43

Programming in Java

Live Interaction Session 10: 02nd Apr 2024

Introduction to the Collections Framework

A Collection

A collection is just an object that represents a group of objects.

In general, the group of objects have some relationship to each other.

Collection objects, include arrays, lists, vectors, sets, queues, tables, dictionaries, and maps.

These are differentiated by the way they store the objects in memory, how objects are retrieved and ordered, and whether nulls and duplicate entries are permitted.

A Collections Framework

Oracle's Java documentation describes it's collections framework as:

"A unified architecture for representing and manipulating collections, enabling collections to be manipulated independently of implementation details.”

What's in the framework, what's not?

Strictly speaking, arrays and the array utilities in the java.util.Arrays class are not considered part of this framework.

All collection objects implement the Collection interface, with the exception of maps.

The Collection Interface

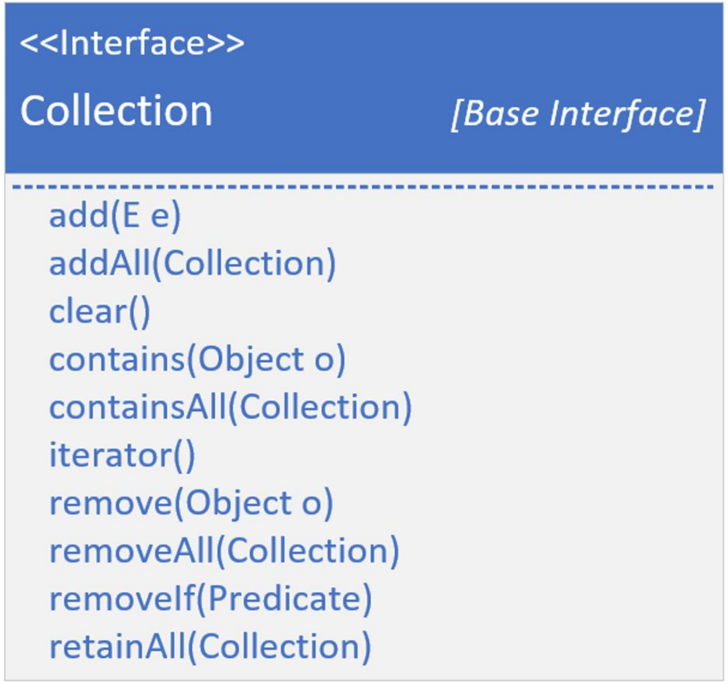
The Collection interface is the root of the collection hierarchy.

Like most roots in software hierarchies, it's an abstract representation of the behavior you'd need, for managing a group of objects.

This type is typically used to **pass collections** around, and manipulate them where **maximum generality** is desired.

Remember, the interface let's us describe objects by what they can do, rather than what they really look like, or how they're ultimately constructed.

If you look at the methods on this interface, you can see the basic operations a collection of any shape, or type, would need to support.



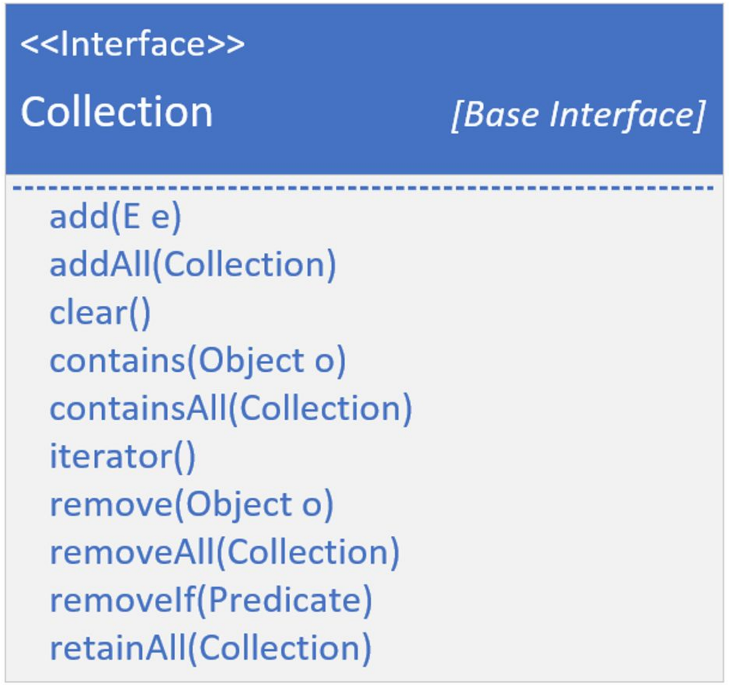
The Collection Interface

We've already looked at every one of these operations, in our study of ArrayLists and LinkedLists.

When managing a group, you'll be adding and removing elements, checking if an element is in the group, and iterating through the elements.

There are some others, but these are the ones that describe nearly everything you'd want to do to manage a group.

Java uses the term **Element** for a member of the group being managed.

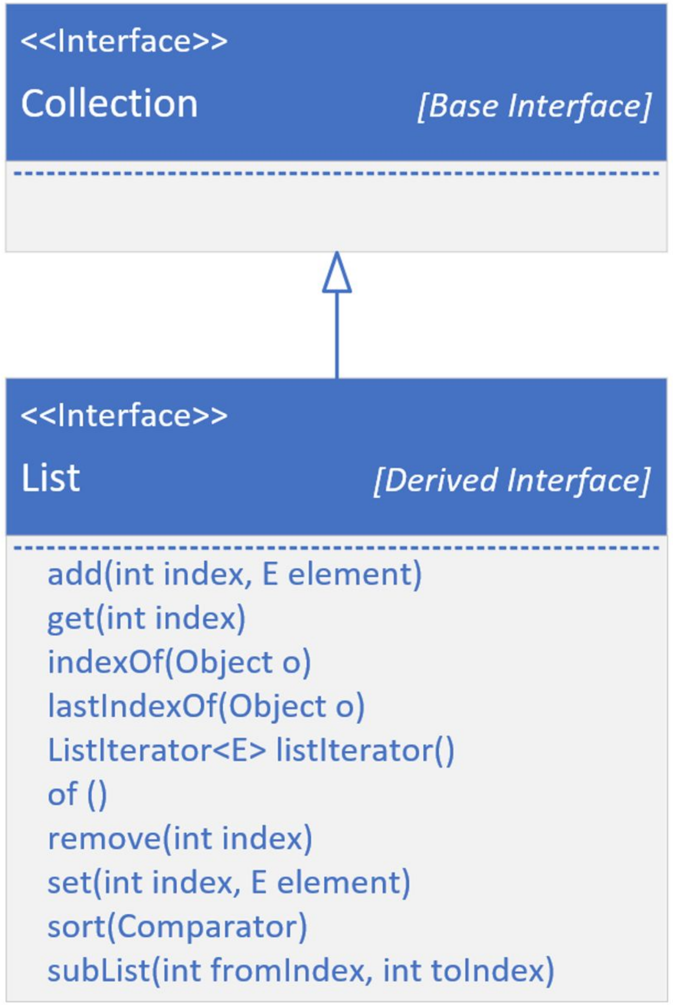


The Collection Interface

The image shows the List interface extending Collection.

For simplicity, I'm not showing the Collection methods that I showed on the previous slide.

I'm only showing additional methods specifically declared on the List interface.



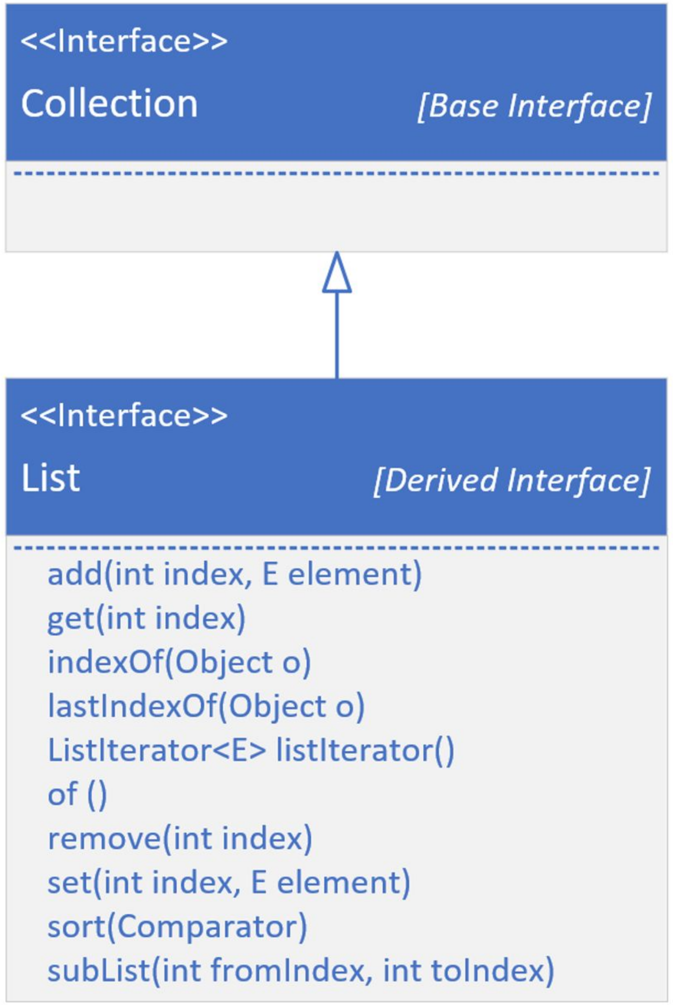
The Collection Interface

We covered most of these methods, but I wanted you to see here, that most of these are dealing with an index.

A list can be either indexed, as an ArrayList, or not, like a LinkedList, but a LinkedList is implemented to support all of these methods as well.

Derived interfaces may have specific ways to add, remove, get, and sort elements for their specific type of collection, in addition to those defined on the Collection Interface itself.

Now, let's look at the big picture of interfaces, and some specific implementations.

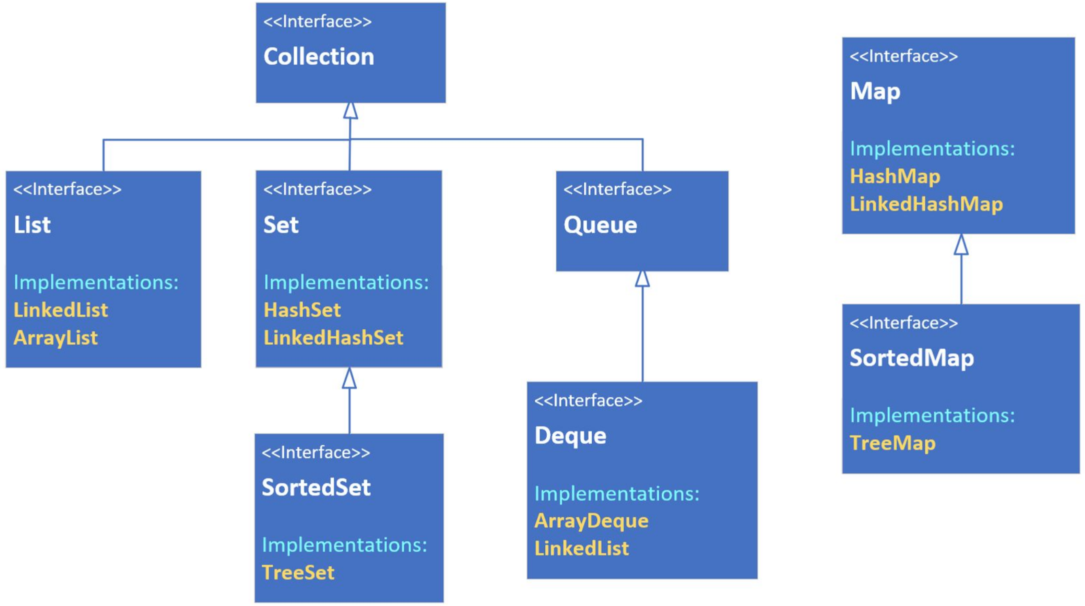


Collections - The Big Picture

This slide shows the interface hierarchy.

It's also showing the implementations or concrete classes, that implement these interfaces, in yellow font.

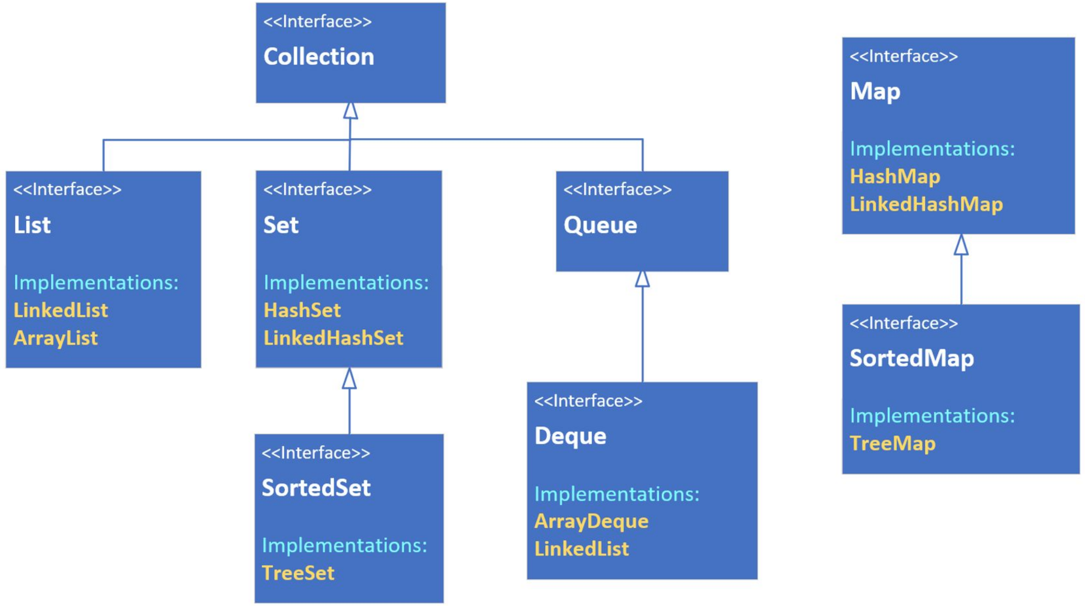
Notice that Map does not extend Collection, although still part of this framework.



Collections - The Big Picture

Maps are uniquely different, which we will see later.

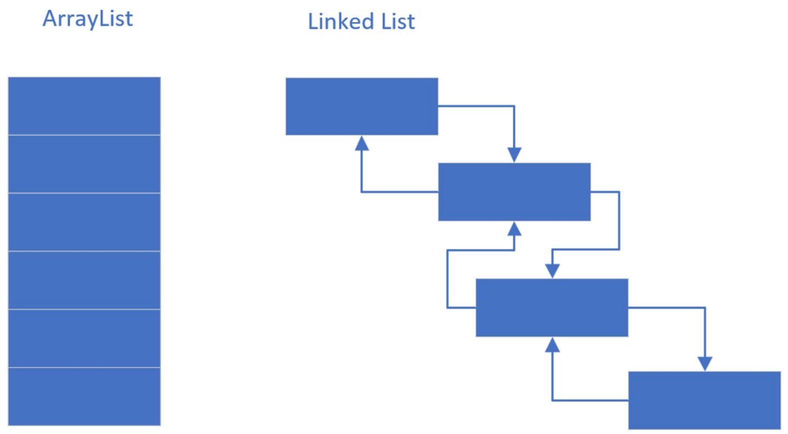
You can see that LinkedList implements both List and Deque.



The List

A List is An ordered collection (also known as a sequence).

These can be sequenced in memory like an ArrayList, or maintain links to the next and previous values, as a LinkedList.

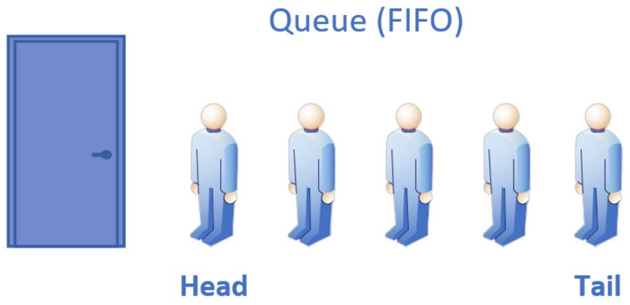


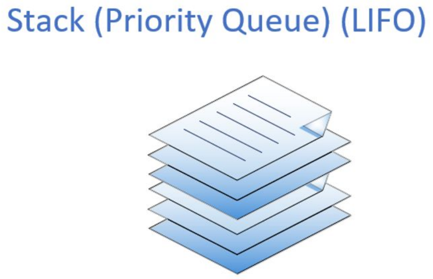
The Queue

A Queue is a collection designed for holding elements prior to processing, in other words the processing order matters, so the first and last positions, or the head and tail, are prioritized.

Most often these may be implemented as First In, First Out (FIFO), but can be implemented like a Stack, as Last In First Out (LIFO) which we've discussed.

Remember a Deque supports both.





The Set

A Set is a collection conceptually based off of a mathematical set.

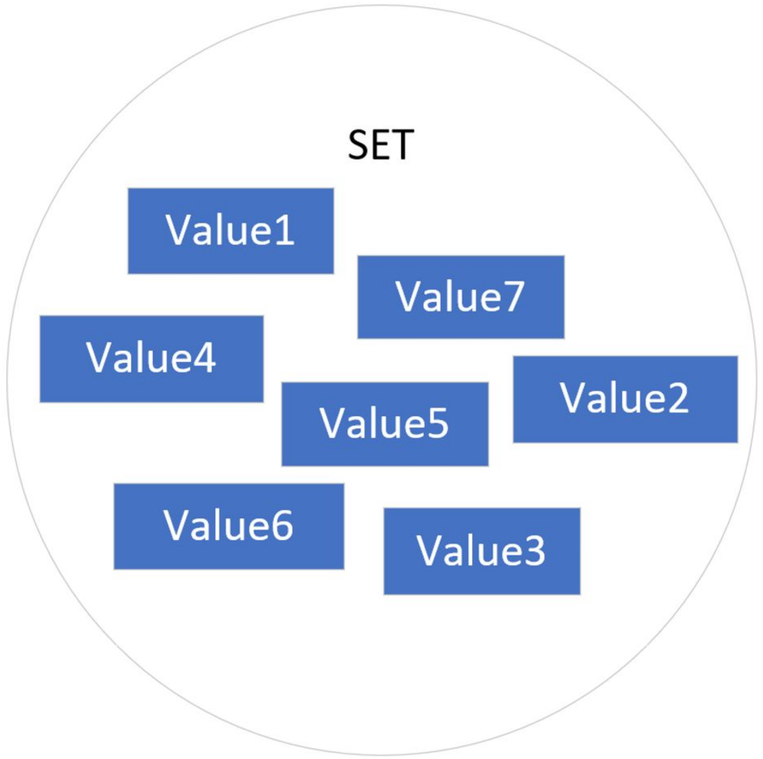
Importantly, it contains no duplicate elements, and isn't naturally sequenced or ordered.

You can think of a set as a kind of penned in chaotic grouping of objects.

Java has three implementations, the HashSet, the TreeSet and the LinkedHashSet.

These are distinguished by the underlying way they store the elements in the set.

A Sorted Set is a set that provides a total ordering of the elements.



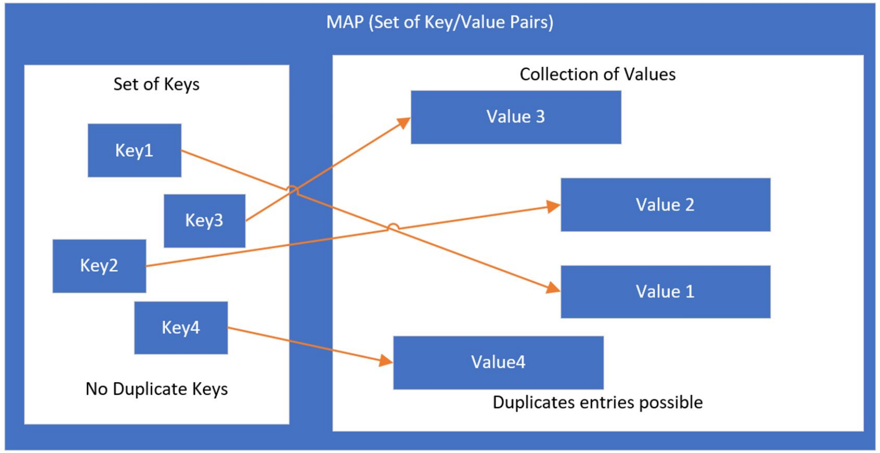
The Map

A Map is a collection that stores key and value pairs.

The keys are a set, and the values are a separate collection, where the key keeps a reference to a value.

Keys need to be unique, but values don't.

Elements in a tree are stored in a key value Node, also called an Entry.



What's a Collections Class?

It's important to understand that the Collections class is not the Collections Framework.

The framework contains many interfaces and implemented classes, as well as helper classes, which this Collections class is just one example.

At one time, Java had interfaces, but no support for static or default methods on them, so useful methods were packaged in these helper classes.

Some of these methods have since been implemented on the interfaces themselves, but there's still some functionality on the Collections class you might find useful.

Understanding the importance of the hash code

HashSet and HashMap, are based on the hash codes of objects.

Since sets are unique because they don't support duplicates, adding an element always incurs the cost of first checking for a match.

If your set is large or very large, this becomes a costly operation, O(n), or linear time, if you remember the Big O notations I covered previously.

A mechanism to reduce this cost, is introduced by something called hashing.

If we created two buckets of elements, and the element could consistently identify which bucket it was stored in, then the lookup could be reduced by half.

If we created four buckets, we could reduce the cost by a quarter.

Understanding the importance of the hash code

A hashed collection will optimally create a limited set of buckets, to provide an even distribution of the objects across the buckets in a full set.

A hash code can be any valid integer, so it could be one of 4.2 billion valid numbers.

If your collection only contains 100,000 elements, you don't want to back it with a storage mechanism of 4 billion possible placeholders.

And you don't want to have to iterate through 100,000 elements one at a time to find a match or a duplicate.

Understanding the importance of the hash code

A hashing mechanism will take an integer hash code, and a capacity declaration which specifies the number of buckets to distribute the objects over.

It then translates the range of hash codes into a range of bucket identifiers.

Hashed implementations use a combination of the hash code and other means, to provide the most efficient bucketing system, to achieve this desired uniform distribution of the objects.

Hashing starts with understanding equality

To understand hashing in Java, first we need to understand the equality of objects.

There are two methods on java.util.Object, that all objects inherit.

These are equals, and hashCode, and these method signatures from Object.

Testing for equality	**The hashcode method**
`public boolean equals(Object obj)`	`public int hashCode()`

The equals method on Object

The implementation of equals on Object is shown here.

It simply returns this == obj.

```java
public boolean equals(Object obj) {
    return (this == obj);
}
```

Do you remember what == means for Objects?

Do you remember what == means for objects?

It means two variables have the **same reference to a single object in memory**.

Because both references are pointing to the same object, then this is obviously a good equality test.

Equality of Objects

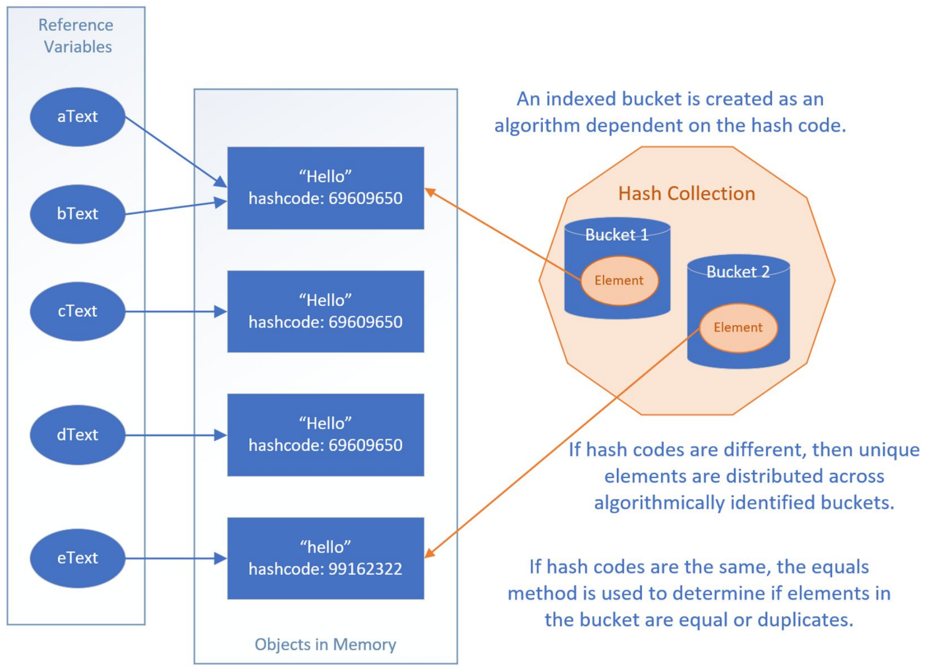
Objects can be considered equal in other instances as well, if their attribute values are equal, for example.

The String class overrides this method, so that it compares all the characters in each String, to confirm that two Strings are equal.

The visual representation of the code

Our code set up five String reference variables, but two of these, referenced the same string object in memory, as shown here with aText and bText pointing to the same string instance.

When we passed our list of five strings to the HashSet, it added only unique instances to its collection.

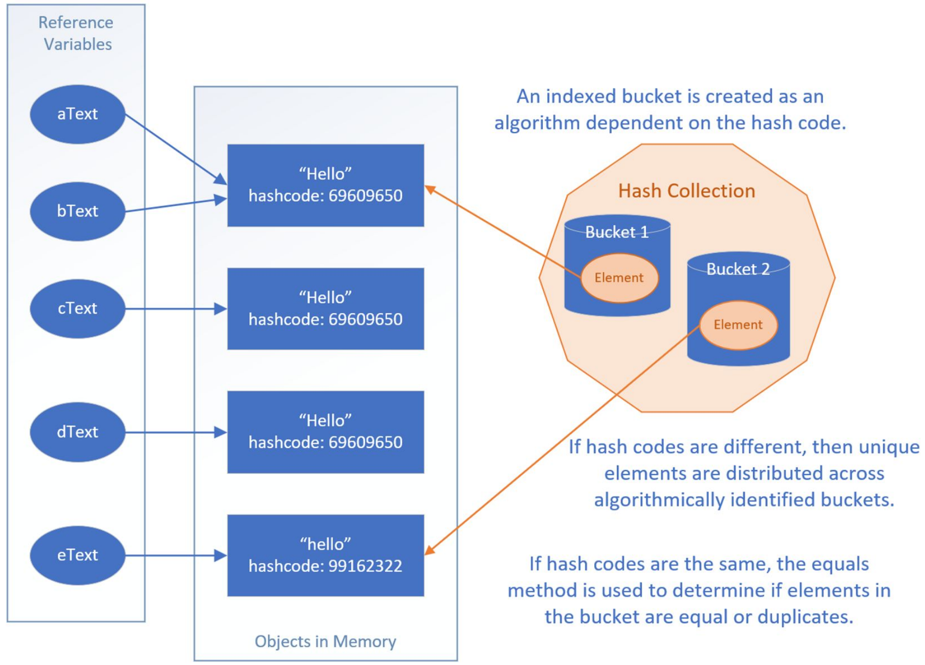


Reference Variables
aText
bText
cText
dText
eText
"Hello" hashcode: 69609650
"Hello" hashcode: 69609650
"Hello" hashcode: 69609650
"hello" hashcode: 99162322
Objects in Memory
An indexed bucket is created as an algorithm dependent on the hash code.
Hash Collection
Bucket 1
Element
Bucket 2
Element
If hash codes are different, then unique elements are distributed across algorithmically identified buckets.
If hash codes are the same, the equals method is used to determine if elements in the bucket are equal or duplicates.

The visual representation of the code

It locates elements to match, by first deriving which bucket to look through, based on the hash code.

It then compares those elements to the next element to be added, with other elements in that bucket, using the equals method.


Reference Variables
aText
bText
cText
dText
eText
"Hello" hashcode: 69609650
"Hello" hashcode: 69609650
"Hello" hashcode: 69609650
"Hello" hashcode: 69609650
"hello" hashcode: 99162322
Objects in Memory
An indexed bucket is created as an algorithm dependent on the hash code.
Hash Collection
Bucket 1
Element
Bucket 2
Element
If hash codes are different, then unique elements are distributed across algorithmically identified buckets.
If hash codes are the same, the equals method is used to determine if elements in the bucket are equal or duplicates.

Creating the hashCode method

You could create your own, but your code should stick to the following rules.

- It should be very fast to compute.
- It should produce a consistent result each time it's called. For example, you wouldn't want to use a random number generator, or a date time based algorithm that would return a different value each time the method is called.
- Objects that are considered equal should produce the same hashCode.
- Values used in the calculation should not be mutable.

Creating the hashCode method

It is common practice to include a small prime number as a multiplicative factor (although some non-prime numbers also provide good distributions).

This helps ensure a more even distribution for the bucket identifier algorithm, especially if your data might exhibit clustering in some way.

IntelliJ and other code generation tools use 31, but other good options could be 29, 33 (not prime but shown to have good results), 37, 43 and so on.

You want to avoid the single digit primes, because more numbers will be divisible by those, and may not produce the even distribution that will lend itself to improved performance.

The Set

A Set is not implicitly ordered.

A Set contains no duplicates.

A Set may contain a single null element.

Sets can be useful because operations on them are very fast.

Set Methods

The set interface defines the basic methods **add, remove** and **clear**, to maintain the items in the set.

We can also check if a specific item is in the set using the contains method.

Interestingly enough, there's no way to retrieve an item from a set.

You can check if something exists, using **contains**, and you can iterate over all the elements in the set, but attempting to get the 10th element, for example, from a set isn't possible, with a single method.

The HashSet class

The best-performing implementation of the Set interface is the **HashSet** class.

This class uses hashing mechanisms to store the items.

Oracle describes this class as offering constant time performance for the basic operations (add, remove, contains and size).

Constant time has the Big O Notation O(1).

The HashSet actually uses a HashMap in it's own implementation, as of JDK 8.